conference

proceedings

6th USENIX Symposium on Networked Systems Design and Implementation

Boston, MA, USA

April 22–24, 2009

Sponsored by
The USENIX Association

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGCOMM
and ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

Thanks to Our Sponsors

Platinum Sponsor

Infosys®
POWERED BY INTELLECT
DRIVEN BY VALUES

Bronze Sponsors

cisco™ Google™ hp® invent intel® Microsoft® Research

Thanks to Our Media Sponsors

ACM *Queue*

Addison-Wesley Professional/
Prentice Hall Professional/
Cisco Press

Distributed Management
Task Force, Inc.

InfoSec News

Linux Gazette

Linux Journal

Linux Pro Magazine

LXer.com

Network World ITRoadmap

Toolbox.com

ISBN: 978-1-931971-67-6

USENIX Association

Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI '09)

April 22–24, 2009
Boston, MA, USA

Conference Organizers

External Reviewers

6th USENIX Symposium on Networked Systems Design and Implementation
April 22–24, 2009
Boston, MA , USA

Index of Authors

Message from the Program Co-Chairs

NSDI '09 carries on the conference's tradition of presenting excellent and innovative work in the area of networked systems. We continue to take a broad view of that charter, selecting papers from across the range of the USENIX, SIGCOMM, and SIGOPS communities, rather than their intersection. The result is a strong program with a broad set of papers addressing topics from trust and privacy to storage systems, from wireless networks to content distribution, from Byzantine fault tolerance to wide-area services, from green networked systems to Internet routing, and from monitoring, understanding, and debugging what exists today to designing the networked systems of tomorrow.

We received 163 paper submissions. All submissions were reviewed by several members of the program committee and selected external reviewers. Papers received at least three reviews, with some papers receiving as many as seven. These written reviews laid the groundwork for the program committee meeting in Chicago, Illinois, on December 15. These discussions led to a final program of 32 accepted papers. Because of the special role conferences play in our field, all papers were shepherded by a program committee member and, where necessary, granted extra pages for a fuller discussion.

We are grateful to everyone whose hard work makes this conference possible. Most of all, we are grateful to all of the authors who submitted their work to this conference. We thank the program committee for their dedication and hard work in reviewing papers and participating in the extensive discussions at the PC meeting. We thank our external reviewers for lending their expertise on short notice. We are grateful to the numerous conference sponsors for their support. Thanks to the USENIX staff for handling the conference logistics, marketing, and proceedings publication; it is a pleasure to work with them. We extend special thanks to Eddie Kohler for providing and supporting his terrific HotCRP reviewing system. Finally, we thank the NSDI '09 attendees and future readers of these papers: in the end, it is your interest in this work that makes all of these efforts worthwhile.

We look forward to seeing you in Boston!

Jennifer Rexford, Princeton University
Emin Gün Sirer, Cornell University
NSDI '09 Program Co-Chairs

TrInc: Small Trusted Hardware for Large Distributed Systems

Dave Levin, *University of Maryland*
John R. Douceur, *Microsoft Research*
Jacob R. Lorch, *Microsoft Research*
Thomas Moscibroda, *Microsoft Research*

Abstract

A simple yet remarkably powerful tool of selfish and malicious participants in a distributed system is “equivocation”: making conflicting statements to others. We present TrInc, a small, trusted component that combats equivocation in large, distributed systems. Consisting fundamentally of only a non-decreasing counter and a key, TrInc provides a new primitive: unique, once-in-a-lifetime attestations.

We show that TrInc is practical, versatile, and easily applicable to a wide range of distributed systems. Its deployment is viable because it is simple and because its fundamental components—a trusted counter and a key—are already deployed in many new personal computers today. We demonstrate TrInc’s versatility with three detailed case studies: attested append-only memory (A2M), PeerReview, and BitTorrent.

We have implemented TrInc and our three case studies using real, currently available trusted hardware. Our evaluation shows that TrInc eliminates most of the trusted storage needed to implement A2M, significantly reduces communication overhead in PeerReview, and solves an open incentives issue in BitTorrent. Microbenchmarks of our TrInc implementation indicate directions for the design of future trusted hardware.

1 Introduction

As wide-area systems grow in scale, so do their exposure to threats. Much of the interesting distributed-systems research of the past decade has focused on the issues of security and adversarial incentive that are inherent to large-scale systems. This research has addressed a wide range of applications, including storage [2, 16, 19, 22, 28], communication [4, 45, 30], databases [40], content distribution [15, 24, 32, 36], grid computation [12], and games [3, 10], in addition to generic infrastructure [1, 5, 9, 18, 23, 43]. Virtually all of this work shares a common supposition, namely that the individual components in the system are completely untrusted.

Recently, the necessity of this supposition has been called into question. The Attested Append-only Memory (A2M) system by Chun et al. [7] showed that a small trusted module in each distributed component can significantly improve system security. In addition to founding this important new research direction, A2M made two key contributions: First, they proposed a particular abstraction for such a module, namely a *trusted log*. Second, they showed specifically that their proposed abstraction could improve the degree of fault tolerance to Byzantine faults in the server components of client-server systems.

Despite our appreciation for this work, we are concerned that distributed-protocol designers may be reluctant to start assuming the availability of such trusted modules. We have two reasons for this concern: First, the abstraction of a trusted log may require more storage space and complexity than researchers are comfortable assuming, particularly for an embedded module inside a potentially hostile component. Second, designers may have difficulty appreciating how broadly applicable a trusted module can be to distributed protocols.

In this paper, we continue the research direction begun by A2M, with an eye toward addressing these two issues. First, we have developed a significantly smaller abstraction: Instead of a trusted log, we propose a *trusted incrementer* (*TrInc*), which is little more than a monotonic counter and a key. Second, we demonstrate a more inclusive set of architectures, running a broader range of protocols, yielding a wider set of benefits: Our architectures include not only client-server systems but also peer-to-peer systems. Our protocols include not only Byzantine-fault-tolerant protocols but also PeerReview [13] and BitTorrent [8]. Our demonstrated benefits include not only improving fault tolerance but also reducing communication overhead and solving an open incentive problem.

We show that TrInc has several benefits over A2M. First, its smaller size and simpler semantics make it easier to deploy, as we demonstrate by implementing it on real, currently available trusted hardware. Second, we observe that TrInc’s core functional elements are included in the Trusted Platform Module (TPM) [38] found on many modern PCs, lending credence to the idea that such a component could become widespread. Third, TrInc makes use of a shared symmetric session key among all participants in a protocol instance, which significantly decreases the cryptographic overhead.

The rest of this paper is structured as follows. §2 provides background on the underlying problem addressed by TrInc (and by A2M), as well as a primer on trusted hardware. §3 then presents the design of TrInc, and §4 analyzes its security. §§5, 6, and 7 respectively describe several protocols we modified to use TrInc, our trusted hardware implementation, and our evaluation thereof.

	Accountability layer		Trusted module	
Property	PeerReview [13]	Nysiad [14]	A2M [7]	TrInc
No centralized trust	✓*	✓*		
Easy to deploy	✓	✓		✓
Easy to apply to existing protocols	✓	✓†		✓‡
Immediate consistency		✓	✓	✓
No assumptions about protocol's determinism		✓†	✓	✓
No additional online assumptions			✓	✓
Additional communication overhead per protocol message, with witness sets of size W	$O(W^2)$	$O(W^2)$	$O(1)$	$O(1)$

Table 1: Summary of the properties of various equivocation-fighting systems. *While PeerReview and Nysiad do not require centralized trust, they do make use of a PKI. †Nysiad deals with nondeterminism by treating nondeterministic events as inputs; this requires protocol changes for nondeterministic state machines. ‡We found that, although TrInc requires a protocol redesign, the modifications are often localized, and vastly simplify security procedures.

2 Background and Related Work

2.1 Equivocation in distributed systems

Since 1982, it has been known that tolerating f Byzantine faults requires at least $3f + 1$ participants [20]. This stands in marked contrast to the case for f stopping faults, which more intuitively requires $2f + 1$ participants. A key insight behind A2M [7] was the observation that a single property of Byzantine faults is responsible for the difference between these two bounds. That property is *equivocation*, meaning the ability to make conflicting statements to different participants. A2M provides a mechanism that prevents participants from equivocating, thereby improving the fault tolerance of Byzantine protocols to f out of $2f + 1$.

We make the further observation that equivocation is a necessary property for many forms of cheating and fraud, not merely for classical Byzantine faults. For instance, in BitTorrent, recent work [21] has shown an exploit in which a peer can obtain an unfairly high download rate by lying about which chunks of a file it has received. This is equivocation, insofar as the peer acknowledges receiving a chunk from the peer that provided it, but then tells another peer that it does not have the chunk.

The following are three more brief examples:

- In a simultaneous-turn game, one can cheat by observing an opponent's move before making one's own move; this is equivocating about whether one has yet moved.
- In a distributed electronic currency system, one can counterfeit money by equivocating to different payees about whether one has spent a particular bill.
- In an election, the tallier can disrupt the vote by equivocating to a voter and an official about whether the voter's vote was recorded.

In §5.5, we will consider many other cases of malicious behavior that can be interpreted as equivocation.

2.2 Prior solutions to equivocation

Several recent efforts have addressed the problem of Byzantine faults in distributed systems. Although their approaches to the problem are very different, they have all effectively focused on the issue of equivocation. Table 1 summarizes our analysis of their properties.

PeerReview [13] is a system that employs witnesses to collect a tamper-evident record of all messages in a distributed system for subsequent checking against a reference implementation. Unlike the remaining approaches we will discuss, PeerReview does not provide fault tolerance. Instead, it provides eventual fault detection and localization, which the system's designers argue leads to fault deterrence. The tamper-evident record is a distributed collection of logs that are authenticated using hash chains. The purpose of the tamper-evidence is to detect equivocation about the messages recorded in a log. As shown in Table 1, the communication required to collectively manage the tamper-evident message log is quadratic in the size of the witness set.

Nysiad [14] is a mechanism that transforms crash-tolerant distributed systems into Byzantine-fault-tolerant ones. It does this by assigning a set of *guards* (comparable to witnesses) to each host in the system. The guards validate the messages sent by their associated hosts, using replicas of the hosts' execution engines. The potential for equivocation in Nysiad is that the host might send different messages to different guards or order its messages differently for different guards. To deal with this equivocation, the guards gossip among each other to agree on the order and content of messages sent by the host. As shown in Table 1, this gossip requires a count of messages that is quadratic in the number of guards. Relative to PeerReview, Nysiad has the benefit of immediate consistency, rather than eventual detection. Nysiad is also able to handle nondeterministic state machines, but doing so requires protocol changes to treat nondeterministic events as inputs.

Attested Append-only Memory, or A2M [7], is a trusted module that is embedded in an untrusted machine, for the purpose of improving the fault tolerance of a distributed protocol. The A2M module provides the abstraction of a trusted log, which the machine can append to but not otherwise modify. This limitation prevents the machine from equivocating about whether it performed a particular action at a particular step, because once the action is recorded in the log, it cannot be overwritten. A2M uses cryptography to enforce its properties and to attest the log's contents to other machines. Relative to Nysiad and PeerReview, A2M does not require any additional online communication between machines beyond what is required in the base protocol. Consequently, the communication overhead is merely a constant factor due to the cryptographic attestations that accompany the protocol's messages.

As we will show in §3, TrInc is significantly smaller than A2M, making it easier to deploy. TrInc also has another advantage, namely that its use is less tightly coupled to the distributed protocol than use of A2M is. Specifically, because A2M's trusted log has finite storage, it provides a log-truncation operation, but opportunities to truncate the log may be limited by the protocol. Conversely, message sequencing in the protocol may be constrained by the available space in A2M's log. Perhaps in part to address this concern, A2M considered various implementations in addition to hardware, some of which would likely have plentiful storage for the log. These include a remote service, a software-isolated process, and a memory-isolated virtual machine. By contrast, the protocol modifications required to use TrInc tend to be quite localized. Furthermore, TrInc's use of a shared session key often simplifies the protocol.

2.3 Trusted hardware

There have been many trusted hardware designs that predate both TrInc and A2M. Perhaps most similar to TrInc is the abstraction of *virtual monotonic counters* [34]. These are similar to the four increment-only counters included in the current specification of the TPM [38]. Van Dijk et al. propose an algorithm by which to emulate multiple counters with a single trusted counter [39]. We believe a similar approach could ease TrInc's deployment by requiring fewer physical counters. Further, other systems have been proposed that make use of trusted hardware, such as for securing database systems [26] and auctions [31]. To the best of our knowledge, TrInc is the first trusted component designed to be used in large-scale, distributed systems.

3 TrInc Design

3.1 Design Goals

The fundamental security goal of TrInc is to remove participants' ability to equivocate. Consider the situation in which Mallory wishes to send conflicting messages to Alice and Bob. Common approaches to combating such equivocation require Alice and Bob to communicate with one another [13, 14, 20] or with a third party, so they can learn of the distinct messages sent to each. Unfortunately, this additional communication overhead can become a bottleneck for the overlying system, and constitutes the super-linear number of messages in PeerReview [13].

One goal of TrInc is to therefore minimize both communication overhead and the number of non-faulty participants required. With trusted hardware, it is possible to remove Mallory's ability to equivocate without *any* communication between Alice and Bob [7].

The other broad goal of TrInc is to be practical for distributed systems *today*. To be practical, a trusted component must be *small* so that it is feasible to manufacture and deploy. Arbitrary computation and a large amount of storage are difficult and costly to make tamper-resistant. Further, to be a practical primitive in distributed systems, the trusted component must have an API with which it is easy to build distributed systems.

3.2 Overview

To gain the benefits of TrInc, a user must attach a trusted piece of hardware we call a *trinket* to his computer. Unlike a typical TPM, which must attest to states of the associated computer, the trinket's API depends only on its internal state, so the trinket does not need access to the state of the computer. All it needs is an untrusted channel over which it can receive input and produce output, so even USB is quite sufficient.

When Mallory wishes to send a message m to Alice, she must include an attestation from her trinket that (1) binds m to a certain value of a counter, and (2) ensures Alice that no other message will ever be bound to that value of that counter, even messages sent to other users. A trinket enables such attestation by using a counter that monotonically increases with each new attestation. In this way, once Mallory has bound a message m to a certain counter value c, she will never be able to bind a different message m' to that value.

As we show in our case studies in §5, some protocols benefit from using multiple counters. In theory, anything done with multiple counters can be done with a single counter, but multiple counters allow certain performance optimizations and simplifications, such as assigning semantic meaning to a particular counter value. Furthermore, the user of a trinket may participate in multiple protocols, each requiring its own counter or counters. Therefore, a trinket provides the ability to allocate new counters. However, we must identify each of them uniquely so that a malicious user cannot create a new counter with the same identity as an old counter and thereby attest to a different message with the same counter identity and value.

As a performance optimization, TrInc allows its attestations to be signed with shared symmetric keys, which vastly improves its performance over using asymmetric cryptography or even secure hashes. To ensure that participants cannot generate arbitrary attestations, the symmetric key is stored in trusted memory, so that users cannot read it directly. Symmetric keys are shared among trinkets using a mechanism that ensures they will not be exposed to untrusted parties.

3.3 Notation

We use the notation $\langle x \rangle_K$ to mean an attestation of x that could only be produced by an entity knowing K. If K is a symmetric key, then this attestation can be verified only by entities that know K; if K is a private key, then this attestation can be verified by anyone, or more accurately anyone who knows the corresponding public key. We use the notation $\{x\}_K$ to mean the value x encrypted with public key K, so that it can only be decrypted by entities knowing the corresponding private key.

3.4 TrInc state

Figure 1 describes the full internal state of a trinket, which we describe in more detail here. Each trinket is endowed by its manufacturer with a unique identity I and a public/private key pair $(K_{\text{pub}}, K_{\text{priv}})$. Typically, I will be the hash of K_{pub}. The manufacturer also includes in the trinket an attestation $\mathcal{A}$ that proves the values I and K_{pub} belong to a valid trusted trinket, and therefore that the corresponding private key is unknown to untrusted parties.

We leave open the question of what form $\mathcal{A}$ will take. This attestation is meant to be evaluated by users, not by trinkets, and so can be of various forms. For instance, it might be a certificate chain leading to a well-known authority trusted to oversee trinket production and ensure their secrets are well kept.

Another element of the trinket's state is the *meta-counter* M. Whenever the trinket creates a new counter, it increments M and gives the new counter identity M. This allows users to create new counters at will, without sacrificing the non-monotonicity of any particular counter. Because M only goes up, once a counter has been created it can never be recreated by a malicious user attempting to reset it.

Yet another element is Q, a limited-size FIFO queue containing the most recent few counter attestations generated by the trinket. It is useful for allowing users to recover from power failures, as we will describe later.

The final part of a trinket's state is an array of counters, not all of which have to be in use at a time. For each in-use counter, the state includes the counter's identity i, its current value c, and its associated key K. The identity i is, as described before, the value of the meta-counter when the counter was created. The value c is initialized to 0 at creation time and cannot go down. The key K contains a symmetric key to use for attestations of this counter; if $K = 0$, attestations will use the private key K_{priv} instead.

Global state:

Notation	Meaning
K_{priv}	Unique private key of this trinket
K_{pub}	Public key corresponding to K_{priv}
I	ID of this trinket, the hash of K_{pub}
$\mathcal{A}$	Attestation of this trinket's validity
M	Meta-counter: the number of counters this trinket has created so far
Q	Limited-size FIFO queue containing the most recent few counter attestations generated by this trinket

Per-counter state:

Notation	Meaning
i	Identity of this counter, i.e., the value of M when it was created
c	Current value of the counter (starts at 0, monotonically non-decreasing)
K	Key to use for attestations, or 0 if K_{priv} should be used instead

Figure 1: State of a trinket

3.5 TrInc API

Figure 2 shows the full API of a trinket, described in more detail in this subsection.

3.5.1 Generating attestations

The core of TrInc's API is `Attest`. `Attest` takes three parameters: i, c', and h. Here, i is the identity of a counter to use, c' is the requested new value for that counter, and h is a hash of the message m to which the user wishes to bind the counter value. `Attest` works as follows:

Algorithm 1 `Attest`(i, c', h, n)

```
1. Assert that $i$ is the identity of a valid counter.
2. Let $c$ be the value of that counter, and $K$ be the key.
3. Assert no roll-over: $c \leq c'$.
4. If $K \neq 0$, then let $a \leftarrow \langle I, i, c, c', h \rangle_K$; otherwise
   let $a \leftarrow \langle I, i, c, c', h \rangle_{K_{\text{priv}}}$.
5. Insert $a$ into $Q$, kicking out oldest value.
6. Update $c \leftarrow c'$.
7. Return $a$.
```

Note that `Attest` allows calls with $c' = c$. This is crucial to allowing peers to attest to what their current counter value is without incrementing it. To allow for this while still keeping peers from equivocating, TrInc includes both the prior counter value and the new one. One can easily differentiate attestations intended to learn a trinket's current counter value ($c = c'$) from attestations that bind new messages ($c < c'$).

3.5.2 Verifying attestations

Suppose a user Alice with trinket A wants to send a message to user Bob with trinket B. She first invokes

Function	Operation
`Attest`(i, c', h)	Verifies that i is a valid counter with some value c and key K. Verifies that $c \leq c'$. Creates an attestation $a = \langle \text{COUNTER}, I, i, c, c', h \rangle_K$; if $K = 0$, uses K_{priv} instead of K. Adds a to Q. Sets $c = c'$. Returns a.
`GetCertificate`()	Returns a certificate of this trinket's validity: $(I, K_{\text{pub}}, \mathcal{A})$.
`CheckAttestation`(a, i)	Returns a boolean indicating whether a is the output of invoking `Attest` on a trinket using the same symmetric key as the one associated with counter i.
`CreateCounter`()	Increments M. Creates a new counter with $i = M$, $c = 0$, and $K = 0$. Returns i.
`FreeCounter`(i)	If i is the identity of a valid counter, deletes that counter.
`ImportSymmetricKey`$(\mathcal{S}, i)$	Verifies that $\mathcal{S}$ is an encrypted symmetric key decryptable with K_{priv}. Decrypts it and installs the included key as K for counter i.
`GetRecentAttestations`()	Returns Q.

Figure 2: API of a trinket

`Attest` on her trinket using the message's hash, and thereby obtains an attestation a. Next, she sends the message to Bob along with this attestation. However, for Bob to accept this message, he needs to be convinced that the attestation was created by a valid trinket. There are two cases to consider: first, that the attestation used A's private key K^A_{priv}, and second, that the attestation used a shared symmetric key K.

In the first case, the API call `GetCertificate` will be useful. This call returns a certificate $\mathcal{C}$ of the form $(I, K_{\text{pub}}, \mathcal{A})$, where I is the trinket's identity, K_{pub} is its public key, and $\mathcal{A}$ is an attestation that I and K_{pub} belong to a valid trinket. Alice can call this API routine and send the resulting certificate $\mathcal{C}^A$ to Bob. Bob can then (a) learn Alice's public key K^A_{pub}, and (b) verify that this is a valid trinket's public key. After this, he can verify the attestation Alice attached to her message, and any future attestations she attaches to messages.

In the second case, the API call `CheckAttestation` is useful. When `CheckAttestation`$(a,\ i)$ is invoked on a trinket, the trinket checks whether a is the output of invoking `Attest` on a trinket using the same symmetric key as the one associated with the local counter i. It returns a boolean indicating whether this is so. So, if Alice sends Bob an attestation signed with a shared symmetric key, Bob can invoke `CheckAttestation` on his trinket to learn whether the attestation is valid.

3.5.3 Allocating counters

Since a trinket may contain many counters, another important component of TrInc's API is the creation of these counters. TrInc creates new logical counters, and allows counters to be deleted, but never resets an existing counter. Logical counters are identified by a unique ID, generated using a non-deletable, monotonic *meta-counter* M. Every trinket has precisely one meta-counter, and when it expires, the trinket can no longer be used; we compensate for this by making M 64 bits, only incrementing M, and assigning no semantic meaning to M's value. TrInc exports a `CreateCounter` function that increments M; allocates a new counter with identity $i = M$, initial value 0, and initial key $K = 0$; and returns this new identity i. When the user no longer needs the counter, she may call `FreeCounter` to free it and thereby provide space in the trinket for a new counter.

3.5.4 Using symmetric keys

TrInc allows its attestations to be signed with shared symmetric keys, which vastly improves its performance over using asymmetric cryptography or even secure hashes. When a set of users are willing to use a single symmetric key for a certain purpose, we call this a *session*. Creating a session requires a *session administrator*, a user trusted by all participants to create a session key and keep it safe, i.e., to not reveal it to any untrusted parties.

To create a session, the session administrator simply generates a random, fresh symmetric key as the session key K. To allow a certain user to join the session, he asks that user for his trinket's certificate $\mathcal{C}$. If the session administrator is satisfied that the certificate represents a valid trinket, he encrypts the key in a way that ensures it can only be decrypted by that trinket. Specifically, he creates $\{\text{KEY}, K\}_{K_{\text{pub}}}$, where K_{pub} is the public key in $\mathcal{C}$. He then sends this encrypted session key to the user who wants to join the session.

Upon receipt of an encrypted session key, the user can join one of his counters to the session by using the API call `ImportSymmetricKey`$(\mathcal{S}, i)$. This call checks that $\mathcal{S}$ is a valid encrypted symmetric key, meant to be decrypted by the local private key. If so, it decrypts the session key and installs it as K for local counter i. From this point forward, attestations for this counter will use the symmetric key. Also, the user will be able to verify any trinket's attestation a using this symmetric key by invoking `CheckAttestation`(a, i).

3.5.5 Handling power failures

One practical concern is that of power failure. Unlike A2M, TrInc users need not query the trusted hardware to obtain attestations. Instead, TrInc relies on the application (or a TrInc driver) to store attestations in untrusted, persistent storage. If there is a power failure between the time that the trinket advances its counter and the application writes it to disk, then the attestation is lost. This can be problematic for many protocols, which rely on the user being able to attest to a message with a particular counter value. For instance, if Charlie cannot produce an attestation for counter value v, Alice may suspect this is because Charlie has already told Bob about some message m associated with that counter value. Not wanting to be fooled about the absence of such a message, Alice may lose all willingness to trust Charlie.

To alleviate this, a trinket includes a queue Q containing the most recent attestations it has created. To limit the storage requirements, this queue only holds a certain fixed number k of entries, perhaps 10. In the event of a power failure, after recovery the user can invoke the API call `GetRecentAttestations` to retrieve the contents of Q. Thus, all a user must do to protect against power failure is make sure she writes a needed attestation to disk before she makes her kth next attestation request. As long as k is at least 1, the user can safely use the trinket for any application. Higher values of k are useful as a performance optimization, allowing greater pipelining between writing to disk and submitting attestations.

So far we have only discussed a power failure affecting the user, but a power failure can also affect the trinket. The `Attest` algorithm ensures that the attestation is inserted into the queue before the counter is updated, so the trinket cannot enter a situation where the counter has been updated but the attestation is unavailable. It can, however, enter the dangerous situation in which the attestation is in Q, and thus available to the user, but the counter has not been incremented. This window of vulnerability could potentially be exploited by a user to generate multiple attestations for the same counter value, if he could arrange to shut off power at precisely this intervening time. However, we guard against this case by having the trinket check Q whenever it starts up. At startup, before handling any requests, it checks all attestations in Q and removes any that refer to counter values beyond the current one.

3.5.6 A TrInc by any other name

The computational demands of a trinket are small. It must be able to do simple operations such as comparison, as well as cryptographic operations including hashing and both symmetric and asymmetric encryption and decryption. Such cryptographic operations are standard in trusted components such as the TPM [38]. However, we recognize that hardware manufacturers and users are often highly cost-conscious and may be willing to do without performance optimization to save hardware costs.

Therefore, we propose three versions of TrInc that make different trade-offs between cost and performance, summarized in Table 2. The bronze version simply offers correctness with no performance optimizations, by leaving out the ability to use symmetric keys. The silver version is as we have described it. The gold version adds one additional optimization: the use of fast persistent memory such as battery-backed RAM. This optimization makes attestations especially fast since they need not incur the cost of writing to the slow flash memory often found in modern TPMs.

	Persistent Memory	Asym. Crypto	Symm. Crypto	Fast Memory
Bronze TrInc	✓	✓		
Silver TrInc	✓	✓	✓	
Gold TrInc	✓	✓	✓	✓

Table 2: Versions of TrInc with different performance.

3.6 Local adversaries

Mutually distrusting principals on a single computer will share access to a single trinket, creating the potential for conflict between them. Although they cannot equivocate to remote parties, they can hurt each other. They can impersonate each other by using the same counter, and they can deny service to each other by exhausting shared resources within the trinket. Resource exhaustion attacks include allocating all available counters, submitting requests at a high rate, and rapidly filling the queue Q to prevent the pipelining performance optimization.

The operating system can solve this problem by mediating access to the trinket, just as it mediates access to other devices. In this way, the OS can prevent a principal from using counters allocated to other principals, and can use rate limiting and quotas to prevent resource exhaustion. Developing a detailed API and policy for such mediation is beyond the scope of this paper, and is left for future work. However, note that a remote party need not care about how or whether such local mediation is done. Equivocation to remote parties is impossible, even if an adversary has root access to the machine, since cryptography allows the trinket to communicate securely even over an untrusted channel.

4 Analysis of TrInc

We now present a brief discussion of why TrInc is sufficient for a broad class of distributed protocols and why it is nearly minimal in size.

4.1 Equivocation

When a trinket creates an attestation with distinct old and new counter values of c and c', we say that attestation *covers* the half-open interval $(c, c']$. TrInc prevents equivocation by ensuring that no two attestations will cover overlapping intervals. This property could be violated only if:

- the counter is decremented,
- the cryptosystem is broken,

- more than one counter has the same identity, or
- more than one trinket has the same identifier.

By construction, it is not possible to decrement the counter nor to assign the same identity to multiple counters. By hypothesis, cryptographic primitives are effectively unbreakable. Finally, no two trinkets will be created with the same identifier, at least not by a trusted manufacturer; recall that users can verify whether the trinket comes from a trusted manufacturer by observing the certificate chain in $\mathcal{A}$.

4.2 Timeliness

When a trinket creates an attestation with the same old and new counter values, there is no change to the trinket's state; however, the attestation demonstrates the current value of the counter. Thus, if a machine attests to a value of a remotely supplied nonce, the remote machine can be certain that the attestation was generated after the nonce was supplied. Since this attestation carries the current counter value, the remote machine can thus also be sure that the local machine's counter is no lower than this value.

Therefore, when the local machine provides attestations of counter values up to the nonce-attested value, the remote machine can be certain that these attestations are timely.

4.3 Minimality

Suppose, during the execution of a protocol, a participant sends n messages requiring attestation, but her attesting module has fewer than $\log_2(n)$ bits of storage. The attesting module must be willing to provide all n attestations, or else it will cause the protocol to halt prematurely. However, since the module can be in fewer than n distinct states, by the pigeonhole principle it must be willing to attest to two different messages while in the same state. Since this state is as it was before the first message, it cannot reflect the trinket's having attested to the first message. This means a malicious user could take advantage of the trinket's inability to remember its first attestation when requesting the second attestation, and thereby obtain an attestation inconsistent with the earlier one. This is clearly inconsistent with the goals of a trusted module, so we come to a contradiction, and conclude that such a module requires at least $\log_2(n)$ bits of storage. In other words, it needs sufficient storage to accommodate a message counter.

Furthermore, an attesting module needs for its attestations to be unforgeable. Otherwise, the user could generate attestations without using the module, and thereby attest to both sides of an equivocation. TrInc achieves this unforgeability with simple cryptographic primitives.

In summary, the core components of TrInc, a counter and cryptography, seem to be essential for equivocation prevention.

5 Designing Systems with TrInc

5.1 Overview

When designing a protocol that incorporates TrInc, we find it important to address the following questions:

5.1.1 What does TrInc's counter represent?

In the applications we have considered, TrInc's counter represents a natural "progression" of the system. In BitTorrent, for instance, the counter represents the number of blocks a given peer has received, a value which is naturally monotonically increasing. In Byzantine Fault Tolerance (BFT), the counter represents which view a replica is in. Ultimately, the choice of what the counter represents is dependent on what data peers will need to attest to.

5.1.2 To what data do peers attest?

There are two broad types of attestations that TrInc offers. *Advance* attestations increase the trinket's counter, thus binding a message to a counter. *Status* attestations attest to the current counter without advancing it.

Advance attestations Advance attestations are largely protocol-dependent, including such elements as the set of pieces received in BitTorrent, or the root of a Merkle tree of file hashes in a file server. The specific data to which to attest often requires a careful analysis of the selfish or malicious ways in which peers could equivocate. It is important to ensure that the impossibility of equivocating about what was assigned to a particular counter value translates into the impossibility of equivocating at the higher desired semantic level.

For instance, suppose an attestation consists solely of a number n of pieces received in BitTorrent and a list of n peers. In this case, a participant Mallory can cheat in the following way. After receiving the first piece a from Alice, she replies with an attestation that her one-piece set contains only a. Next, after receiving her next two pieces b from Bob and c from Charlie, she sends them both an identical attestation that her two-piece set is b and c. In this way, Mallory gets away with hiding the fact that she has received piece a, despite not being able to get different attestations for the same value of $n = 2$. As we will see later, in §5.4, we prevent this by having an attestation include the last piece received.

Status attestations Most distributed systems do not have an implicit system-wide "counter." Rather, peers progress at varying rates: BitTorrent peers download at rates largely dependent on their own upload rates, DHT peers store varying amounts of data, and so on. Status attestations enable peers to determine others' current counter values. The data in a status attestation is generally a nonce, to ensure freshness in peers' reports of their counters. Coupled with a counter that has semantic meaning, status attestations can provide peers with up-to-date information about their neighbors. In BitTorrent, for instance, knowing how much of a file a neighbor has downloaded can help determine whether to bootstrap him

Algorithm 2 Implementation of A2M with TrInc

```
Init()
  1. Create low and high counters:
       $\mathcal{L}_q \leftarrow$ CreateCounter(); $\mathcal{H}_q \leftarrow$ CreateCounter()
  2. Return $\{\mathcal{L}_q, \mathcal{H}_q\}$

Append(queue $q$, value $x$)
  1. Bind $h(x)$ to a unique counter (the current "high counter"):
       $a \leftarrow$ Attest($\mathcal{H}_q$.id, $\mathcal{H}_q$.ctr $+ 1$, $h(x)$)
  2. Store the attestation in untrusted memory:
       $q$.append($a, x$)

Lookup(queue $q$, sequence number $n$, nonce $z$)
  1. If $n < \mathcal{L}_q$, the entry was truncated. Attest to this by returning an
     attestation of the supplied nonce using the low-counter:
       Attest($\mathcal{L}_q$.id, $\mathcal{L}_q$.ctr, $h(\text{FORGOTTEN}||z)$)
  2. If $n > \mathcal{H}_q$, the query is too early. Attest to this by returning an
     attestation of the supplied nonce using the high-counter:
       Attest($\mathcal{H}_q$.id, $\mathcal{H}_q$.ctr, $h(\text{TOOEARLY}||z)$)
  3. Otherwise, return the entry in $q$ that spans $n$, i.e., the one such that
     $a.c < n \le a.c'$. Note that if $n < a.c'$, this means $n$ was skipped
     by an Advance.

End(queue $q$, sequence number $n$, nonce $z$)
  1. Retrieve the latest entry from the given log:
       $\{a, x\} \leftarrow q$.end()
  2. Attest that this is the latest entry with a high-
     counter attestation of the supplied nonce:
       $a' \leftarrow$ Attest($\mathcal{H}_q$.id, $\mathcal{H}_q$.ctr, $z$)
  3. Return $\{a', \{a, x\}\}$

Truncate(queue $q$, sequence number $n$)
  1. Remove the entries from untrusted memory:
       $q$.truncate($n$)
  2. Move up the low counter:
       $a \leftarrow$ Attest($\mathcal{L}_q$.id, $n$, FORGOTTEN)

Advance(queue $q$, sequence number $n$, value $x$)
  1. Append a new item with sequence number $n$:
       $a \leftarrow$ Attest($\mathcal{H}_q$.id, $n$, $h(x)$)
  2. Store the attestation in untrusted memory:
       $q$.append($a, x$)
```

with free pieces (because he is new to the swarm) or to initiate a trade with him (because he has many interesting pieces of the file).

5.2 Case study 1: A2M

Attested Append-only Memory (A2M) [7] is another proposed trusted hardware design with the intent of combating equivocation. A2M offers *trusted logs*, to which users can only append. The fundamental difference between the designs of A2M and TrInc are in the amount of state and computation required from the trusted hardware. To demonstrate that TrInc's decreased complexity is enough, we present, as our first case study, how to build A2M using TrInc.

5.2.1 A2M overview

A2M's state consists of a set of logs, each containing entries with monotonically increasing sequence numbers. A2M supports operations to add (append and advance), retrieve (lookup and end), and delete (truncate) items from its logs. The basis of A2M's resilience to equivocation is append, which binds a message to a unique sequence number. For each log q, A2M stores the lowest sequence number, $\mathcal{L}_q$, and the highest sequence number, $\mathcal{H}_q$, stored in q. A2M appends an entry to log q by incrementing the sequence number $\mathcal{H}_q$ and setting the new entry's sequence number to be this incremented value. The low and high sequence numbers allow A2M to attest to failed lookups; for instance, if a user requests an item with sequence number $s > \mathcal{H}_q$, A2M returns an attestation of $\mathcal{H}_q$.

5.2.2 Trusted logs with TrInc

In our TrInc-based design of A2M, we store logs in untrusted memory, as opposed to within a trinket. As in A2M, we make use of two counters per log, representing the highest ($\mathcal{H}_q$) and lowest ($\mathcal{L}_q$) sequence number in the respective log q.

We present the detailed protocol in Algorithm 2, and summarize some of its characteristics here. Note the power of TrInc's simple API; our design is built predominately on calls to a trinket's Attest function. Our protocol uses advance attestations for moving the high sequence number when appending to the log, and for moving the low sequence number when deleting from the log. We perform status attestations of the low counter value to attest to failed lookups, and of the high counter to attest to the end of the log. No additional attestations are necessary for a successful lookup, even if the lookup is to a skipped entry. Conversely, A2M requires calls to the trusted hardware even for successful lookups.

5.2.3 Properties of a TrInc-based A2M

Chun et al. [7] demonstrate how to apply A2M to BFT [20], SUNDR [22], and Q/U [1]. Our implementation of A2M in TrInc demonstrates that TrInc, too, can be applied to these systems.

Implementing trusted logs using TrInc has several benefits over a completely in-hardware design like A2M. Because TrInc stores the logs in untrusted storage, we decouple the usage demand of the trusted log from the amount of available trusted storage. Conversely, limited by the amount of trusted storage, A2M must make more frequent calls to `truncate` to keep the logs small. Some systems, such as PeerReview [13], benefit from large logs, making TrInc a more suitable addition, which we consider next.

5.3 Case study 2: PeerReview

Accountability systems, such as PeerReview [13] and Nysiad [14], strive to augment existing protocols to make them tolerant to Byzantine faults. This is a powerful approach, as it allows system designers to focus on the system at hand, rather than consider Byzantine faults at all layers of the system. The general approach is to have participants in the system communicate with and audit one another, resulting in what is sometimes, unfortunately, a massive amount of additional communication overhead.

Our main observation in this case study is that the means by which these systems combat equivocation constitutes the bulk of their communication overhead. By applying TrInc to PeerReview, we are able to vastly reduce PeerReview's communication overhead.

5.3.1 PeerReview review

PeerReview [13] is a system that enables accountability in general distributed protocols. Unlike BFT, which ensures that bad behavior never has an effect, PeerReview allows bad behavior to affect the system but ensures that the improper act will eventually be detected. This allows a system to correct for bad behavior after the fact, and also deters bad behavior to begin with.

PeerReview works on any protocol in which each participant acts according to a deterministic state machine. PeerReview assigns each participant a set of *witnesses*, machines whose job it is to detect bad behavior by that participant. The participant is required to log all of the messages it sends and receives, and report these to the witnesses. The witnesses then run the participant's state machine to ensure the participant's outgoing messages were consistent with proper operation.

A participant might try to cheat by sending different messages to the witnesses than it sends to other participants. For this reason, when a participant receives a message from another, it forwards this message to the sender's witnesses, so they can ensure this message actually appears in the sender's log.

As a practical matter, full messages do not have to be transmitted to witnesses thanks to the use of a *tamper-evident log*. Each log entry is associated with a sequence number, and the log itself is represented by a recursive hash reflecting all log entries. When a participant sends a message, it includes a signed statement that this message has a particular sequence number and that the log had a particular recursive hash when this message was logged. In this way, the receiver only needs to report this authenticator to the witness.

PeerReview's tamper-evident log has another important use. When a participant or witness discovers bad behavior in a participant, the authenticators signed by the malefactor stand as clear proof of the misbehavior. Thus, a faulty witness cannot improperly accuse a participant, and an incompletely trusted witness can be believed when it presents evidence of a participant's misbehavior.

5.3.2 Simplifying PeerReview with TrInc

By augmenting PeerReview with TrInc, we are able to simplify much of PeerReview's protocol. We detail here the modifications we make to PeerReview in augmenting it with TrInc.

Trusted logs As demonstrated with A2M, TrInc can easily supply a trusted log without the assistance of a witness set. Our first modification is to include such a trusted log. Whenever a participant sends or receives a message, it logs that message with an attestation from its trinket. A participant should only process a received message if it is accompanied by an attestation that the message has been logged by the sender's trinket.

Audits Each witness w for a participant p keeps track of n, a log sequence number, and s, the state that p should have been in after sending or receiving the message in log entry n. It initializes n to 0 and s to the initial state of participant p.

Whenever w wants to audit p, it sends it n and a nonce. The participant returns an attestation of its current log entry number n' using the nonce, and also returns a log entry and attestation for every index i such that $n < i \leq n'$. Note that witnesses need only obtain these entries directly from p, and not from other peers with whom p has communicated. The witness then runs the reference implementation, starting at state s, and progressing through the log entries between n and n'. If the reference implementation sends the same messages that are in the log, then the witness simply updates n to n' and updates s to the state of the reference implementation at that point. If not, then the witness has proof it can present of the participant's failure to act properly.

5.3.3 Properties of a TrInc-enabled PeerReview

The benefits from applying TrInc to PeerReview are evident when considering what the protocol *no longer has to do*.

Challenge/response Enabled with TrInc, PeerReview's challenge/response protocol is no longer needed for a participant to verify a hash chain of log entries. The fact that TrInc signs the messages is sufficient. The only time a participant i has to challenge another participant j is when it sends participant j a message and receives no acknowledgment of it. In this case, the challenge works as in regular PeerReview.

Consistency TrInc further removes the need for witness-to-witness communication. In PeerReview, if p receives an authenticator from q, then p's witnesses must forward it to q's witnesses. This is not necessary in a TrInc-augmented PeerReview because there would be no way for those other participants to avoid sending the authenticators themselves to their witnesses. Another way to look at it is that it is not necessary for a participant to pass on authenticators it receives to witnesses, so it is not necessary for a witness to do this on behalf of participants.

To summarize, we find that by applying TrInc to PeerReview, we are able to vastly decrease the amount of communication overhead. We demonstrate this empirically in Section 7.

5.4 Case study 3: BitTorrent

The previous two systems demonstrate that TrInc is a minimal counterpart to a related trusted component, and that it can reduce the overhead of achieving accountability in a distributed setting. Our third case study demonstrates TrInc's versatility. We show how TrInc can be applied to solving an open incentive problem [21] in the immensely popular BitTorrent system [8].

5.4.1 A brief overview of BitTorrent

BitTorrent [8] is a decentralized file swarming system whose goal is to disseminate large files to a large number of downloaders. Rather than rely on a highly provisioned server, BitTorrent peers trade small *pieces* of a file with one another, thereby contributing to the system while gaining from it. *Bitfields* represent which pieces of a file a peer has. Peers trade bitfields in order to gain one another's interest; a peer is *interested* in peers who have pieces that it does not. Since peers only upload to peers in whom they are interested, peers have incentive to be as interesting to as many others as possible.

5.4.2 Piece under-reporting

BitTorrent peers can sometimes have incentive to under-report what pieces they have to their neighbors, since by doing so they can limit the degree to which their neighbors find interest in one another [21]. For instance, suppose peer i has neighbors j and k, both of whom want pieces p and q from i. If i were to tell them both about both pieces, one might demand p and the other might demand q. After obtaining them, they might gain interest in one another and exchange p and q among themselves, thus *decoupling* from i. Thus, i may prefer to under-report by sending to j and k a bitfield that contains p but not q. As a result, both neighbors request and obtain p, gaining no interest in one another; only then does i reveal that he also has piece q, forcing j and k to download it from i.

Such under-reporting leads to a tragedy of the commons, since although strategic under-reporters' download times improve, the system as a whole suffers [21]. Since its recent discovery, strategic under-reporting has yet to be solved; we demonstrate how to solve it with TrInc.

5.4.3 Solving under-reporting with TrInc

We observe that *under-reporting in file swarming systems is an act of equivocation.* Using the above example, when peer i received piece q from peer ℓ, i must have sent an acknowledgment, stating to ℓ that he received the piece. However, by under-reporting q to peers j and k, i is effectively contradicting a statement he made earlier to ℓ.

Algorithm 3 Fighting equivocation in BitTorrent

```
Upon receipt of piece $p$:
  1. Add $p$ to bitfield $B$
  2. $a_{curr} \leftarrow \texttt{Attest}(i, |B|, h(p, B))$

Upon sending piece $p$ to neighbor $j$:
  1. Request an attestation from $j$ with a random nonce.
  2. Do not send any piece other than $p$ to $j$ until $j$ admits to having $p$.

Periodically, for each neighbor $j$:
  1. Request an attestation of $j$'s current bitfield with a random nonce.

Upon receiving an attestation request with nonce $z$:
  1. $a_{tmp} \leftarrow \texttt{Attest}(i, |B|, z)$.
  2. Reply with $(a_{curr}, a_{tmp})$.
```

Our goal is therefore to remove BitTorrent peers' ability to undetectably equivocate. We present in Algorithm 3 a TrInc-based protocol for fighting equivocation in BitTorrent. In this protocol, a peer attests to his bitfield, incrementing a trinket counter for each piece he receives. Also, peers periodically request up-to-date attestations from their neighbors, to maintain fresh state.

Because they join the swarm at different times and download at different rates, peers' counters are not synchronized. In Algorithm 3, the TrInc counter does not correspond to some system-wide "round" the protocol is in, as it would in, say, BFT machine replication. Instead, peer i's counter represents how many pieces i has downloaded. This is a natural fit for the counter, because it is a monotonically increasing number, and because the type of malicious behavior we want to prevent corresponds to pretending it is not monotonic.

Algorithm 3 demonstrates the importance of choosing the correct data to which to attest. Suppose, for instance, peers were to attest *only* to their bitfields. Clearly, when s sends an attested bitfield to neighbor n, s must include the piece n sent him, p_n, in the bitfield, otherwise n will observe an under-report. Were s to attest only to the bitfield, then s could under-report as follows, where B_{old} represents the bitfield before receiving pieces p_a, p_b, and p_c, and $\oplus$ denotes adding a piece to the bitfield:

- To a: $B_{old} \oplus p_a$
- To b and c: $B_{old} \oplus p_b \oplus p_c$

The problem arises because the data to which s is attesting does not enforce monotonicity at the semantic level we desire. Specifically, though the counter cannot decrease, it does not have to correspond to the number of distinct pieces acknowledged, allowing a malicious participant to misstate the number of distinct pieces he has acknowledged.

In our solution, a peer attests not only to the hash of his bitfield B, but also to the most recent piece he has received, p. Neighbor n therefore expects an advance attestation including both p_n and a bitfield containing p_n. As a result, every piece must have a unique advance attestation, ensuring that s's counter must be as large as the number of pieces he has acknowledged receiving.

5.4.4 Properties of a TrInc-augmented BitTorrent

Our TrInc-based solution to equivocation in BitTorrent solves two difficult incentives-related problems. First, peers have incentive to truthfully reveal the pieces they have whenever they are asked to. TrInc removes the ability to equivocate, and step-omission failures (remaining silent) result in getting no further pieces from a neighbor. Peers can therefore obtain long-lived trades with others only by truthfully reporting their pieces.

Second, our solution adds additional security to BitTorrent's bootstrapping mechanism. In BitTorrent, peers *optimistically unchoke* new participants, sending them pieces without requiring anything in return, to introduce them into the system. BitThief [24] exploits this by pretending not to be able to make progress [35]. However, such artifice is not possible with TrInc since with it a peer cannot hide the rate at which he is downloading pieces.

Note, however, that what we propose is not a complete solution to problems with bootstrapping. Even with TrInc-enabled BitTorrent, a peer can steal a single piece from each other peer. Our goal of applying TrInc here is to ensure truthfulness in long-lived peerings, which (surprisingly) does not arise automatically.

5.5 Other applications

We see many other potential applications for TrInc. We briefly described three such apps in Section 2.1: simultaneous-turn games, electronic currency, and elections. Here, we detail several others:

Secure DNS is intended to protect the integrity of the Internet domain name system. One identified threat [6] is that a resolving name server could be compromised and forge incorrect responses. The official solution to this threat is *data origin identification* in the DNS Security Extensions (DNSSEC), which uses public-key signatures to authenticate name updates. However, this solution does not address a threat in which the compromised name server replies to a query with out-of-date data, which would still bear a valid signature. Modifying DNSSEC with TrInc could address this problem by preventing the resolving name server from equivocating about whether it has received an update. Once it acknowledges receipt to the authoritative name server, it can no longer pretend it has not received the update.

Secure Origin BGP (soBGP) [44] is intended to protect the integrity of Internet routing updates. Like DNSSEC, soBGP uses public-key signatures to authenticate updates. Also like DNSSEC, soBGP is vulnerable to a threat in which a compromised router advertises out-of-date routes, which would still bear valid signatures. TrInc could address this problem by preventing a router from equivocating about whether it has received a routing update.

Distributed hash tables (DHTs), such as Chord [37], Bamboo [33], and Kademlia [27], are vulnerable to misbehaving nodes. In particular, a node can lie about which region of the keyspace it is responsible for. As nodes join and leave the DHT, these regions of responsibility change (sometimes quite rapidly [33]) in response to reconfiguration messages. A node can equivocate about whether it has received a particular message, which may allow it to claim responsibility for a region of the keyspace it does not own. TrInc could be used to prevent this equivocation.

Version control systems, such as CVS [41] and Subversion [29] are often run on remote servers. Thus, they are vulnerable to a threat model in which the server presents different views of the repository to different clients. Although this threat could be addressed at the block-store level [22], it might be more efficient to address it at the application level, in which case TrInc could prevent this equivocation.

Distributed auctions [42] are vulnerable to cheating participants. A bidder can try to manipulate others' bids by equivocating about the value of his current bid. An auctioneer can try to manipulate the bidding by equivocating about her reserve price for a particular auction. TrInc could protect against both of these classes of cheating, by preventing both bidders and auctioneers from equivocating.

Leader election protocols [25] rely on a quorum of participants to agree on a choice of leader. For a quorum of size q, it can legitimately happen that two groups of size $q - 1$ will nominate different leaders. In this case, one participant can equivocate about which leader to nominate, causing the protocol to select two leaders concurrently. TrInc could be used to prevent this equivocation.

Digital signatures are used in many cryptographic protocols, but commonly use slow asymmetric key operations [17]. However, TrInc allows faster symmetric key operations to be used instead. To do so, a signer merely has to have his trinket attest to the hash of the message to be signed using a shared symmetric key. Since this attestation can only be generated by a party with access to the symmetric key, and since the hardware includes the ID in any attestation, no other party (except the trusted session administrator) can have generated the attestation. Thus, it functions effectively as a digital signature, verifiable by anyone whose trinket has the same symmetric key installed.

Operation		Time (msec)
Noop		6.14 ± 0.15
Attest	(asymmetric, advance > 0)	230.24 ± 0.28
	(asymmetric, advance = 0)	198.21 ± 0.10
	(symmetric, advance > 0)	128.95 ± 0.08
	(symmetric, advance = 0)	105.90 ± 0.08
Verify Symmetric Attestation		85.81 ± 0.11

Table 3: TrInc microbenchmarks on a Gemalto .NET Smartcard, with 95% confidence intervals.

6 TrInc Implementation

The application case studies demonstrate the strong theoretical properties of TrIncs. In this section, we study the performance of TrIncs *today*. To this end, we have implemented TrInc on Gemalto .NET SmartCards [11], and present microbenchmarks that measure TrInc's performance on these widely available pieces of trusted hardware.

6.1 Microbenchmarks

Our experimental setup consists of an Intel Core 2 Duo 1.6GHz machine with 3GB of RAM, and a smartcard connected via a USB card reader. We present our microbenchmarks in Table 3, with results averaged over 1,000 runs. In addition to TrInc's API, we include a noop to essentially measure the round-trip time between PC and smartcard.

Compare the `Attest` results on the card to those on the untrusted PC, where 3-DES took 0.017 ± 0.008 msec, and RSA took 8.6 ± 0.67 msec. It is no surprise that a smartcard does not perform as well, but the difference in relative performance between symmetric and asymmetric encryption is striking. On the PC, they differ by a factor of over 500, while on the card they differ by less than a factor of 2. While using symmetric instead of asymmetric operations improves TrInc's performance, we were surprised to see it was by this small a factor.

6.2 Why so slow?

The conclusion is clear: today's trusted hardware is *slow*! Indeed, it is much slower than would be allowed by most components of a distributed system. But why is it slow, and why do current applications that use trusted hardware not suffer as a result?

We believe this is attributable to the fact that *TrInc uses trusted hardware in a fundamentally different way than that for which the hardware is currently designed.* Today's trusted hardware is designed to bootstrap software, generally performing few operations during a machine's boot cycle. Conversely, TrInc makes use of trusted hardware *during* operation, in some cases multiple times for each message sent.

We proposed several versions in §3.5.6 that we believe would be viable directions for future designs of trusted hardware to take. In the interim, a logical solution is to design protocols that limit the number of necessary attestations, but such approaches are beyond the scope of this paper. Nevertheless, our empirical results in the following section indicate that making trusted hardware more suitable for use in distributed systems *today* is a valuable area of future work.

	Time (msec)	
Operation	TrInc	A2M
Noop	6.99 ± 0.01	
Append	187.60 ± 0.15	551.93 ± 154
Lookup (Successful)	0.0122 ± 0.02	304.14 ± 6.87
Lookup (TooEarly)	162.24 ± 0.08	289.68 ± 2.23
Lookup (Forgotten)	162.35 ± 0.10	350.51 ± 1.43
End	162.31 ± 0.11	294.16 ± 2.04
Truncate	187.94 ± 0.10	28.99 ± 0.02
Advance	187.81 ± 0.12	288.20 ± 11.4

Table 4: TrInc-A2M microbenchmarks, with 95% confidence intervals.

7 Application Evaluation

We now turn to macrobenchmarks, evaluating TrInc as it applies to our three case studies: A2M, PeerReview, and BitTorrent.

7.1 TrInc-A2M

In Section 5.2, we proposed a way to build A2M using TrInc. While demonstrating TrInc's ease of use and versatility, it also allows us to compare the two trusted-component designs. To this end, we have implemented A2M in the Gemalto .NET SmartCard, and a TrInc library—run on an untrusted machine—that accesses TrInc as prescribed in Algorithm 2.

We present microbenchmark comparisons in Table 4. As expected, TrInc performs `Appends` much more quickly, as it does not require as many writes to trusted storage. Where TrInc offers vast speed improvements over A2M is in successful `Lookups`; since these do not have to be either stored in trusted hardware or attested, they are merely local operations. Interestingly, A2M improves with `Truncate`, since A2M simply increases the log's low counter and postpones the attestation of the operation until a lookup that needs to return FORGOTTEN. TrInc amortizes this cost, in the expectation that there will be more FORGOTTEN lookups than truncations.

These results demonstrate that TrInc performs better on *today's* trusted hardware. As trusted components improve, particularly in terms of memory writes and cryptographic operations, it is likely that A2M and TrInc will perform comparably well. However, the slowness of today's trusted hardware brings to light the difference in complexity between A2M and TrInc. We believe TrInc's relative simplicity makes it a more suitable candidate even with future designs of trusted hardware.

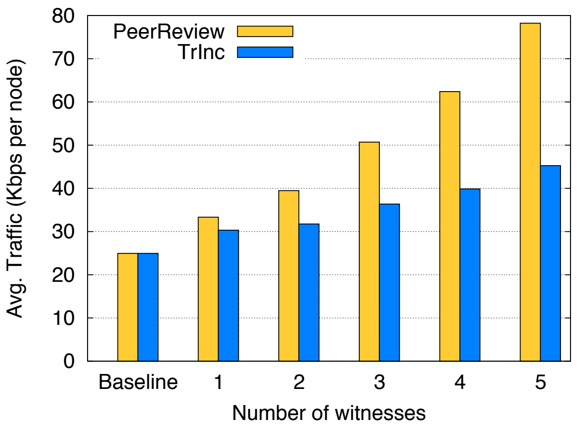


Figure 3: Reduction in PeerReview's message overhead due to TrInc.

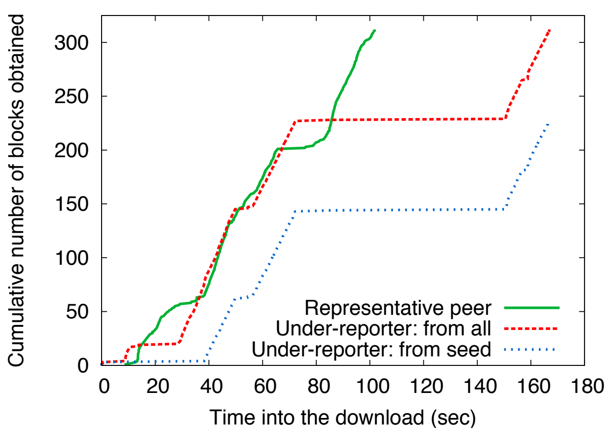


Figure 4: Rate of progress for various BitTorrent clients when TrInc is used.

7.2 TrInc-PeerReview

In Section 5.3, we demonstrated how including TrInc into the design of an accountability system such as PeerReview can decrease the amount of communication required between participants. This represents one of the fundamental strengths of including a small, trusted component into an otherwise untrusted system.

Applying TrInc to PeerReview removes the requirement for a peer p to communicate with the witness set of any other peer q, unless, of course, p happens in q's witness set. Using data from the original PeerReview study [13], we demonstrate in Figure 3 the extent to which TrInc reduces PeerReview's communication overhead. TrInc effectively removes the $O(W^2)$ witness-set-to-witness-set communication, for reasons described in Section 5.3. As a result, the amount of additional communication overhead scales linearly rather than quadratically with the size of the witness sets.

7.3 TrInc-BitTorrent

To evaluate our TrInc-based solution for BitTorrent, we simulated using a "gold-standard" trinket in the Azureus BitTorrent client. To do so, we modified BitTorrent's Have messages to include attestations to counters. We observed that Have messages, originally intended simply to inform others when a peer receives a piece, come frequently enough in practice to also satisfy peers' continual need for fresh attestations.

We modified the BitTorrent code to recognize these new messages, and to cut off peers thereby discovered to be under-reporting. However, we never have the seeder punish a peer in this way. It seems reasonable to have such a forgiving seeder since otherwise peers who suffer failures—for example, from a corrupted disk—could never request blocks after they have attested to them.

We ran our experiments on a local cluster consisting of 23 leechers, each with upload bandwidth capped at 50Kbps, and one seeder, with upload bandwidth capped at 80Kbps. We chose one host to act as a strategic piece revealer using an algorithm from a prior study [21]. We chose this host arbitrarily since, on the local cluster, we found them to be virtually indistinguishable in terms of performance.

Our experiments demonstrated a clear loss in performance from under-reporting. In a representative run, the under-reporting peer took 27% longer to download the file than the other peers did on average, and 33% longer than the median.

The under-reporter's download times would have been much worse if not for the forgiving seeder. We show in Figure 4 the total number of blocks the under-reporter received over time, compared to the number of blocks he received from the seeder. We plot a representative, truthful peer from the swarm as a point of comparison. Because other peers refused to send to the under-reporter until he revealed all the pieces in his possession, the seeder became the under-reporter's only remaining option. Indeed, the under-reporting peer obtained more pieces (73%) from the seeder than any other peer in the swarm (11% on average, 6% median).

These results indicate the power of applying a small amount of trust, and small attestations piggybacked on existing protocol messages, to a large-scale decentralized system.

8 Conclusions

In this paper, we presented TrInc, a simple yet powerful abstraction for improving security in distributed systems. TrInc is a trusted hardware module that holds a non-decreasing counter and a hidden cryptographic key. This combination, along with the computational machinery to support it, yields an abstraction that significantly improves various aspects of security in distributed systems.

TrInc was inspired by the seminal work of A2M, which introduced the idea of a trusted log for improving system security. Relative to A2M, TrInc has a significantly simpler abstraction: a counter instead of a log. We have also demonstrated a wider range of applications for, and benefits from, a trusted module than previously shown.

We have implemented TrInc on real, currently available trusted hardware. We have performed three detailed case studies of TrInc as applied to different distributed protocols. Our results show that this abstraction is easy to deploy, powerful, and versatile.

Acknowledgments

We would like to thank Josh Benaloh, Paul England, Sandro Forin, Atul Singh, Talha Bin Tariq, and Gideon Yuval for helpful discussions and assistance in evaluating TrInc on real trusted hardware. We also thank Adam Bender, Stefan Saroiu, the anonymous reviewers, and our shepherd, Greg Minshall, for their helpful comments on improving the presentation of this paper. Dave Levin was supported in part by a Microsoft Live Labs fellowship.

References

[1] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pp. 59–74, 2005.

[2] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–14, 2002.

[3] N. E. Baughman and B. N. Levine. Cheat-proof playout for centralized and distributed online games. In *Proc. Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, pp. 104–113, 2001.

[4] A. Blanc, Y.-K. Liu, and A. Vahdat. Designing incentives for peer-to-peer routing. In *Proc. NetEcon*, 2004.

[5] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[6] R. Chandramouli and S. Rose. Secure domain name system (DNS) deplopment guide. *Special Publication 800-81, NIST*, 2006.

[7] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: Making adversaries stick to their word. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 189–204, 2007.

[8] B. Cohen. Incentives build robustness in BitTorrent. In *P2PEcon*, 2003.

[9] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 177–190, 2006.

[10] C. GauthierDickey, D. Zappala, V. Lo, and J. Marr. Low latency and cheat-proof event ordering for peer-to-peer games. In *NOSSDAV*, 2004.

[11] Gemalto. http://www.gemalto.com/.

[12] R. Gupta and A. K. Somani. CompuP2P: An architecture for sharing of compute power in peer-to-peer networks with selfish nodes. In *Proc. NetEcon*, 2004.

[13] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 175–188, 2007.

[14] C. Ho, R. van Renesse, M. Bickford, and D. Dolev. Nysiad: Practical protocol transformation to tolerate Byzantine failures. In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 175–188, 2008.

[15] D. Hughes, G. Coulson, and J. Walkerdine. Free riding on Gnutella revisited: The bell tolls? *IEEE Distributed Systems Online*, 6(6):1–18, 2005.

[16] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pp. 29–42, 2003.

[17] J. Katz and Y. Lindell. *Introduction to Modern Cryptography*. CRC Press, 2007.

[18] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine fault tolerance. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 45–58, 2007.

[19] J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proc. Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[20] L. Lamport, R. E. Shostak, and M. C. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, 1982.

[21] D. Levin, K. LaCurts, N. Spring, and B. Bhattacharjee. BitTorrent is an auction: Analyzing and improving BitTorrent's incentives. In *Proc. SIGCOMM Conference on Data Communication*, pp. 243–254, 2008.

[22] J. Li, M. N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 121–136, 2004.

[23] Q. Lian, Y. Peng, M. Yang, Z. Zhang, Y. Dai, and X. Li. Robust incentives via multi-level tit-for-tat. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[24] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free riding in BitTorrent is cheap. In *Proc. Workshop on Hot Topics in Networks (HotNets)*, pp. 85–90, 2006.

[25] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[26] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 135–150, 2000.

[27] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, pp. 109–114, 2002.

[28] A. Muthitacharoen, R. Morris, T. Gil, and B. Chen. Ivy: A read/write peer-to-peer file system. In *Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 31–44, 2002.

[29] W. Nagel. *Subversion Version Control: Using the Subversion Version Control System in Development Projects*. Prentice Hall, 2005.

[30] T.-W. Ngan, D. S. Wallach, and P. Druschel. Incentives-compatible peer-to-peer multicast. In *Proc. NetEcon*, 2004.

[31] A. Perrig, S. Smith, D. Song, and J. D. Tygar. SAM: A flexible and secure auction architecture using trusted hardware. In *ICEC*, 2001.

[32] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *Proc. Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 1–14, 2007.

[33] S. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. USENIX Annual Technical Conference*, pp. 127–140, 2004.

[34] L. F. G. Sarmenta, M. van Dijk, C. W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a TPM without a trusted OS. In *Proc. Workshop on Scalable Trusted Computing (STC)*, 2006.

[35] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding in BitTorrent networks with the large view exploit. In *Proc. Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[36] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *Proc. USENIX Annual Technical Conference*, pp. 157–170, 2007.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. SIGCOMM Conference on Data Communication*, pp. 149–160, 2001.

[38] Trusted Computing Group. Trusted Platform Module Specifications. Online at https://www.trustedcomputinggroup.org/specs/TPM/.

[39] M. van Dijk, L. F. G. Sarmenta, C. W. O'Donnell, and S. Devadas. Proof of freshness: How to efficiently use an online single secure clock to secure shared untrusted memory. Technical report, 2006.

[40] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine faults in transaction processing systems using commit barrier scheduling. In *Proc. Symposium on Operating Systems Principles (SOSP)*, pp. 59–72, 2007.

[41] J. Vesperman. *Essential CVS, 2nd Edition*. O'Reilly, 2006.

[42] J. M. Vidal. Multiagent coordination using a distributed combinatorial auction. In *AAAI Workshop on Auction Mechanisms for Robot Coordination*, 2006.

[43] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. KARMA: A Secure Economic Framework for P2P Resource Sharing. In *Proc. NetEcon*, 2003.

[44] R. White. Securing BGP through Secure Origin BGP. *Internet Protocol Journal*, 6(3), 2003.

[45] S. Zhong, J. Chen, and Y. R. Yang. Sprite: A simple, cheat-proof, credit-based system for mobile ad-hoc networks. In *Proc. Joint Conference of the IEEE Computer and Communication Societies (INFOCOM)*, 2003.

Sybil-Resilient Online Content Voting

Nguyen Tran, Bonan Min, Jinyang Li, and Lakshminarayanan Subramanian
New York University

Abstract

Obtaining user opinion (using votes) is essential to ranking user-generated online content. However, any content voting system is susceptible to the Sybil attack where adversaries can out-vote real users by creating many Sybil identities. In this paper, we present *SumUp*, a Sybil-resilient vote aggregation system that leverages the trust network among users to defend against Sybil attacks. SumUp uses the technique of *adaptive vote flow* aggregation to limit the number of bogus votes cast by adversaries to no more than the number of attack edges in the trust network (with high probability). Using user feedback on votes, SumUp further restricts the voting power of adversaries who continuously misbehave to below the number of their attack edges. Using detailed evaluation of several existing social networks (YouTube, Flickr), we show SumUp's ability to handle Sybil attacks. By applying SumUp on the voting trace of Digg, a popular news voting site, we have found strong evidence of attack on many articles marked "popular" by Digg.

1 Introduction

The Web 2.0 revolution has fueled a massive proliferation of user-generated content. While allowing users to publish information has led to democratization of Web content and promoted diversity, it has also made the Web increasingly vulnerable to content pollution from spammers, advertisers and adversarial users misusing the system. Therefore, the ability to rank content accurately is key to the survival and the popularity of many user-content hosting sites. Similarly, content rating is also indispensable in peer-to-peer file sharing systems to help users avoid mislabeled or low quality content [7, 16, 25].

People have long realized the importance of incorporating user opinion in rating online content. Traditional ranking algorithms such as PageRank [2] and HITS [12] rely on implicit user opinions reflected in the link structures of hypertext documents. For arbitrary content types, user opinion can be obtained in the form of explicit votes. Many popular websites today rely on user votes to rank news (Digg, Reddit), videos (YouTube), documents (Scribd) and consumer reviews (Yelp, Amazon).

Content rating based on users' votes is prone to vote manipulation by malicious users. Defending against vote manipulation is difficult due to the *Sybil attack* where the attacker can out-vote real users by creating many Sybil identities. The popularity of content-hosting sites has made such attacks very profitable as malicious entities can promote low-quality content to a wide audience. Successful Sybil attacks have been observed in the wild. For example, online polling on the best computer science school motivated students to deploy automatic scripts to vote for their schools repeatedly [9]. There are even commercial services that help paying clients promote their content to the top spot on popular sites such as YouTube by voting from a large number of Sybil accounts [22].

In this paper, we present SumUp, a Sybil-resilient online content voting system that prevents adversaries from arbitrarily distorting voting results. SumUp leverages the trust relationships that already exist among users (e.g. in the form of social relationships). Since it takes human efforts to establish a trust link, the attacker is unlikely to possess many attack edges (links from honest users to an adversarial identity). Nevertheless, he may create many links among Sybil identities themselves.

SumUp addresses the *vote aggregation problem* which can be stated as follows: *Given m votes on a given object, of which an arbitrary fraction may be from Sybil identities created by an attacker, how do we collect votes in a Sybil resilient manner?* A Sybil-resilient vote aggregation solution should satisfy three properties. First, the solution should collect a significant fraction of votes from honest users. Second, if the attacker has e_A attack edges, the maximum number of bogus votes should be bounded by e_A, independent of the attacker's ability to create many Sybil identities behind him. Third, if the attacker repeatedly casts bogus votes, his ability to vote in the future should be diminished. SumUp achieves all three properties with high probability in the face of Sybil attacks. The key idea in SumUp is the *adaptive vote flow* technique that appropriately assigns and adjusts link capacities in the trust graph to collect the net vote for an object.

Previous works have also exploited the use of trust networks to limit Sybil attacks [3, 15, 18, 26, 27, 30], but none directly addresses the vote aggregation problem. SybilLimit [26] performs admission control so that at most $O(\log n)$ Sybil identities are accepted per attack edge among n honest identities. As SybilLimit results in 10∼30 bogus votes per attack edge in a million-user system [26], SumUp provides notable improvement by limiting bogus votes to one per attack edge. Additionally, SumUp leverages user feedback to further diminish the voting power of adversaries that repeatedly vote maliciously.

In SumUp, each vote collector assigns capacities to links in the trust graph and computes a set of approximate max-flow paths from itself to all voters. Because only votes on paths with non-zero flows are counted, the number of bogus votes collected is limited by the total capacity of attack edges instead of links among Sybil identities. Typically, the number of voters on a given object is much smaller than the total user population (n). Based on this insight, SumUp assigns C_{max} units of capacity in total, thereby limiting the number of votes that can be collected to be C_{max}. SumUp adjusts C_{max} automatically according to the number of honest voters for each object so that it can aggregate a large fraction of votes from honest users. As C_{max} is far less than n, the number of bogus votes collected on a single object (i.e. the attack capacity) is no more than the number of attack edges (e_A). SumUp's security guarantee on bogus votes is probabilistic. If a vote collector happens to be close to an attack edge (a low probability event), the attack capacity could be much higher than e_A. By re-assigning link capacities using feedback, SumUp can restrict the attack capacity to be below e_A even if the vote collector happens to be close to some attack edges.

Using a detailed evaluation of several existing social networks (YouTube, Flickr), we show that SumUp successfully limits the number of bogus votes to the number of attack edges and is also able to collect $> 90\%$ of votes from honest voters. By applying SumUp to the voting trace and social network of Digg (an online news voting site), we have found hundreds of suspicious articles that have been marked "popular" by Digg. Based on manual sampling, we believe that at least 50% of suspicious articles exhibit strong evidence of Sybil attacks.

This paper is organized as follows. In Section 2, we discuss related work and in Section 3 we define the system model and the vote aggregation problem. Section 4 outlines the overall approach of SumUp and Sections 5 and 6 present the detailed design. In Section 7, we describe our evaluation results. Finally in Section 8, we discuss how to extend SumUp to decentralize setup and we conclude in Section 9.

2 Related Work

Ranking content is arguably one of the Web's most important problems. As users are the ultimate consumers of content, incorporating their opinions in the form of either explicit or implicit votes becomes an essential ingredient in many ranking systems. This section summarizes related work in vote-based ranking systems. Specifically, we examine how existing systems cope with Sybil attacks [6] and compare their approaches to SumUp.

2.1 Hyperlink-based ranking

PageRank [2] and HITS [12] are two popular ranking algorithms that exploit the implicit human judgment embedded in the hyperlink structure of web pages. A hyperlink from page A to page B can be viewed as an implicit endorsement (or vote) of page B by the creator of page A. In both algorithms, a page has a higher ranking if it is linked to by more pages with high rankings. Both PageRank and HITS are vulnerable to Sybil attacks. The attacker can significantly amplify the ranking of a page A by creating many web pages that link to each other and also to A. To mitigate this attack, the ranking system must probabilistically reset its PageRank computation from a small set of trusted web pages with probability ϵ [20]. Despite probabilistic resets, Sybil attacks can still amplify the PageRank of an attacker's page by a factor of $1/\epsilon$ [29], resulting in a big win for the attacker because ϵ is small.

2.2 User Reputation Systems

A user reputation system computes a reputation value for each identity in order to distinguish well-behaved identities from misbehaving ones. It is possible to use a user reputation system for vote aggregation: the voting system can either count votes only from users whose reputations are above a threshold or weigh each vote using the voter's reputation. Like SumUp, existing reputation systems mitigate attacks by exploiting two resources: the trust network among users and explicit user feedback on others' behaviors. We discuss the strengths and limitations of existing reputation systems in the context of vote aggregation and how SumUp builds upon ideas from prior work.

Feedback based reputations In EigenTrust [11] and Credence [25], each user independently computes *personalized* reputation values for all users based on past transactions or voting histories. In EigenTrust, a user increases (or decreases) another user's rating upon a good (or bad) transaction. In Credence [25], a user gives a high (or low) rating to another user if their voting records on the same set of file objects are similar (or dissimilar). Because not all pairs of users are known to each other based on direct interaction or votes on overlapping sets of objects, both Credence and EigenTrust use a PageRank-style algorithm to propagate the reputations of known users in order to calculate the reputations of unknown users. As such, both systems suffer from the same vulnerability as PageRank where an attacker can amplify the reputation of a Sybil identity by a factor of $1/\epsilon$.

Neither EigenTrust nor Credence provide provable guarantees on the damage of Sybil attacks under arbitrary attack strategies. In contrast, SumUp bounds the voting power of an attacker on a single object to be no more than the number of attack edges he possesses irrespective of the attack strategies in use. SumUp uses only negative feedback as opposed to EigenTrust and Credence that use both positive and negative feedback. Using only negative feedback has the advantage that an attacker cannot boost his attack capacity easily by casting correct votes on objects that he does not care about.

DSybil [28] is a feedback-based recommendation system that provides provable guarantees on the damages of arbitrary attack strategies. DSybil differs from SumUp in its goals. SumUp is a vote aggregation system which allows for arbitrary ranking algorithms to incorporate collected votes to rank objects. For example, the ranking algorithm can rank objects by the number of votes collected. In contrast, DSybil's recommendation algorithm is fixed: it recommends a *random* object among all objects whose sum of the weighted vote count exceeds a certain threshold.

Trust network-based reputations A number of proposals from the semantic web and peer-to-peer literature rely on the trust network between users to compute reputations [3, 8, 15, 21, 30]. Like SumUp, these proposals exploit the fact that it is difficult for an attacker to obtain many trust edges from honest users because trust links reflect offline social relationships. Of the existing work, Advogato [15], Appleseed [30] and Sybilproof [3] are resilient to Sybil attacks in the sense that an attacker cannot boost his reputation by creating a large number of Sybil identities "behind" him. Unfortunately, a Sybil-resilient user reputation scheme does not directly translate into a Sybil-resilient voting system: Advogato only computes a non-zero reputation for a small set of identities, disallowing a majority of users from being able to vote. Although an attacker cannot improve his reputation with Sybil identities in Appleseed and Sybilproof, the reputation of Sybil identities is almost as good as that of the attacker's non-Sybil accounts. Together, these reputable Sybil identities can cast many bogus votes.

2.3 Sybil Defense using trust networks

Many proposals use trust networks to defend against Sybil attacks in the context of different applications: SybilGuard [27] and SybilLimit [26] help a node admit another node in a decentralized system such that the admitted node is likely to be an honest node instead of a Sybil identity. Ostra [18] limits the rate of unwanted communication that adversaries can inflict on honest nodes. Sybil-resilient DHTs [5, 14] ensure that DHT routing is correct in the face of Sybil attacks. Kaleidoscope [23] distributes proxy identities to honest clients while minimizing the chances of exposing them to the censor with many Sybil identities. SumUp builds on their insights and addresses a different problem, namely, aggregating votes for online content rating. Like SybilLimit, SumUp bounds the power of attackers according to the number of attack edges. In SybilLimit, each attack edge results in $O(\log n)$ Sybil identities accepted by honest nodes. In SumUp, each attack edge leads to at most one vote with high probability. Additionally, SumUp uses user feedback on bogus votes to further reduce the attack capacity to below the number of attack edges. The feedback mechanism of SumUp is inspired by Ostra [18].

3 The Vote Aggregation Problem

In this section, we outline the system model and formalize the vote aggregation problem that SumUp addresses.

System model: We describe SumUp in a centralized setup where a trusted central authority maintains all the information in the system and performs vote aggregation using SumUp in order to rate content. This centralized mode of operation is suitable for web sites such as Digg, YouTube and Facebook, where all users' votes and their trust relationships are collected and maintained by a single trusted entity. We describe how SumUp can be applied in a distributed setting in Section 8.

SumUp leverages the trust network among users to defend against Sybil attacks [3,15,26,27,30]. Each trust link is directional. However, the creation of each link requires the consent of both users. Typically, user i creates a trust link to j if i has an offline social relationship to j. Similar to previous work [18, 26], SumUp requires that links are difficult to establish. As a result, an attacker only possesses a small number of attack edges (e_A) from honest users to colluding adversarial identities. Even though e_A is small, the attacker can create many Sybil identities and link them to adversarial entities. We refer to votes from colluding adversaries and their Sybil identities as bogus votes.

SumUp aggregates votes from one or more trusted *vote collectors*. A trusted collector is required in order to break the symmetry between honest nodes and Sybil nodes [3]. SumUp can operate in two modes depending on the choice of trusted vote collectors. In *personalized vote aggregation*, SumUp uses each user as his own vote collector to collect the votes of others. As each user collects a different number of votes on the same object, she also has a different (personalized) ranking of content. In *global vote aggregation*, SumUp uses one or more pre-selected vote collectors to collect votes on behalf of all users. Global vote aggregation has the advantage of allowing for a single global ranking of all objects; however, its performance relies on the proper selection of trusted collectors.

Vote Aggregation Problem: Any identity in the trust network including Sybils can cast a vote on any object to express his opinion on that object. In the simplest case, each vote is either positive or negative (+1 or -1). Alternatively, to make a vote more expressive, its value can vary within a range with higher values indicating more favorable opinions. A vote aggregation system collects votes on a given object. Based on collected votes and various other features, a separate ranking system determines the final ranking of an object. The design of the final ranking system is outside the scope of this paper. However, we note that many ranking algorithms utilize *both* the number of votes and the average value of votes to determine an object's rank [2, 12]. Therefore, to enable arbitrary ranking algorithms, a vote aggregation system should collect

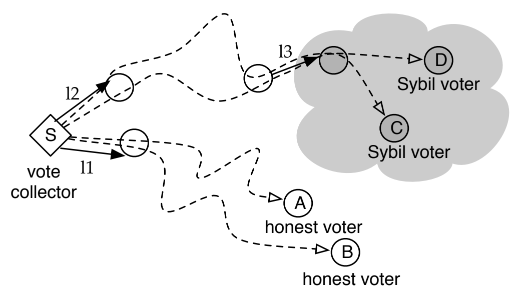


Figure 1: SumUp computes a set of approximate max-flow paths from the vote collector s to all voters (A,B,C,D). Straight lines denote trust links and curly dotted lines represent the vote flow paths along multiple links. Vote flow paths to honest voters are "congested" at links close to the collector while paths to Sybil voters are also congested at far-away attack edges.

a significant fraction of votes from honest voters.

A voting system can also let the vote collector provide *negative* feedback on malicious votes. In personalized vote aggregation, each collector gives feedback according to his personal taste. In global vote aggregation, the vote collector(s) should only provide objective feedback, e.g. negative feedback for positive votes on corrupted files. Such feedback is available for a very small subset of objects.

We describe the desired properties of a vote aggregation system. Let $G = (V, E)$ be a trust network with vote collector $s \in V$. V is comprised of an unknown set of honest users $V_h \subset V$ (including s) and the attacker controls all vertices in $V \setminus V_h$, many of which represent Sybil identities. Let e_A represent the number of attack edges from honest users in V_h to $V \setminus V_h$. Given that nodes in G cast votes on a specific object, a vote aggregation mechanism should achieve three properties:

1. Collect a large fraction of votes from honest users.
2. Limit the number of bogus votes from the attacker by e_A independent of the number of Sybil identities in $V \setminus V_h$.
3. Eventually ignore votes from nodes that repeatedly cast bogus votes using feedback.

4 Basic Approach

This section describes the intuition behind *adaptive vote flow* that SumUp uses to address the vote aggregation problem. The key idea of this approach is to appropriately assign link capacities to bound the attack capacity.

In order to limit the number of votes that Sybil identities can propagate for an object, SumUp computes a set of max-flow paths in the trust graph from the vote collector to all voters on a given object. Each vote flow consumes one unit of capacity along each link traversed. Figure 1 gives an example of the resulting flows from the collector s to voters A,B,C,D. When all links are assigned unit capacity, the attack capacity using the max-flow based approach is bounded by e_A.

The concept of max-flow has been applied in several reputation systems based on trust networks [3, 15]. When applied in the context of vote aggregation, the challenge is that links close to the vote collector tend to become "congested" (as shown in Figure 1), thereby limiting the total number of votes collected to be no more than the collector's node degree. Since practical trust networks are sparse with small median node degrees, only a few honest votes can be collected. We cannot simply enhance the capacity of each link to increase the number of votes collected since doing so also increases the attack capacity. Hence, a flow-based vote aggregation system faces the tradeoff between the maximum number of honest votes it can collect and the number of potentially bogus votes collected.

The *adaptive vote flow* technique addresses this tradeoff by exploiting two basic observations. First, the number of honest users voting for an object, even a popular one, is significantly smaller than the total number of users. For example, 99% of popular articles on Digg have fewer than 4000 votes which represents 1% of active users. Second, vote flow paths to honest voters tend to be only "congested" at links close to the vote collector while paths to Sybil voters are also congested at a few attack edges. When e_A is small, attack edges tend to be far away from the vote collector. As shown in Figure 1, vote flow paths to honest voters A and B are congested at the link l_1 while paths to Sybil identities C and D are congested at both l_2 and attack edge l_3.

The adaptive vote flow computation uses three key ideas. First, the algorithm restricts the maximum number of votes collected on an object to a value C_{max}. As C_{max} is used to assign the overall capacity in the trust graph, a small C_{max} results in less capacity for the attacker. SumUp can adaptively adjust C_{max} to collect a large fraction of honest votes on any given object. When the number of honest voters is $O(n^{\alpha})$ where $\alpha < 1$, the expected number of bogus votes is limited to $1 + o(1)$ per attack edge (Section 5.4).

The second important aspect of SumUp relates to *capacity assignment*, i.e. how to assign capacities to each trust link in order to collect a large fraction of honest votes and only a few bogus ones? In SumUp, the vote collector distributes C_{max} *tickets* downstream in a breadth-first search manner within the trust network. The capacity assigned to a link is the number of tickets distributed along the link plus one. As Figure 2 illustrates, the ticket distribution process introduces a *vote envelope* around the vote collector s; beyond the envelope all links have capacity 1. The vote envelope contains C_{max} nodes that can be viewed as entry points. There is enough capacity within the envelope to collect C_{max} votes from entry points. On the other hand, an attack edge beyond the envelope can propagate at most 1 vote regardless of the number of Sybil

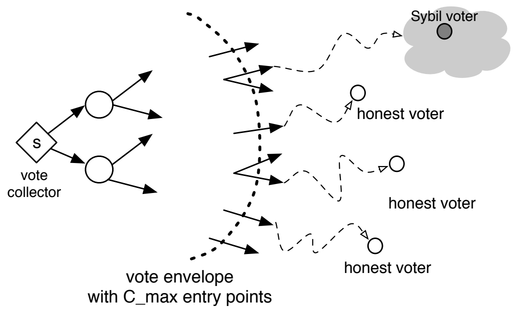


Figure 2: Through ticket distribution, SumUp creates a vote envelope around the collector. The capacities of links beyond the envelope are assigned to be one, limiting the attack capacity to be at most one per attack edge for adversaries outside this envelope. There is enough capacity within the envelope, such that nodes inside act like entry points for outside voters.

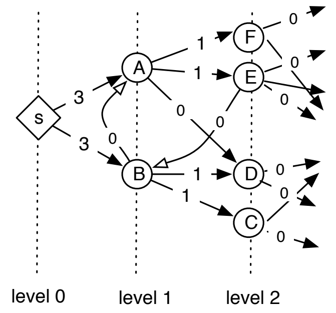


Figure 3: Each link shows the number of tickets distributed to that link from s (C_{max}=6). A node consumes one ticket and distributes the remaining evenly via its outgoing links to the next level. Tickets are not distributed to links pointing to the same level (B→A), or to a lower level (E→B). The capacity of each link is equal to one plus the number of tickets.

identities behind that edge. SumUp re-distributes tickets based on feedback to deal with attack edges within the envelope.

The final key idea in SumUp is to leverage user feedback to penalize attack edges that continuously propagate bogus votes. One cannot penalize individual identities since the attacker may always propagate bogus votes using new Sybil identities. Since an attack edge is always present in the path from the vote collector to a malicious voter [18], SumUp re-adjusts capacity assignment across links to reduce the capacity of penalized attack edges.

5 SumUp Design

In this section, we present the basic capacity assignment algorithm that achieves two of the three desired properties discussed in Section 3: (a) Collect a large fraction of votes from honest users; (b) Restrict the number of bogus votes to one per attack edge with high probability. Later in Section 6, we show how to adjust capacity based on feedback to deal with repeatedly misbehaved adversarial nodes.

We describe how link capacities are assigned given a particular C_{max} in Section 5.1 and present a fast algorithm to calculate approximate max-flow paths in Section 5.2. In Section 5.3, we introduce an additional optimization strategy that prunes links in the trust network so as to reduce the number of attack edges. We formally analyze the security properties of SumUp in Section 5.4 and show how to adaptively set C_{max} in Section 5.5.

5.1 Capacity assignment

The goal of capacity assignment is twofold. On the one hand, the assignment should allow the vote collector to gather a large fraction of honest votes. On the other hand, the assignment should minimize the attack capacity such that $C_A \approx e_A$.

As Figure 2 illustrates, the basic idea of capacity assignment is to construct a vote envelope around the vote collector with at least C_{max} entry points. The goal is to minimize the chances of including an attack edge in the envelope and to ensure that there is enough capacity within the envelope so that all vote flows from C_{max} entry points can reach the collector.

We achieve this goal using a *ticket distribution* mechanism which results in decreasing capacities for links with increasing distance from the vote collector. The distribution mechanism is best described using a propagation model where the vote collector is to spread C_{max} tickets across all links in the trust graph. Each ticket corresponds to a capacity value of 1. We associate each node with a level according to its shortest path distance from the vote collector, s. Node s is at level 0. Tickets are distributed to nodes one level at a time. If a node at level l has received t_{in} tickets from nodes at level $l-1$, the node consumes one ticket and re-distributes the remaining tickets evenly across all its outgoing links to nodes at level $l+1$, i.e. $t_{out} = t_{in} - 1$. The capacity value of each link is set to be one plus the number of tickets distributed on that link. Tickets are not distributed to links connecting nodes at the same level or from a higher to lower level. The set of nodes with positive incoming tickets fall within the vote envelope and thus represent the entry points.

Ticket distribution ensures that all C_{max} entry points have positive vote flows to the vote collector. Therefore, if there exists an edge-independent path connecting one of the entry points to an outside voter, the corresponding vote can be collected. We show in Section 5.4 that such a path exists with good probability. When C_{max} is much smaller than the number of honest nodes (n), the vote envelope is very small. Therefore, all attack edges reside outside the envelope, resulting in $C_A \approx e_A$ with high probability.

Figure 3 illustrates an example of the ticket distribution process. The vote collector (s) is to distribute C_{max}=6 tickets among all links. Each node collects tickets from its lower level neighbors, keeps one to itself and re-distributes the rest evenly across all outgoing links to the next level. In Figure 3, s sends 3 tickets down each of its outgoing links. Since A has more outgoing links (3) than its remaining tickets (2), link A→D receives no tickets. Tickets are not distributed to links between nodes at the same level (B→A) or to links from a higher to lower level (E→B). The final number of tickets distributed on each link is shown in Figure 3. Except for immediate outgoing edges from the vote collector, the capacity value of each link is equal to the amount of tickets it receives plus one.

5.2 Approximate Max-flow calculation

Once capacity assignment is done, the task remains to calculate the set of max-flow paths from the vote collector to all voters on a given object. It is possible to use existing max-flow algorithms such as Ford-Fulkerson and Preflow push [4] to compute vote flows. Unfortunately, these existing algorithms require $O(E)$ running time to find each vote flow, where E is the number of edges in the graph. Since vote aggregation only aims to collect a large fraction of honest votes, it is not necessary to compute exact max-flow paths. In particular, we can exploit the structure of capacity assignment to compute a set of approximate vote flows in $O(\Delta)$ time, where Δ is the diameter of the graph. For expander-like networks, $\Delta = O(\log n)$. For practical social networks with a few million users, $\Delta \approx 20$.

Our approximation algorithm works incrementally by finding one vote flow for a voter at a time. Unlike the classic Ford-Fulkerson algorithm, our approximation performs a greedy search from the voter to the collector in $O(\Delta)$ time instead of a breadth-first-search from the collector which takes $O(E)$ running time. Starting at a voter, the greedy search strategy attempts to explore a node at a lower level if there exists an incoming link with positive capacity. Since it is not always possible to find such a candidate for exploration, the approximation algorithm allows a threshold (t) of non-greedy steps which explores nodes at the same or a higher level. Therefore, the number of nodes visited by the greedy search is bounded by $(\Delta + 2t)$. Greedy search works well in practice. For links within the vote envelope, there is more capacity for lower-level links and hence greedy search is more likely to find a non-zero capacity path by exploring lower-level nodes. For links outside the vote envelope, greedy search results in short paths to one of the vote entry points.

5.3 Optimization via link pruning

We introduce an optimization strategy that performs link pruning to reduce the number of attack edges, thereby reducing the attack capacity. Pruning is performed prior to link capacity assignment and its goal is to bound the in-degree of each node to a small value, d_{in_thres}. As a result, the number of attack edges is reduced if some adversarial nodes have more than d_{in_thres} incoming edges from honest nodes. We speculate that the more honest neighbors an adversarial node has, the easier for it to trick an honest node into trusting it. Therefore, the number of attack edges in the pruned network is likely to be smaller than those in the original network. On the other hand, pruning is unlikely to affect honest users since each honest node only attempts to cast one vote via one of its incoming links.

Since it is not possible to accurately discern honest identities from Sybil identities, we give all identities the chance to have their votes collected. In other words, pruning should never disconnect a node. The minimally connected network that satisfies this requirement is a tree rooted at the vote collector. A tree topology minimizes attack edges but is also overly restrictive for honest nodes because each node has exactly one path from the collector: if that path is saturated, a vote cannot be collected. A better tradeoff is to allow each node to have at most $d_{in\_thres} > 1$ incoming links in the pruned network so that honest nodes have a large set of diverse paths while limiting each adversarial node to only d_{in_thres} attack edges. We examine the specific parameter choice of d_{in_thres} in Section 7.

Pruning each node to have at most d_{in_thres} incoming links is done in several steps. First, we remove all links except those connecting nodes at a lower level (l) to neighbors at the next level $(l + 1)$. Next, we remove a subset of incoming links at each node so that the remaining links do not exceed d_{in_thres}. In the third step, we add back links removed in step one for nodes with fewer than d_{in_thres} incoming links. Finally, we add one outgoing link back to nodes that have no outgoing links after step three, with priority given to links going to the next level. By preferentially preserving links from lower to higher levels, pruning does not interfere with SumUp's capacity assignment and flow computation.

5.4 Security Properties

This section provides a formal analysis of the security properties of SumUp assuming an expander graph. Various measurement studies have shown that social networks are indeed expander-like [13]. The link pruning optimization does not destroy a graph's expander property because it preserves the level of each node in the original graph.

Our analysis provides bounds on the expected attack capacity, C_A, and the expected fraction of votes collected if C_{max} honest users vote. The average-case analysis assumes that each attack edge is a random link in the graph. For personalized vote aggregation, the expectation is taken over all vote collectors which include all honest nodes. In the unfortunate but rare scenario where an adversarial node is close to the vote collector, we can use feedback to re-adjust link capacities (Section 6).

Theorem 5.1 *Given that the trust network G on n nodes is a bounded degree expander graph, the expected capacity per attack edge is* $\frac{E(C_A)}{e_A} = 1 + O(\frac{C_{max}}{n} \log C_{max})$

which is $1 + o(1)$ *if* $C_{max} = O(n^{\alpha})$ *for* $\alpha < 1$. *If* $e_A \cdot C_{max} \ll n$*, the capacity per attack edge is bounded by* 1 *with high probability.*

Proof Sketch Let L_i represent the number of nodes at level i with $L_0 = 1$. Let E_i be the number of edges pointing from level $i-1$ to level i. Notice that $E_i \geq L_i$. Let T_i be the number of tickets propagated from level $i-1$ to i with $T_0 = C_{max}$. The number of tickets at each level is reduced by the number of nodes at the previous level (i.e. $T_i = T_{i-1} - L_{i-1}$). Therefore, the number of levels with non-zero tickets is at most $O(log(C_{max}))$ as L_i grows exponentially in an expander graph. For a randomly placed attack edge, the probability of its being at level i is at most L_i/n. Therefore, the expected capacity of a random attack edge can be calculated as $1 + \sum_i (\frac{L_i}{n} \cdot \frac{T_i}{E_i}) < 1 + \sum_i (\frac{L_i}{n} \cdot \frac{C_{max}}{L_i}) = 1 + O(\frac{C_{max}}{n} \log C_{max})$. Therefore, if $C_{max} = O(n^{\alpha})$ for $\alpha < 1$, the expected attack capacity per attack edge is $1 + o(1)$.

Since the number of nodes within the vote envelope is at most C_{max}, the probability of a random attack edge being located outside the envelope is $1 - \frac{C_{max}}{n}$. Therefore, the probability that any of the e_A attack edges lies within the vote envelope is $1 - (1 - \frac{C_{max}}{n})^{e_A} < \frac{e_A \cdot C_{max}}{n}$. Hence, if $e_A \cdot C_{max} = n^{\alpha}$ where $\alpha < 1$, the attack capacity is bounded by 1 with high probability.

Theorem 5.1 is for expected capacity per attack edge. In the worse case when the vote collector is adjacent to some adversarial nodes, the attack capacity can be a significant fraction of C_{max}. Such rare worst case scenarios are addressed in Section 6.

Theorem 5.2 *Given that the trust network* G *on* n *nodes is a d-regular expander graph, the expected fraction of votes that can be collected out of* C_{max} *honest voters is* $\frac{d-\lambda_2}{d}(1 - \frac{C_{max}}{n})$ *where* λ_2 *is the second largest eigenvalue of the adjacency matrix of G.*

Proof Sketch SumUp creates a vote envelop consisting of C_{max} entry points via which votes are collected. To prove that there exists a large fraction of vote flows, we argue that the minimum cut of the graph between the set of C_{max} entry points and an arbitrary set of C_{max} honest voters is large.

Expanders are well-connected graphs. In particular, the Expander mixing lemma [19] states that for any set S and T in a d-regular expander graph, the expected number of edges between S and T is $(d - \lambda_2)|S| \cdot |T|/n$, where λ_2 is the second largest eigenvalue of the adjacency matrix of G. Let S be a set of nodes containing C_{max} entry points and T be a set of nodes containing C_{max} honest voters, thus $|S| + |T| = n$ and $|S| \geq C_{max}, |T| \geq C_{max}$. Therefore, the min-cut value between S and T is $= (d - \lambda_2)|S| \cdot |T|/n \geq (d - \lambda_2) \cdot C_{max}(n - C_{max})/n$. The number of vote flows between S and T is at least $1/d$ of the min-cut value because each vote flow only uses one of an honest voter's d incoming links. Therefore, the fraction of votes that can be collected is at least $(d - \lambda_2) \cdot C_{max}(n - C_{max})/(n \cdot d \cdot C_{max}) = \frac{d-\lambda_2}{d}(1 - \frac{C_{max}}{n})$. For well-connected graphs like expanders, λ_2 is well separated from d, so that a significant fraction of votes can be collected.

5.5 Setting C_{max} adaptively

When n_v honest users vote on an object, SumUp should ideally set C_{max} to be n_v in order to collect a large fraction of honest votes on that object. In practice, n_v/n is very small for any object, even a very popular one. Hence, $C_{max} = n_v \ll n$ and the expected capacity per attack edge is 1. We note that even if $n_v \approx n$, the attack capacity is still bounded by $O(\log n)$ per attack edge.

It is impossible to precisely calculate the number of honest votes (n_v). However, we can use the actual number of votes collected by SumUp as a lower bound estimate for n_v. Based on this intuition, SumUp adaptively sets C_{max} according to the number of votes collected for each object. The adaptation works as follows: For a given object, SumUp starts with a small initial value for C_{max}, e.g. $C_{max} = 100$. Subsequently, if the number of actual votes collected exceeds ρC_{max} where ρ is a constant less than 1, SumUp doubles the C_{max} in use and re-runs the capacity assignment and vote collection procedures. The doubling of C_{max} continues until the number of collected votes becomes less than ρC_{max}.

We show that this adaptive strategy is robust, i.e. the maximum value of the resulting C_{max} will not dramatically exceed n_v regardless of the number of bogus votes cast by adversarial nodes. Since adversarial nodes attempt to cast enough bogus votes to saturate attack capacity, the number of votes collected is at most $n_v + C_A$ where $C_A = e_A(1 + \frac{C_{max}}{n} \log C_{max})$. The doubling of C_{max} stops when the number of collected votes is less than ρC_{max}. Therefore, the maximum value of C_{max} that stops the adaptation is one that satisfies the following inequality:

$$n_v + e_A(1 + \frac{C_{max}}{n} \log C_{max}) < \rho C_{max}$$

Since $\log C_{max} \leq \log n$, the adaptation terminates with $C'_{max} = (n_v + e_A)/(\rho - \frac{\log n}{n})$. As $\rho \gg \frac{\log n}{n}$, we derive $C'_{max} = \frac{1}{\rho}(n_v + e_A)$. The adaptive strategy doubles C_{max} every iteration, hence it overshoots by at most a factor of two. Therefore, the resulting C_{max} found is $C_{max} = \frac{2}{\rho}(n_v + e_A)$. As we can see, the attacker can only affect the C_{max} found by an additive factor of e_A. Since e_A is small, the attacker has negligible influence on the C_{max} found.

The previous analysis is done for the expected case with random attack edges. Even in a worst case scenario where some attack edges are very close to the vote collector, the adaptive strategy is still resilient against manipulation. In the worst case scenario, the attack capacity is proportional to C_{max}, i.e. $C_A = xC_{max}$. Since no vote aggregation scheme can defend against an attacker who controls a majority of immediate links from the vote collector, we are only interested in the case where $x < 0.5$. The adaptive strategy stops increasing C_{max} when $n_v + xC_{max} < \rho C_{max}$, thus resulting in $C_{max} \le \frac{2n_v}{\rho - x}$. As we can see, ρ must be greater than x to prevent the attacker from causing SumUp to increase C_{max} to infinity. Therefore, we set $\rho = 0.5$ by default.

6 Leveraging user feedback

The basic design presented in Section 5 does not address the worst case scenario where C_A could be much higher than e_A. Furthermore, the basic design only bounds the number of bogus votes collected on a single object. As a result, adversaries can still cast up to e_A bogus votes on *every* object in the system. In this section, we utilize feedback to address both problems.

SumUp maintains a penalty value for each link and uses the penalty in two ways. First, we adjust each link's capacity assignment so that links with higher penalties have lower capacities. This helps reduce C_A when some attack edges happen to be close to the vote collector. Second, we eliminate links whose penalties have exceeded a certain threshold. Therefore, if adversaries continuously misbehave, the attack capacity will drop below e_A over time. We describe how SumUp calculates and uses penalty in the rest of the section.

6.1 Incorporating negative feedback

The vote collector can choose to associate negative feedback with voters if he believes their votes are malicious. Feedback may be performed for a very small set of objects-for example, when the collector finds out that an object is a bogus file or a virus.

SumUp keeps track of a penalty value, p_i, for each link i in the trust network. For each voter receiving negative feedback, SumUp increments the penalty values for all links along the path to that voter. Specifically, if the link being penalized has capacity c_i, SumUp increments the link's penalty by $1/c_i$. Scaling the increment by c_i is intuitive; links with high capacities are close to the vote collector and hence are more likely to propagate some bogus votes even if they are honest links. Therefore, SumUp imposes a lesser penalty on high capacity links.

It is necessary to penalize *all* links along the path instead of just the immediate link to the voter because that voter might be a Sybil identity created by some other attacker along the path. Punishing a link to a Sybil identity is useless as adversaries can easily create more such links. This way of incorporating negative feedback is inspired by Ostra [18]. Unlike Ostra, SumUp uses a customized flow network per vote collector and only allows the collector to incorporate feedback for its associated network in order to ensure that feedback is always trustworthy.

6.2 Capacity adjustment

The capacity assignment in Section 5.1 lets each node distribute incoming tickets evenly across all outgoing links. In the absence of feedback, it is reasonable to assume that all outgoing links are equally trustworthy and hence to assign them the same number of tickets. When negative feedback is available, a node should distribute fewer tickets to outgoing links with higher penalty values. Such adjustment is particularly useful in circumstances where adversaries are close to the vote collector and hence might receive a large number of tickets.

The goal of capacity adjustment is to compute a weight, $w(p_i)$, as a function of the link's penalty. The number of tickets a node distributes to its outgoing link i is proportional to the link's weight, i.e. $t_i = t_{out} * w(p_i)/\sum_{\forall i \in nbrs} w(p_i)$. The question then becomes how to compute $w(p_i)$. Clearly, a link with a high penalty value should have a smaller weight, i.e. $w(p_i) < w(p_j)$ if $p_i > p_i$. Another desirable property is that if the penalties on two links increase by the same amount, the ratio of their weights remains unchanged. In other words, the weight function should satisfy: $\forall p', p_i, p_j, \quad \frac{w(p_i)}{w(p_j)} = \frac{w(p_i+p')}{w(p_j+p')}$. This requirement matches our intuition that if two links have accumulated the same amount of additional penalties over a period of time, the relative capacities between them should remain the same. Since the exponential function satisfies both requirements, we use $w(p_i) = 0.2^{p_i}$ by default.

6.3 Eliminating links using feedback

Capacity adjustment cannot reduce the attack capacity to below e_A since each link is assigned a minimum capacity value of one. To further reduce e_A, we eliminate those links that received high amounts of negative feedback.

We use a heuristic for link elimination: we remove a link if its penalty exceeds a threshold value. We use a default threshold of five. Since we already prune the trust network (Section 5.3) before performing capacity assignment, we add back a previously pruned link if one exists after eliminating an incoming link. The reason why link elimination is useful can be explained intuitively: if adversaries continuously cast bogus votes on different objects over time, all attack edges will be eliminated eventually. On the other hand, although an honest user might have one of its incoming links eliminated because of a downstream attacker casting bad votes, he is unlikely to experience another elimination due to the same attacker since the attack edge connecting him to that attacker has also been eliminated. Despite this intuitive argument, there always exist pathological scenarios where link elimination affects some honest users, leaving them with no voting

Network	Nodes $\times 1000$	Edges $\times 1000$	Degree **50%(90%)**	Directed?
YouTube [18]	446	3,458	2 (12)	No
Flickr [17]	1,530	21,399	1 (15)	Yes
Synthetic [24]	3000	24,248	6 (15)	No

Table 1: Statistics of the social network traces or synthetic model used for evaluating SumUp. All statistics are for the strongly connected component (SCC).

power. To address such potential drawbacks, we re-enact eliminated links at a slow rate over time. We evaluate the effect of link elimination in Section 7.

7 Evaluation

In this section, we demonstrate SumUp's security property using real-world social networks and voting traces. Our key results are:

1. For all networks under evaluation, SumUp bounds the average number of bogus votes collected to be no more than e_A while being able to collect >90% of honest votes when less than 1% of honest users vote.
2. By incorporating feedback from the vote collector, SumUp dramatically cuts down the attack capacity for adversaries that continuously cast bogus votes.
3. We apply SumUp to the voting trace and social network of Digg [1], a news aggregation site that uses votes to rank user-submitted news articles. SumUp has detected hundreds of suspicious articles that have been marked as "popular" by Digg. Based on manual sampling, we believe at least 50% of suspicious articles found by SumUp exhibit strong evidence of Sybil attacks.

7.1 Experimental Setup

For the evaluation, we use a number of network datasets from different online social networking sites [17] as well as a synthetic social network [24] as the underlying trust network. SumUp works for different types of trust networks as long as an attacker cannot obtain many attack edges easily in those networks. Table 1 gives the statistics of various datasets. For undirected networks, we treat each link as a pair of directed links. Unless explicitly mentioned, we use the YouTube network by default.

To evaluate the Sybil-resilience of SumUp, we inject $e_A = 100$ attack edges by adding 10 adversarial nodes each with links from 10 random honest nodes in the network. The attacker always casts the maximum bogus votes to saturate his capacity. Each experimental run involves a randomly chosen vote collector and a subset of nodes which serve as honest voters. SumUp adaptively adjusts C_{max} using an initial value of 100 and $\rho = 0.5$. By default, the threshold of allowed non-greedy steps is 20. We plot the average statistic across five experimental runs in all graphs. In Section 7.6, we apply SumUp on the real world voting trace of Digg to examine how SumUp can be used to resist Sybil attacks in the wild.

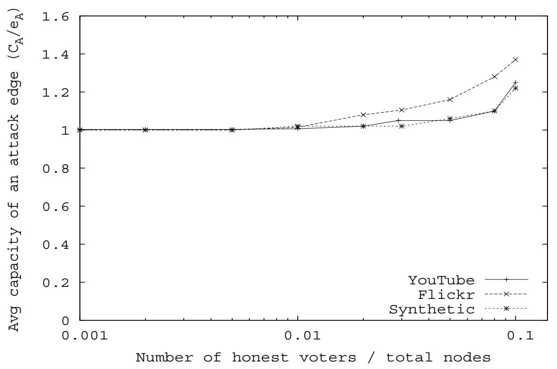


Figure 4: The average capacity per attack edge as a function of the fraction of honest nodes that vote. The average capacity per attack edge remains close to 1, even if $1/10$ of honest nodes vote.

7.2 Sybil-resilience of the basic design

The main goal of SumUp is to limit attack capacity while allowing honest users to vote. Figure 4 shows that the average attack capacity per attack edge remains close to 1 even when the number of honest voters approaches 10%. Furthermore, as shown in Figure 5, SumUp manages to collect more than 90% of all honest votes in all networks. Link pruning is disabled in these experiments. The three networks under evaluation have very different sizes and degree distributions (see Table 1). The fact that all three networks exhibit similar performance suggests that SumUp is robust against the topological details. Since SumUp adaptively sets C_{max} in these experiments, the results also confirm that adaptation works well in finding a C_{max} that can collect most of the honest votes without significantly increasing attack capacity. We point out that the results in Figure 4 correspond to a random vote collector. For an unlucky vote collector close to an attack edge, he may experience a much larger than average attack capacity. In personalized vote collection, there are few unlucky collectors. These unlucky vote collectors need to use their own feedback on bogus votes to reduce attack capacity.

Benefits of pruning: The link pruning optimization, introduced in Section 5.3, further reduces the attack capacity by capping the number of attack edges an adversarial node can have. As Figure 6 shows, pruning does not affect the fraction of honest votes collected if the threshold d_{in_thres} is greater than 3. Figure 6 represents data from the YouTube network and the results for other networks are similar. SumUp uses the default threshold (d_{in_thres}) of 3. Figure 7 shows that the average attack capacity is greatly reduced when adversarial nodes have more than 3 attack edges. Since pruning attempts to restrict each node to at most 3 incoming links, additional attack edges are excluded from vote flow computation.

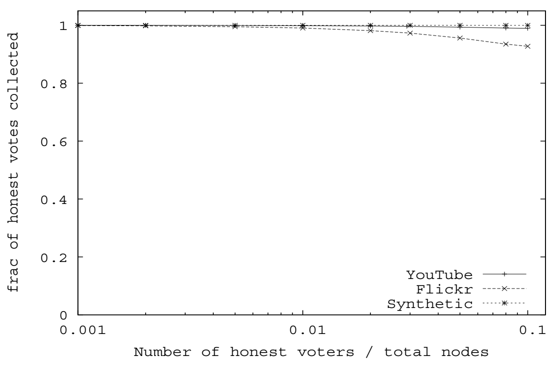


Figure 5: The fraction of votes collected as a function of fraction of honest nodes that vote. SumUp collects more than 80% votes, even $1/10$ honest nodes vote.

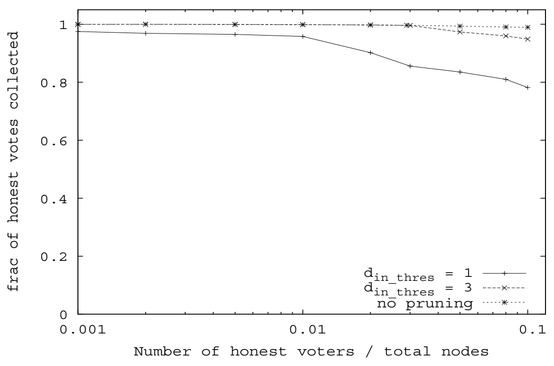


Figure 6: The fraction of votes collected for different d_{in_thres} (YouTube graph). More than 90% votes are collected when $d_{in\_thres} = 3$.

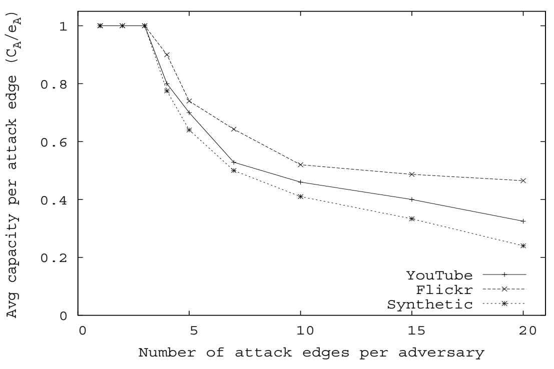


Figure 7: Average attack capacity per attack edge decreases as the number of attack edges per adversary increases.

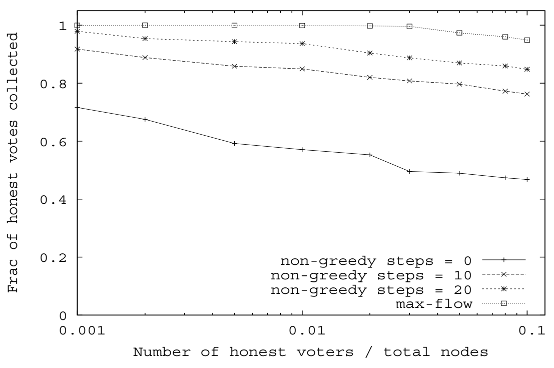


Figure 8: The fraction of votes collected for different threshold for non-greedy steps. More than 70% votes are collected even with a small threshold (10) for non-greedy steps.

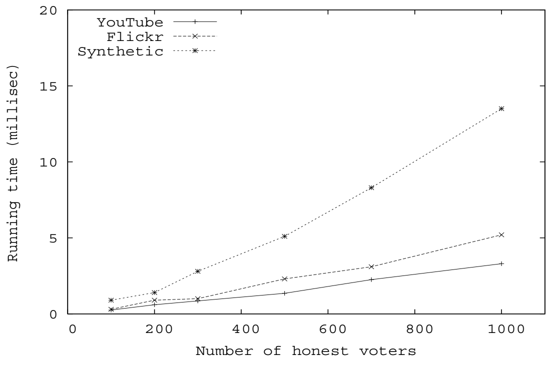


Figure 9: The running time of one vote collector gathering up to 1000 votes. The Ford-Fulkerson max-flow algorithm takes 50 seconds to collect 1000 votes for the YouTube graph.

7.3 Effectiveness of greedy search

SumUp uses a fast greedy algorithm to calculate approximate max vote flows to voters. Greedy search enables SumUp to collect a majority of votes while using a small threshold (t) of non-greedy steps. Figure 8 shows the fraction of honest votes collected for the pruned YouTube graph. As we can see, with a small threshold of 20, the fraction of votes collected is more than 80%. Even when disallowing non-greedy steps completely, SumUp manages to collect $> 40\%$ of votes.

Figure 9 shows the running time of greedy-search for different networks. The experiments are performed on a single machine with an AMD Opteron 2.5GHz CPU and 8GB memory. SumUp takes around 5ms to collect 1000 votes from a single vote collector on YouTube and Flickr. The synthetic network incurs more running time as its links are more congested than those in YouTube and Flickr. The average non-greedy steps taken in the synthetic network is 6.5 as opposed to 0.8 for the YouTube graph. Greedy-search dramatically reduces the flow computation time. As a comparison, the Ford-Fulkerson max-flow algorithm requires 50 seconds to collect 1000 votes

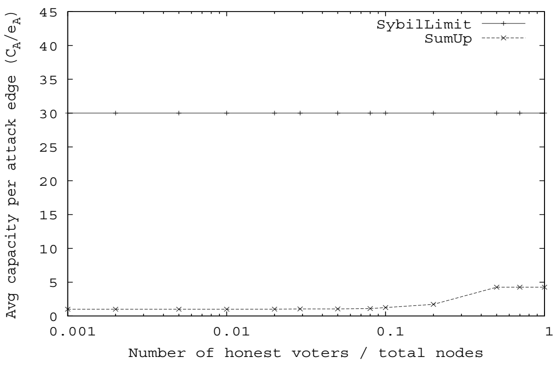


Figure 10: Average attack capacity per attack edge as a function of voters. SumUp is better than SybilLimit in the average case.

for the YouTube graph.

7.4 Comparison with SybilLimit

SybilLimit is a node admission protocol that leverages the trust network to allow an honest node to accept other honest nodes with high probability. It bounds the number of Sybil nodes accepted to be $O(\log n)$. We can apply SybilLimit for vote aggregation by letting each vote collector compute a fixed set of accepted users based on the trust network. Subsequently, a vote is collected if and only if it comes from one of the accepted users. In contrast, SumUp does not calculate a fixed set of allowed users; rather, it dynamically determines the set of voters that count toward each object. Such dynamic calculation allows SumUp to settle on a small C_{max} while still collecting most of the honest votes. A small C_{max} allows SumUp to bound attack capacity by e_A.

Figure 10 compares the average attack capacity in SumUp to that of SybilLimit for the un-pruned YouTube network. The attack capacity in SybilLimit refers to the number of Sybil nodes that are accepted by the vote collector. Since SybilLimit aims to accept nodes instead of votes, its attack capacity remains $O(\log n)$ regardless of the number of actual honest voters. Our implementation of SybilLimit uses the optimal set of parameters ($w = 15$, $r = 3000$) we determined manually. As Figure 10 shows, while SybilLimit allows 30 bogus votes per attack edge, SumUp results in approximately 1 vote per attack edge when the fraction of honest voters is less than 10%. When all nodes vote, SumUp leads to much lower attack capacity than SybilLimit even though both have the same $O(\log n)$ asymptotic bound per attack edge. This is due to two reasons. First, SumUp's bound of $1 + \log n$ in Theorem 5.1 is a loose upper bound of the actual average capacity. Second, since links pointing to lower-level nodes are not eligible for ticket distribution, many incoming links of an adversarial nodes have zero tickets and thus are assigned capacity of one.

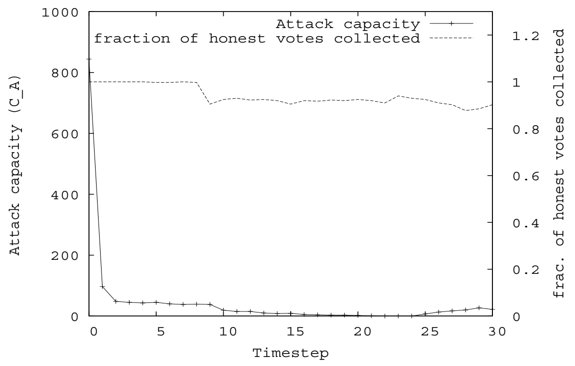


Figure 11: The change in attack capacity as adversaries continuously cast bogus votes (YouTube graph). Capacity adjustment and link elimination dramatically reduce C_A while still allowing SumUp to collect more than 80% of the honest votes.

7.5 Benefits of incorporating feedback

We evaluate the benefits of capacity adjustment and link elimination when the vote collector provides feedback on the bogus votes collected. Figure 11 corresponds to the worst case scenario where one of the vote collector's four outgoing links is an attack edge. At every time step, there are 400 random honest users voting on an object and the attacker also votes with its maximum capacity. When collecting votes on the first object at time step 1, adaption results in $C_{max} = \frac{2n_v}{\rho - x} = 3200$ because $n_v = 400, \rho = 0.5, x = 1/4$. Therefore, the attacker manages to cast $\frac{1}{4}C_{max} = 800$ votes and outvote honest users. After incorporating the vote collector's feedback after the first time step, the adjacent attack edge incurs a penalty of 1 which results in drastically reduced C_A (97). If the vote collector continues to provide feedback on malicious votes, 90% of attack edges are eliminated after only 12 time steps. After another 10 time steps, all attack edges are eliminated, reducing C_A to zero. However, because of our decision to slowly add back eliminated links, the attack capacity doesn't remains at zero forever. Figure 11 also shows that link elimination has little effects on honest nodes as the fraction of honest votes collected always remains above 80%.

7.6 Defending Digg against Sybil attacks

In this section, we ask the following questions: Is there evidence of Sybil attacks in real world content voting systems? Can SumUp successfully limit bogus votes from Sybil identities? We apply SumUp to the voting trace and social network crawled from Digg to show the real world benefits of SumUp.

Digg [1] is a popular news aggregation site where any registered user can submit an article for others to vote on. A positive vote on an article is called a *digg*. A negative vote is called a *bury*. Digg marks a subset of submitted articles as "popular" articles and displays them on its front page. In subsequent discussions, we use the terms *pop-*

Number of Nodes	3,002,907
Number of Edges	5,063,244
Number of Nodes in SCC	466,326
Number of Edges in SCC	4,908,958
Out degree avg(50%, 90%)	10(1, 9)
In degree avg(50%, 90%)	10(2, 11)
Number of submitted (popular) articles 2004/12/01-2008/09/21	6,494,987 (137,480)
Diggs on all articles avg(50%, 90%)	24(2, 15)
Diggs on popular articles avg(50%, 90%)	862(650, 1810)
Hours since submission before a popular article is marked as popular. avg (50,%,90%)	16(13, 23)
Number of submitted (popular) articles with *bury* data available 2008/08/13-2008/09/15	38,033 (5,794)

Table 2: Basic statistics of the crawled Digg dataset. The strongly connected component (SCC) of Digg consists of 466,326 nodes.

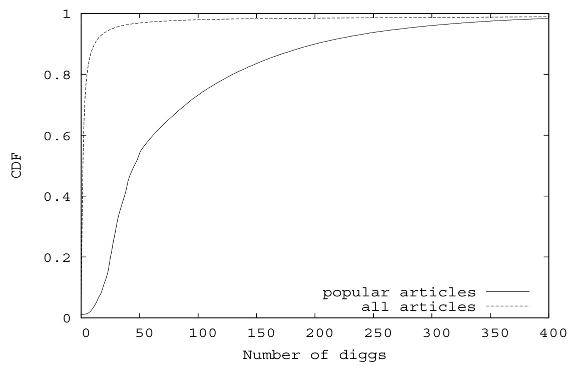


Figure 12: Distribution of diggs for all popular articles before being marked as popular and for all articles within 24 hours after submission.

ular or *popularity* only to refer to the popularity status of an article as marked by Digg. A Digg user can create a "follow" link to another user if he wants to browse all articles submitted by that user. We have crawled Digg to obtain the voting trace on all submitted articles since Digg's launch (2004/12/01-2008/09/21) as well as the complete "follow" network between users. Unfortunately, unlike diggs, bury data is only available as a live stream. Furthermore, Digg does not reveal the user identity that cast a bury, preventing us from evaluating SumUp's feedback mechanism. We have been streaming bury data since 2008/08/13. Table 2 shows the basic statistics of the Digg "follow" network and the two voting traces, one with bury data and one without. Although the strongly connected component (SCC) consists of only 15% of total nodes, 88% of votes come from nodes in the SCC.

There is enormous incentive for an attacker to get a submitted article marked as popular, thus promoting it to the front page of Digg which has several million page views per day. Our goal is to apply SumUp on the voting trace to reduce the number of successful attacks on the popularity marking mechanism of Digg. Unfortunately, unlike experiments done in Section 7.2 and Section 7.5, there is no ground truth about which Digg users are adversaries. Instead, we have to use SumUp itself to find evidence of attacks and rely on manual sampling and other types of data to cross check the correctness of results.

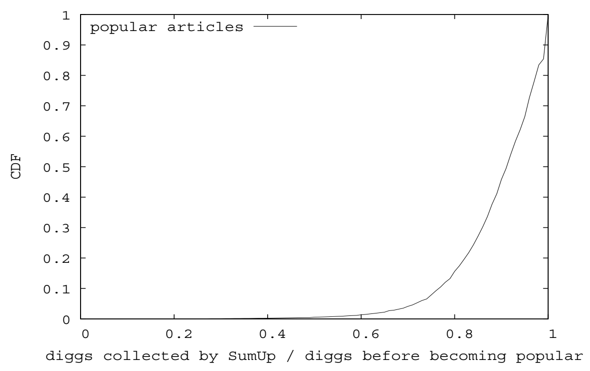


Figure 13: The distribution of the fraction of diggs collected by SumUp over all diggs before an article is marked as popular.

Digg's popularity ranking algorithm is intentionally not revealed to the public in order to mitigate gaming of the system. Nevertheless, we speculate that the number of diggs is a top contributor to an article's popularity status. Figure 12 shows the distribution of the number of diggs an article received before it was marked as popular. Since more than 90% of popular articles are marked as such within 24 hours after submission, we also plot the number of diggs received within 24 hours of submission for all articles. The large difference between the two distributions indicates that the number of diggs plays an important role in determining an article's popularity status.

Instead of simply adding up the actual number of diggs, what if Digg uses SumUp to collect all votes on an article? We use the identity of Kevin Rose, the founder of Digg, as the vote collector to aggregate all diggs on an article before it is marked as popular. Figure 13 shows the distribution of the fraction of votes collected by SumUp over all diggs before an article is marked as popular. Our previous evaluation on various network topologies suggests that SumUp should be able to collect at least 90% of all votes. However, in Figure 13, there are a fair number of popular articles with much fewer than the expected fraction of diggs collected. For example, SumUp only manages to collect less than 50% of votes for 0.5% of popular articles. We hypothesize that the reason for collecting fewer than the expected votes is due to real world Sybil attacks.

Since there is no ground truth data to verify whether

Threshold of the fraction of collected diggs	20%	30%	40%	50%
# of suspicious articles	41	131	300	800
Advertisement	5	4	2	1
Phishing	1	0	0	0
Obscure political articles	2	2	0	0
Many newly registered voters	11	7	8	10
Fewer than 50 total diggs	1	3	6	4
No obvious attack	10	14	14	15

Table 3: Manual classification of 30 randomly sampled suspicious articles. We use different thresholds of the fraction of collected diggs for marking suspicious articles. An article is labeled as having many new voters if $> 30\%$ of its votes are from users who registered on the same day as the article's submission date.

few collected diggs are indeed the result of attacks, we resort to manual inspection. We classify a popular article as suspicious if its fraction of diggs collected is less than a given threshold. Table 3 shows the result of manually inspecting 30 random articles out of all suspicious articles. The random samples for different thresholds are chosen independently. There are a number of obvious bogus articles such as advertisements, phishing articles and obscure political opinions. Of the remaining, we find many of them have an unusually large fraction (>30%) of new voters who registered on the same day as the article's submission time. Some articles also have very few total diggs since becoming popular, a rare event since an article typically receives hundreds of votes after being shown on the front page of Digg. We find no obvious evidence of attack for roughly half of the sampled articles. Interviews with Digg attackers [10] reveal that, although there is a fair amount of attack activities on Digg, attackers do not usually promote obviously bogus material. This is likely due to Digg being a highly monitored system with fewer than a hundred articles becoming popular every day. Instead, attackers try to help paid customers promote normal or even good content or to boost their profiles within the Digg community.

As further evidence that a lower than expected fraction of collected diggs signals a possible attack, we examine Digg's *bury* data for articles submitted after 2008/08/13, of which 5794 are marked as popular. Figure 14 plots the correlation between the average number of bury votes on an article *after* it became popular vs. the fraction of the diggs SumUp collected before it was marked as popular. As Figure 14 reveals, the higher the fraction of diggs collected by SumUp, the fewer bury votes an article received after being marked as popular. Assuming most bury votes come from honest users that genuinely dislike the article, a large number of bury votes is a good indicator that the article is of dubious quality.

What are the voting patterns for suspicious articles?

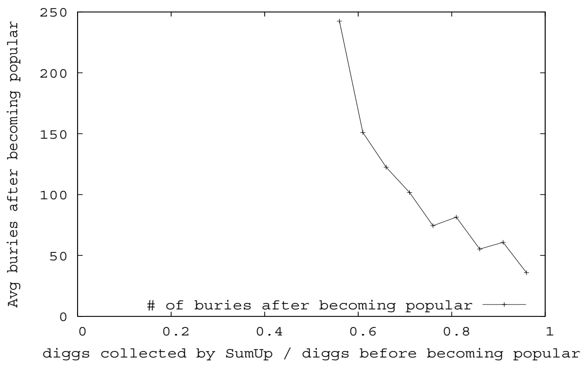


Figure 14: The average number of buries an article received *after* it was marked as popular as a function of the fraction of diggs collected by SumUp *before* it is marked as popular. The Figure covers $5,794$ popular articles with bury data available.

Since 88% diggs come from nodes within the SCC, we expect only 12% of diggs to originate from the rest of the network, which mostly consists of nodes with no incoming follow links. For most suspicious articles, the reason that SumUp collecting fewer than expected diggs is due to an unusually large fraction of votes coming from outside the SCC component. Since Digg's popularity marking algorithm is not known, attackers might not bother to connect their Sybil identities to the SCC or to each other. Interestingly, we found 5 suspicious articles with sophisticated voting patterns where one voter is linked to many identities (~ 30) that also vote on the same article. We believe the many identities behind that single voter are likely Sybil identities because those identities were all created on the same day as the article's submission. Additionally, those identities all have similar usernames.

8 SumUp in a Decentralized Setting

Even though SumUp is presented in a centralized setup such as a content-hosting Web site, it can also be implemented in a distributed fashion in order to rank objects in peer-to-peer systems. We outline one such distributed design for SumUp. In the peer-to-peer environment, each node and its corresponding user is identified by a self-generated public key. A pair of users create a trust link relationship between them by signing the trust statement with their private keys. Nodes gossip with each other or perform a crawl of the network to obtain a complete trust network between any pair of public keys. This is different from Ostra [18] and SybilLimit [26] which address the harder problem of decentralized routing where each user only knows about a small neighborhood around himself in the trust graph. In the peer-to-peer setup, each user naturally acts as his own vote collector to aggregate votes and compute a personalized ranking of objects. To obtain all votes on an object, a node can either perform flooding (like in Credence [25]) or retrieve votes stored in a distributed hash table. In the latter case, it is important that the DHT itself be resilient against Sybil attacks. Recent work on Sybil-resilient DHTs [5, 14] addresses this challenge.

9 Conclusion

This paper presented SumUp, a content voting system that leverages the trust network among users to defend against Sybil attacks. By using the technique of adaptive vote flow aggregation, SumUp aggregates a collection of votes with strong security guarantees: with high probability, the number of bogus votes collected is bounded by the number of attack edges while the number of honest votes collected is high. We demonstrate the real-world benefits of SumUp by evaluating it on the voting trace of Digg: SumUp detected many suspicious articles marked as "popular" by Digg. We have found strong evidence of Sybil attacks on many of these suspicious articles.

Acknowledgments

We thank Alan Mislove and Krishna Gummadi for making their social network traces publicly available. We are grateful to Krishna Gummadi (our shepard), Haifeng Yu, Aditya Dhananjay, Michael Paik, Eric Hielscher, and the anonymous reviewers whose comments have helped us improve the paper. This work is supported by NSF CAREER Award CNS-0747052.

References

[1] Digg. http://www.digg.com.

[2] BRIN, S., AND PAGE, L. The anatomy of a large-scale hypertextual web search engine. In *WWW* (1998).

[3] CHENG, A., AND FRIEDMAN, E. Sybilproof reputation mechanisms. In *P2PECON '05: Proceedings of the 2005 ACM SIGCOMM workshop on Economics of peer-to-peer systems* (New York, NY, USA, 2005), ACM, pp. 128–132.

[4] CORMEN, T., LEISERSON, C., AND RIVEST, R. *Introduction to Algorithms*. The MIT Press, 1985.

[5] DANEZIS, G., LESNIEWSKI-LAAS, C., KAASHOEK, M. F., AND ANDERSON, R. Sybil-resistant dht routing. In *European Symposium On Research In Computer Security* (2008).

[6] DOUCEUR, J. The sybil attack. In *1st International Workshop on Peer-to-Peer Systems* (2002).

[7] FENG, Q., AND DAI, Y. Lip: A lifetime and popularity based ranking approach to filter out fake files in p2p file sharing systems. In *IPTPS* (2007).

[8] GUHA, R., KUMAR, R., RAGHAVAN, P., AND TOMKINS, A. Propagation of trust and distrust. In *WWW* (2004).

[9] HSIEH, H. Doonesbury online poll hacked in favor of MIT. In *MIT Tech* (2006).

[10] INVESPBLOG. An interview with digg top user, 2008. `http://www.invesp.com/blog/social-media/an-interview-with-digg-top-user.html`.

[11] KAMVAR, S. D., SCHLOSSER, M. T., AND GARCIA-MOLINA, H. The eigentrust algorithm for reputation management in p2p networks. In *WWW '03: Proceedings of the 12th international conference on World Wide Web* (New York, NY, USA, 2003), ACM, pp. 640–651.

[12] KLEINBERG, J. Authoritative sources in a hyperlinked environment. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms* (1998).

[13] LESKOVEC, J., LANG, K., DASGUPTA, A., AND MAHONEY, M. W. Statistical properties of community structure in large social and information networks. In *7th international conference on WWW* (2008).

[14] LESNIEWSKI-LAAS, C. A sybil-proof one-hop dht. In *1st Workshop on Social Network Systems* (2008).

[15] LEVIEN, R., AND AIKEN, A. Attack-resistant trust metrics for public key certification. In *SSYM'98: Proceedings of the 7th conference on USENIX Security Symposium, 1998* (Berkeley, CA, USA, 1998), USENIX Association, pp. 18–18.

[16] LIANG, J., KUMAR, R., XI, Y., AND ROSS, K. Pollution in p2p file sharing systems. In *IEEE Infocom* (2005).

[17] MISLOVE, A., MARCON, M., GUMMADI, K., DRUSCHEL, P., AND BHATTACHARJEE, S. Measurement and analysis of online social networks. In *7th Usenix/ACM SIGCOMM Internet Measurement Conference (IMC)* (2007).

[18] MISLOVE, A., POST, A., DRUSCHEL, P., AND GUMMADI, K. P. Ostra: Leveraging trust to thwart unwanted communication. In *NSDI'08: Proceedings of the 5th conference on 5th Symposium on Networked Systems Design & Implementation* (2008).

[19] MOTWANI, R., AND RAGHAVAN, P. *Randomized Algorithms*. Cambridge University Press, 1995.

[20] NG, A., ZHENG, A., AND JORDAN, M. Link analysis, eigenvectors and stability. In *International Joint Conference on Artificial Intelligence (IJCAI)* (2001).

[21] RICHARDSON, M., AGRAWAL, R., AND DOMINGOS, P. Trust management for the semantic web. In *Proceedings of the 2nd Semantic Web Conference* (2003).

[22] RILEY, D. Techcrunch: Stat gaming services come to youtube, 2007. `http://www.techcrunch.com/2007/08/23/myspace-style-profile-gaming-comes-to-youtub`

[23] SOVRAN, Y., LIBONATI, A., AND LI, J. Pass it on: Social networks stymie censors. In *Proc. of the 7th International Workshop on Peer-to-Peer Systems (IPTPS)* (Feb 2008).

[24] TOIVONEN, R., ONNELA, J.-P., SARAMÄKI, J., HYVÖNEN, J., AND KASKI, K. A model for social networks. *Physica A Statistical Mechanics and its Applications 371* (2006), 851–860.

[25] WALSH, K., AND SIRER, E. G. Experience with an object reputation system for peer-to-peer filesharing. In *NSDI'06: Proceedings of the 3rd conference on 3rd Symposium on Networked Systems Design & Implementation* (Berkeley, CA, USA, 2006), USENIX Association, pp. 1–1.

[26] YU, H., GIBBONS, P., KAMINSKY, M., AND XIAO, F. Sybillimit: A near-optimal social network defense against sybil attacks. In *IEEE Symposium on Security and Privacy* (2008).

[27] YU, H., KAMINSKY, M., GIBBONS, P. B., AND FLAXMAN, A. Sybilguard: defending against sybil attacks via social networks. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications* (New York, NY, USA, 2006), ACM, pp. 267–278.

[28] YU, H., SHI, C., KAMINSKY, M., GIBBONS, P. B., AND XIAO, F. Dsybil: Optimal sybil-resistance for recommendation systems. In *IEEE Security and Privacy* (2009).

[29] ZHANG, H., GOEL, A., GOVINDAN, R., AND MASON, K. Making eigenvector-based reputation systems robust to collusions. In *Proc. of the Third Workshop on Algorithms and Models for the Web Graph* (2004).

[30] ZIEGLER, C.-N., AND LAUSEN, G. Propagation models for trust and distrust in social networks. *Information Systems Frontiers 7*, 4-5 (2005), 337–358.

Bunker: A Privacy-Oriented Platform for Network Tracing

Andrew G. Miklas†, Stefan Saroiu‡, Alec Wolman‡, and Angela Demke Brown†
†University of Toronto and ‡Microsoft Research

Abstract: *ISPs are increasingly reluctant to collect and store raw network traces because they can be used to compromise their customers' privacy. Anonymization techniques mitigate this concern by protecting sensitive information. Trace anonymization can be performed* offline *(at a later time) or* online *(at collection time). Offline anonymization suffers from privacy problems because raw traces must be stored on disk – until the traces are deleted, there is the potential for accidental leaks or exposure by subpoenas. Online anonymization drastically reduces privacy risks but complicates software engineering efforts because trace processing and anonymization must be performed at line speed. This paper presents Bunker, a network tracing system that combines the software development benefits of offline anonymization with the privacy benefits of online anonymization. Bunker uses virtualization, encryption, and restricted I/O interfaces to protect the raw network traces and the tracing software, exporting only an anonymized trace. We present the design and implementation of Bunker, evaluate its security properties, and show its ease of use for developing a complex network tracing application.*

1 Introduction

Network tracing is an indispensable tool for many network management tasks. Operators need network traces to perform routine network management operations, such as traffic engineering [19], capacity planning [38], and customer accounting [15]. Several research projects have proposed using traces for even more sophisticated network management tasks, such as diagnosing faults and anomalies [27], recovering from security attacks [45], or identifying unwanted traffic [9]. Tracing is also vital to networking researchers. As networks and applications grow increasingly complex, understanding the behavior of such systems is harder than ever. Gathering network traces helps researchers guide the design of future networks and applications [42, 49].

Customer privacy is a paramount concern for all online businesses, including ISPs, search engines, and e-commerce sites. Many ISPs view possessing raw network traces as a liability: such traces sometimes end up compromising their customers' privacy through leaks or subpoenas. These concerns are real: the RIAA has subpoenaed ISPs to reveal customer identities when pursuing cases of copyright infringement [16]. Privacy concerns go beyond subpoenas, however. Oversights or errors in preparing and managing network trace and server log files can seriously compromise users' privacy by disclosing social security numbers, names, addresses, or telephone numbers [5, 54].

Trace anonymization is the most common technique for addressing these privacy concerns. A typical implementation uses a keyed one-way secure hash function to obfuscate sensitive information contained in the trace. This could be as simple as transforming a few fields in the IP headers, or as complex as performing TCP connection reconstruction and then obfuscating data (e.g., email addresses) deep within the payload. There are two current approaches to anonymizing network traces: *offline* and *online*. Offline anonymization collects and stores the entire raw trace and then performs anonymization as a post-processing step. Online anoymization is done on-the-fly by extracting and anonymizing sensitive information before it ever reaches the disk. In practice, both methods have serious shortcomings that make network trace collection increasingly difficult for network operators and researchers.

Offline anonymization poses risks to customer privacy because of how raw network traces are stored. These risks are growing more severe because of the need to look "deeper" into packet payloads, revealing more sensitive information. Current privacy trends make it unlikely that ISPs will continue to accept the risks associated with offline anonymization. We have first-hand experience with tracing Web, P2P, and e-mail traffic at two universities. In both cases the universities deemed the privacy risks associated with offline anonymization to be unacceptable.

While online anonymization offers much stronger privacy benefits, it is very difficult to deploy in practice because it creates significant software engineering issues. Any portion of the trace analysis that requires access to sensitive data must be performed on-the-fly and at a rate that can handle the network's peak throughput. This is practical for simple tracing applications that analyze only IP and TCP headers; however, it is much more difficult for tracing applications that require deep packet inspection. Developing complex online tracing software therefore poses a significant challenge. Developers are limited in their selection of software: adopting garbage-collected (e.g., Java, C#) and dynamic scripting (e.g., Python, Perl) languages can be difficult; reusing existing libraries (e.g., HTML parsers or regexp engines) may also be hard if their implementation choices are incompatible with performance requirements. A network tracing experiment illustrates the performance challenges of online tracing.

Our goal was to run hundreds of regular expressions to identify phishing Web forms. However, an Intel 3.6GHz processor running just one of these regular expressions (using the off-the-shelf "libpcre" regexp library) could only handle less than 50 Mbps of incoming traffic.

This paper presents Bunker, a network tracing system built and deployed at the University of Toronto. Bunker offers the software development benefits of offline anonymization and the privacy benefits of online anonymization. Our key insight is that we can use the buffer-on-disk approach of offline anonymization if we can "lock down" the trace files and trace analysis software. This approach lets Bunker avoid all the software engineering downsides of online trace analysis. To implement Bunker, we use virtual machines, encryption, and restriction of I/O device configuration to construct a *closed-box* environment; Bunker requires no specialized hardware (e.g., a Trusted Platform Module (TPM) or a secure co-processor) to provide its security guarantees. The trace analysis and anonymization software is preloaded into a closed-box VM before any raw trace data is gathered. Bunker makes it difficult for network operators to interact with the tracing system or to access its internal state once it starts running and thereby protects the anonymization key, the tracing software, and the raw network trace files inside the closed-box environment. The closed-box environment produces an anonymized trace as its only output.

To protect against physical attacks (e.g., hardware tampering), we design Bunker to be *safe-on-reboot*: upon a reboot, all sensitive data gathered by the system is effectively destroyed. This property makes physical attacks more difficult because the attacker must tamper with Bunker's hardware without causing a reboot. While a small class of physical attacks remains feasible (e.g., cold boot attacks [21]), in our experience ISPs find the privacy benefits offered by a closed-box environment that is safe-on-reboot a significant step forward. Although the system cannot stop ISPs from being subject to wiretaps, Bunker helps protect ISPs against the privacy risks inherent in collecting and storing network traces.

Bunker's privacy properties come at a cost. Bunker requires the network operator to pre-plan what data to collect and how to anonymize it before starting to trace the network. Bunker prevents anyone from changing the configuration while tracing; it can be reconfigured only through a reboot that will erase all sensitive data.

The remainder of this paper describes Bunker's threat model (Section 2), design goals and architecture (Section 3), as well as the benefits of Bunker's architecture (Section 4). It then analyzes Bunker's security properties when confronted with a variety of attacks (Section 5), describes operational issues (Section 6), and evaluates Bunker's software engineering benefits by examining a tracing application (phishing analysis) built by one student in two months that leverages off-the-shelf components and scripting languages (Section 7). The paper's final sections review legal issues posed by Bunker's architecture (Section 8) and related work (Section 9).

2 Threat Model

This section outlines the threat model for network tracing systems. We present five classes of attacks and discuss how Bunker addresses each.

2.1 Subpoenas For Network Traces

ISPs are discovering that traces gathered for diagnostic and research purposes can be used in court proceedings against their customers. As a result, they may view the benefits of collecting network traces as being outweighed by the liability of possessing such information. Once a subpoena has been issued, an ISP must cooperate and reveal the requested information (e.g., traces or encryption keys) as long as the cooperation does not pose an undue burden. Consequently, *a raw trace is protected against a subpoena only if no one has access to it or to the encryption and anonymization keys used to protect it.*

Our architecture was designed to collect traces while preserving user privacy even if a court permits a third party to have full access to the system. Once a Bunker trace has been initiated, all sensitive information is protected from the system administrator in the same way it is protected from any adversary. Thus, our solution makes it a hardship for the ISP to surrender sensitive information. We eliminate potential downsides to collecting traces for legitimate purposes but do not prevent those with legal wiretap authorization from installing their own trace collection system.

2.2 Accidental Disclosure

ISPs face another risk, that of accidental disclosure of sensitive information from a network trace. History has shown that whenever people handle sensitive data, the danger of accidental disclosure is substantial. For example, the British Prime Minister recently had to publicly apologize when a government agency accidentally lost 25 million child benefit records containing names and bank details because the agency did not follow the correct procedure for sending these records by courier [5]. Bunker vastly reduces the risk that sensitive data will be accidentally released or stolen because no human can access the unanonymized trace.

2.3 Remote Attacks Over The Internet

Remote theft of data collected by a tracing machine presents another threat to network tracing systems. There are many possible ways to break into a system over the network, yet there is one simple solution that eliminates this entire class of attacks. To collect traces, Bunker uses a specialized network capture card that is incapable of sending outgoing data. It also uses firewall rules to limit access to the tracing machine from the internal private network. Section 5.3 examines in-depth Bunker's security measures against such attacks.

2.4 Operational Attacks

Attacks that traverse the network link being monitored, such as denial-of-service (DoS) attacks, may also incidentally affect the tracing system. This is a problem when tracing networks with direct connections to the Internet: Internet hosts routinely receive attack traffic such as vulnerability probes, denial-of-service (DoS) attacks, and back-scatter from attacks occurring elsewhere on the Internet [36]. Methods exist to reduce the impact of DoS attacks [31] and adversarial traffic [13]. However, these methods may have limited effectiveness against a large enough attack. Both Bunker and offline anonymization systems are more resilient to such attacks because they need not process the traffic in real time.

Because many network studies collect traces for relatively long time periods, an attacker with physical access could tamper with the monitoring system after it has started tracing, creating the appearance that the original system is still running. For example, the attacker might reboot the system and then set up a new closed-box environment that uses anonymization keys known to the attacker. Section 6 describes a simple modification to Bunker that addresses this type of attack.

2.5 Attacks On Anonymization

Packet injection attacks attempt to partially learn the anonymization mapping by injecting traffic and then analyzing the anonymized trace. To perform such attacks, an adversary transmits traffic over the network being traced and later identifies this traffic in the anonymized trace. These attacks are possible when non-sensitive trace information (e.g., times or request sizes) is used to correlate entries in the anonymized trace with the specific traffic being generated by the adversary. Packet injection attacks do not completely break the anonymization mapping because they do not let the adversary deduce the anonymization key. Even without packet injection, recent work has shown that private information can still be recovered from data anonymized with state-of-the-art techniques [10, 34]. These attacks typically make use of public information and attempt to correlate it with the obfuscated data. Our tracing system is susceptible to attacks on the anonymization scheme. The best way to defend against this class of attacks is to avoid public release of anonymized trace data [10].

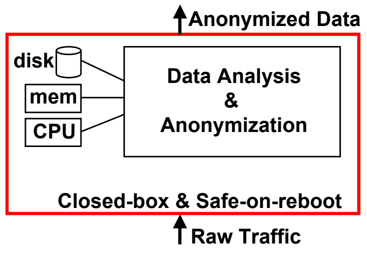


Figure 1. ***Logical view of Bunker:*** *Raw data enters the closed-box perimeter and only anonymized data leaves this perimeter.*

Another problem involves ensuring that the anonymization policy is specified correctly, and that the implementation correctly implements the specification. Bunker does not explicitly address these issues. We recommend code reviews of the trace analysis and anonymization software. However, even a manual audit of this software can miss certain properties and anomalies that could be exploited by a determined adversary [34]. Although there is no simple checklist to follow that ensures a trace does not leak private data, there are tools that can aid in the design and implementation of sound anonymization policies [35].

2.6 Summary

Bunker's design raises the bar for mounting any of these attacks successfully. At a high level, our threat model assumes that: (1) the attacker has physical access to the tracing infrastructure but no specialized hardware, such as a bus monitoring tool; (2) the attacker did not participate in implementing the trace analysis software. While Bunker's security design is motivated by the threat of subpoenas, it also addresses the other four classes of attacks described in this section. We examine security attacks against Bunker in Section 5 and we discuss legal issues in Section 8.

3 The Bunker Architecture

Our main insight when designing Bunker is that a tracing infrastructure can maintain large caches of sensitive data without compromising user privacy as long as none of that data leaves the host. Figure 1 illustrates Bunker's high-level design, which takes raw traffic as input and generates an anonymized trace.

3.1 Design Goals

1. Privacy. While the system may store sensitive data such as unanonymized packets, it must not permit an outside agent to extract anything other than analysis output.

2. Ease of development. The system should place as few constraints as possible on implementing the analysis software. For example, protocol reconstruction and parsing should not have real-time performance requirements.

3. Robustness. Common bugs found in handling corner cases in parsing and analysis code should lead to small errors in the trace rather than crashing the system or completely corrupting its output.

4. Performance. The proposed system must perform as well as today's network tracers when running on equivalent hardware. In particular, it should be possible to trace a high-capacity link with inexpensive hardware.

5. Use commodity hardware and software. The proposed system should not require specialized hardware, such as a Trusted Platform Module (TPM).

3.2 Privacy Properties

To meet our privacy design goal, we must protect all gathered trace data even from an attacker who has physical access to the network tracing platform. To achieve this high-level of protection, we designed Bunker to have the following two properties:

1. Closed-box. The tracing infrastructure runs all software that has direct access to the captured trace data inside a closed-box environment. Administrators, operators, and users cannot interact with the tracing system or access its internal state once it starts running. Input to the closed-box environment is raw traffic; output is an anonymized trace.

2. Safe-on-reboot. Upon a reboot, all gathered sensitive data is effectively destroyed. This means that all unencrypted data is actually destroyed; the encryption key is destroyed for all encrypted data placed in stable storage. Bunker uses ECC RAM modules that are zeroed out by the BIOS before booting [21]. Thus, it is safe-on-reboot for reboots caused by pressing the RESET button or by powering off the machine.

The closed-box property prevents an attacker from gaining access to the data or to the tracing code while it is running. However, this property is not sufficient. An attacker could restart the system and boot a different software image to access data stored on the tracing system, or an attacker could tamper with the tracing hardware (e.g., remove a hard drive and plug it in to another system). To protect sensitive data against such physical attacks, we use the safe-on-reboot property to erase all sensitive data upon a reboot. Together, these two properties prevent an attacker from gaining access to sensitive data via system tampering.

3.3 The Closed-Box Property

Bunker uses virtual machines to provide the closed-box property. We now describe the rationale for our design and implementation.

3.3.1 Design Approach

In debating whether to use virtual or physical machines (e.g., a sealed appliance) to design our closed-box environment, we chose the virtual machine option primarily for flexibility and ease of development. We anticipated that our design would undergo small modifications to accommodate unforeseen problems and worried that making small changes to a sealed appliance would be too difficult after the initial system was implemented and deployed. With VMs, Bunker's software can be easily retrofitted to trace different types of traffic. For example, we used Bunker to gather a trace of Hotmail e-mails and to gather flow-level statistics about TCP traffic.

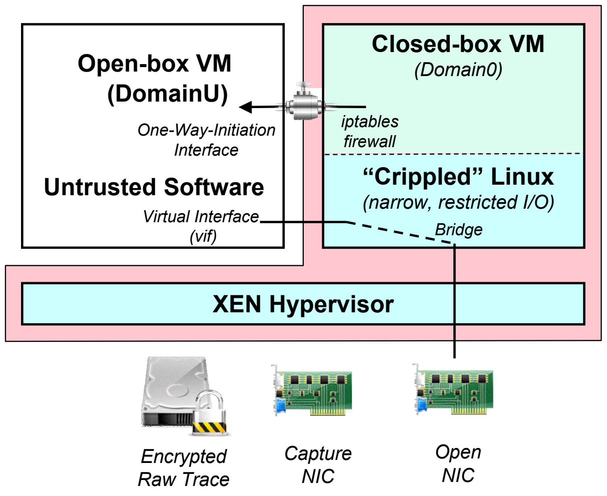


Figure 2. ***Overview of Bunker's implementation.*** *The closed-box VM runs a carefully configured Linux kernel. The shaded area represents the Trusted Computing Base (TCB) of our system.*

Virtual machine monitors (VMMs) have been used in the past for building closed-box VMs [20, 11]. Using virtual machines to provide isolation is especially beneficial for tasks that require little interaction [6], such as network tracing. Bunker runs all software that processes captured data inside a highly trusted closed-box VM. Users, administrators, and software in other VMs cannot interact with the closed-box or access any of its internal state once it starts running.

3.3.2 Implementation Details

We used the Xen 3.1 VMM to implement Bunker's closed-box environment. Xen, an open-source VMM, provides para-virtualized x86 virtual machines [4]. The VMM executes at the highest privilege level on the processor. Above the VMM are the virtual machines, which Xen calls *domains*. Each domain executes a guest operating system, such as Linux, which runs at a lower privilege level than the VMM.

In Xen, Domain0 has a special role: it uses a control interface provided by the VMM to perform management functions outside of the VMM, such as creating other domains and providing access to physical devices (including the network interfaces). Both its online trace

```
iptables -P INPUT DROP
iptables -A INPUT -m state --state ESTABLISHED -j ACCEPT
iptables -A OUTPUT -m state --state NEW,ESTABLISHED -j ACCEPT
```

Figure 3. ***iptables firewall rules:*** *An abbreviated list of the rules that creates a one-way-initiation interface between the closed-box VM and the open-box VM. These rules allow connections only if they are initiated by the closed-box VM. Note that the ESTABLISHED state above refers to a connection state used by iptables and not to the ESTABLISHED state in the TCP stack.*

collection and offline trace analysis components are implemented as a collection of processes that execute on a "crippled" Linux kernel that runs in the Domain0 VM, as shown in Figure 2.

We carefully configured the Linux kernel running in Domain0 to run as a closed-box VM. To do this, we severely limited the closed-box VM's I/O capabilities and disabled all the kernel functionality (i.e., kernel subsystems and modules) not needed to support tracing. We disabled all drivers (including the monitor, mouse and keyboard) inside the kernel except for: 1) the network capture card driver; 2) the hard disk driver; 3) the virtual interface driver, used for closed-box VM to open-box VM communication, and 4) the standard NIC driver used to enable networking in the open-box VM. We also disabled the login functionality; nobody, ourselves included, can login to the closed-box VM. Once the kernel boots, the kernel init process runs a script that launches the tracer. We provide a publicly downloadable copy of the kernel configuration file[1] used to compile the Domain0 kernel so that anyone can audit it.

The closed-box VM sends anonymized data and non-sensitive diagnostic data to the open-box VM via a *one-way-initiation interface*, as follows. We setup a layer-3 firewall (e.g., iptables) that allows only those connections initiated by the closed-box VM; this firewall drops any unsolicited traffic from the open-box VM. Figure 3 presents an abbreviated list of the firewall rules used to configure this interface.

We deliberately *crippled* the kernel to restrict all other I/O except that from the four remaining drivers. We configured and examined each driver to eliminate any possibility of an adversary taking advantage of these channels to attack Bunker. Section 5 describes Bunker's system security in greater detail.

3.4 The Safe-on-Reboot Property

To implement the safe-on-reboot property, we need to ensure that all sensitive data and the anonymization key are stored in volatile memory only. However, tracing experiments frequently generate more sensitive data than can fit into memory. For example, a researcher might need to capture a very large raw packet trace before running a trace analysis program that makes multiple passes through the trace. VMMs alone cannot protect data written to disk, because an adversary could simply move the drive to another system to extract the data.

3.4.1 Design Approach

On boot-up, the closed-box VM selects a random key that will be used to encrypt any data written to the hard disk. This key (along with the anonymization key) is stored only in the closed box VM's volatile memory, ensuring that it is both inaccessible to other VMs and lost on reboot. Because data stored on the disk can be read only with the encryption key, this approach effectively destroys the data after a reboot. The use of encryption to make disk storage effectively volatile is not novel; swap file encryption is used on some systems to ensure that fragments of an application's memory space do not persist once the application has terminated or the system has restarted [39].

3.4.2 Implementation Details

To implement the safe-on-reboot property, we need to ensure that all sensitive information is either stored only in volatile memory or on disk using encryption where the encryption key is stored only in volatile memory. To implement the encrypted store, we use the *dm-crypt* [41] device-mapper module from the Linux 2.6.18 kernel. This module provides a simple abstraction: it adds an encrypted device on top of any ordinary block device. As a result, it works with any file system. The dm-crypt module supports several encryption schemes; we used the optimized implementation of AES. To ensure that data in RAM does not accidentally end up on disk, we disabled the swap partition. If swapping is needed in the future, we could enable dm-crypt on the swap partition. The root file system partition that contains the closed-box operating system is initially mounted read only. Because most Linux configurations expect the root partition to be writable, we enable a read-write overlay for the root partition that is protected by dm-crypt. This also ensures that the trace analysis software does not accidentally write any sensitive data to disk without encryption.

3.5 Trace Analysis Architecture

Bunker's tracing software consists of two major pieces: 1) the online component, independent of the particular network tracing experiment, and 2) the offline component, which in our case is a phishing analysis tracing application. Figure 4 shows Bunker's entire pipeline, including the online and offline components.

Bunker uses *tcpdump* version 3.9.5 to collect packet traces. We fine-tuned tcpdump to increase the size of its

[1]`http://www.slup.cs.toronto.edu/utmtrace/config-2.6.18-xen0-noscreen`

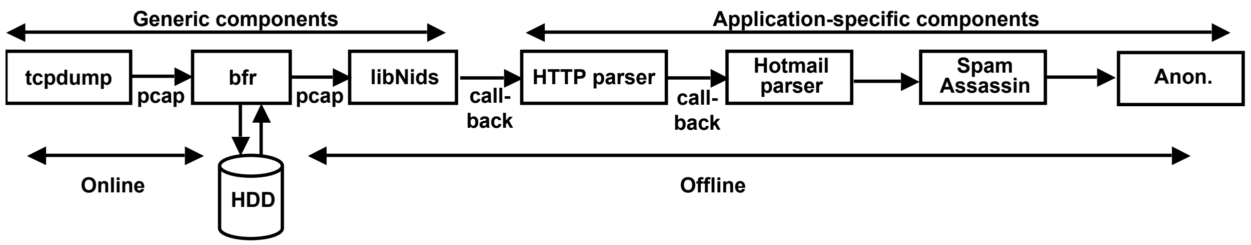


Figure 4. ***Flow of trace data through Bunker's modules.*** *The online part of Bunker consists of tcpdump and the bfr buffering module. The offline part of Bunker consists of bfr, libNids, HTTP parser, Hotmail parser, SpamAssassin, and an anonymizer module. Also, tcpdump, bfr, and libNids are generic components to Bunker, wherease HTTP parser, Hotmail parser, SpamAssassin, and the anonymized module are specific to our current application: collecting traces of phishing e-mail.*

receive buffers. All output from tcpdump is sent directly to *bfr*, a Linux non-blocking pipe buffer that buffers data between Bunker's offline and online components. We use multiple memory mapped files residing on the encrypted disks as the bfr buffer and we allocate 380 GB of disk space to it, sufficient to buffer over 8 hours of HTTP traffic for our network. Figure 5 shows how bfr's buffer size varies over time.

Our Bunker deployment at the University of Toronto is able to trace continuously, even with an unoptimized offline component. This is because of the cyclical nature of network traffic (e.g., previous studies showed that university traffic is 1.5 to 2 times lower on a weekend day than on a week day [42, 50]). This allows the offline component to catch up with the online component during periods of low load, such as nights and weekends. In general, Bunker can only trace continuously if the buffer drains completely at least once during the week. If the peak buffer size during a week day is p and Bunker's offline component leaves Δ unprocessed at the end of a week day (see Figure 5), Bunker is able to trace continuously if the following two conditions hold:

1. Bunker's buffer size is larger than $4 \times \Delta + p$, or the amount of unprocessed data after four consecutive week days plus the peak traffic on the fifth week day;

2. During the weekend, Bunker's offline component can catch up to the online component by at least $5 \times \Delta$ of the unprocessed data in the buffer.

The tracing application we built using Bunker gathers traces of phishing e-mails received by Hotmail users at the University of Toronto. The offline trace analysis component performs five tasks: 1) reassembling packets into TCP streams; 2) parsing HTTP; 3) parsing Hotmail; 4) running SpamAssassin over the Hotmail e-mails, and 5) anonymizing output. To implement each of these tasks, we wrote simple Python and Perl scripts that made extensive use of existing libraries and tools.

For TCP/IP reconstruction, we used *libNids* [48], a C library that runs the TCP/IP stack from the Linux 2.0 kernel in user-space. libNids supports reassembly of both IP fragments and TCP streams. Both the HTTP and the Hotmail parsers are written in Python version 2.5. We used a wrapper for libNids in Python to interface with our HTTP parsing code. Whenever a TCP stream is assembled, libNids calls a Python function that passes on the content to the HTTP and Hotmail parsers. The Hotmail parser passes the bodies of the e-mail messages to SpamAssassin (written in Perl) to utilize its spam and phishing detection algorithms. The output of SpamAssassin is parsed and then added to an internal object that represents the Hotmail message. This object is then serialized as a Python "pickled" object before it is transferred to the anonymization engine. We used an HTTP anonymization policy similar to the one described in [35]. We took two additional steps towards ensuring that the anonymization policy is correctly specified and implemented: (1) we performed a code review of the policy and its implementation, and (2) we made the policy and the code available to the University of Toronto's network operators encouraging them to inspect it.

3.6 Debugging

Debugging a closed-box environment is challenging because an attacker could use the debugging interface to extract sensitive internal state from the system. Despite this restriction, we found the development of Bunker's analysis software to be relatively easy. Our experience found the off-the-shelf analysis code we used in Bunker to be well tested and debugged. We used two additional techniques for helping to debug Bunker's analysis code. First, we tested our software extensively in the lab against synthetic traffic sources that do not pose any privacy risks. To do this, we booted Bunker into a special diagnostic mode that left I/O devices (such as the keyboard and monitor) enabled. This configuration allowed us to easily debug the system and patch the analysis software without rebooting.

Second, we ensured that every component of our analysis software produced diagnostic logs. These logs were sent from the closed-box VM to the open-box VM using

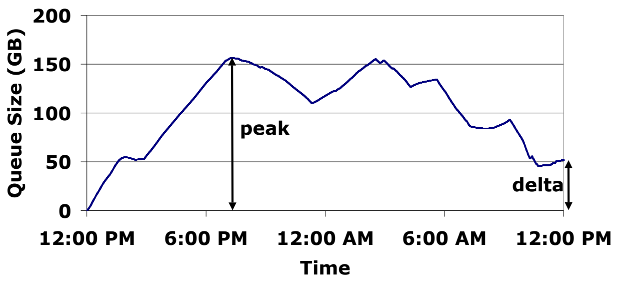


Figure 5. ***The size of bfr's buffer over time.*** *While the queue size increases during the day, it decreases during night when there is less traffic. At the end of this particular day, Bunker's offline component still had 50GB of unprocessed raw trace left in the buffer.*

the same interface as the anonymized trace. They proved helpful in shedding light on the "health" of the processes inside the closed-box VM. We were careful to ensure that no sensitive data could be written to the log files in order to preserve trace data privacy.

4 The Benefits of Bunker

This section presents the benefits offered by Bunker's architecture.

4.1 Privacy Benefits

Unlike offline anonymization, our approach does not allow network administrators or researchers to work directly with sensitive data at any time. Because unanonymized trace data cannot be directly accessed, it cannot be produced under a subpoena. Our approach also greatly reduces the chance that unanonymized data will be stolen or accidentally released because individuals cannot easily extract such data from the system.

The privacy guarantees provided by our tracing system are more powerful than those offered by online anonymization. Bunker's anonymization key is stored within the closed-box VM, which prevents anyone from accessing it. While online anonymization tracing systems are typically careful to avoid writing unanonymized data to stable storage, they generally do not protect the anonymization key against theft by an adversary with the ability to login to the machine.

4.2 Software Engineering Benefits

When an encrypted disk is used to store the raw network trace for later processing, the trace analysis code is free to run offline at slower than line speeds. Bunker supports two models for tracing. In *continuous tracing*, the disk acts as a large buffer, smoothing the traffic's bursts and its daily cycles. To trace network traffic continuously, Bunker's offline analysis code needs to run fast enough for the *average* traffic rate, but it need not keep up with the *peak* traffic rate. Bunker also supports deferred trace analysis, where the length of the tracing period is limited by the amount of disk storage, but there are no constraints on the performance of the offline trace analysis code. In contrast, online anonymization tracing systems process data as it arrives and therefore must handle peak traffic in real-time.

Bunker's flexible performance requirements let the developer use managed languages and sophisticated libraries when creating trace analysis software. As a result, its code is both easier to write and less likely to contain bugs. The phishing analysis application using Bunker was built by one graduate student in less than two months, including the time spent configuring the closed-box environment (a one-time cost with Bunker). This development effort contrasts sharply with our experience developing tracing systems with online anonymization. To improve performance, these systems required developers to write carefully optimized code in low-level languages using sophisticated data structures. Bunker lets us use Python scripts to parse HTTP, a TCP/IP reassembly library, and Perl scripts running SpamAssassin.

4.3 Fault Handling Benefits

One serious drawback of most online trace analysis techniques is their inability to cope gracefully with bugs in the analysis software. Often, these are "corner-case" bugs that arise in abnormal traffic patterns. In many cases researchers and network operators would prefer to ignore these abnormal flows and continue the data gathering process; however, if the tracing software crashes, all data would be lost until the system can be restarted. This could result in the loss of megabytes of data even if the restart process is entirely automated. Worse, this process introduces systematic bias in the data collection because crashes are more likely to affect long-lived than short-lived flows.

Bunker can better cope with bugs because its online and offline components are fully decoupled. This provides a number of benefits. First, Bunker's online trace collection software is simple because it only captures packets and loads them in RAM (encryption is handled automatically at the file system layer). Its simplicity and size make it easy to test extensively. Second, the online software need not change even when the type of trace analysis being performed changes. Third, the offline trace analysis software also becomes much simpler because it need not be heavily optimized to run at line speed. Unoptimized software tends to have a simpler program structure and therefore fewer bugs. Simpler program structure also makes it easier to recover from bugs when they do arise. Finally, a decoupled architecture makes it possible to identify the flow that caused the error in the trace analyzer, filter out that flow from the buffered raw trace, and restart the trace analyzer so that it never sees that flow as input and thereby avoids the bug entirely. Section 7 quantifies the effect of this improved fault handling on the number of flows that are dropped due to a parsing bug.

5 Security Attacks

Bunker's design is inspired by Terra, a VM-based platform for trusted computing [20]. Both Terra and Bunker protect sensitive data by encapsulating it in a closed-box VM with deliberately restricted I/O interfaces. The security of such architectures does not rest on the size of the trusted computing base (TCB) but on whether an attacker can exploit a vulnerability through the system's narrow interfaces. Even if there is a vulnerability in the OS running in the closed-box VM, Bunker remains secure as long as attackers cannot exploit the vulnerability through the restricted channels. In our experience, ISPs have found Bunker's security properties a significant step forward in protecting users privacy when tracing.

Attacks on Bunker can be categorized into three classes. The first are those that attempt to subvert the narrow interfaces of the closed-box VM. A successful attack on these interfaces exposes the closed-box VM's internals. The second class are physical attacks, in which the attacker tampers with Bunker's hardware. The third possibility are attacks whereby Bunker deliberately allows network traffic into the closed-box VM: an attacker could try to exploit a vulnerability in the trace analysis software by injecting traffic in the network being monitored. We now examine each attack type in greater detail.

5.1 Attacking the Restricted Interfaces of the Closed-Box VM

There are three ways to attack the restricted interfaces of the closed-box VM: 1) subverting the isolation provided by the VMM to access the memory contents of the closed-box VM; 2) exploiting a security vulnerability in one of the system's drivers; and 3) attacking the closed-box VM directly using the one-way-initiation interface between the closed and open-box VMs.

5.1.1 Attacking the VMM

We use a VMM to enforce isolation between software components that need access to sensitive data and those that do not. Bunker's security rests on the assumption that VMM-based isolation is hard to attack, an assumption made by many in industry [23, 47] and the research community [20, 11, 6, 43]. There are other approaches we could have used to confine sensitive data strictly to the pre-loaded analysis software. For example, we could have used separate physical machines to host the closed and open box systems. Alternatively, we could have relied on a kernel and its associated isolation mechanisms, such as processes and file access controls. However, VM-based isolation is generally thought to provide stronger security than process-based isolation because VMMs are small enough to be rigorously verified and export only a very narrow interface to their VMs [6, 7, 29]. In contrast, kernels are complex pieces of software that expose a rich interface to their processes.

5.1.2 Attacking the Drivers

Drivers are among the buggiest components of an OS [8]. Security vulnerabilities in drivers let attackers bypass all access restrictions imposed by the OS. Systems without an IOMMU are especially susceptible to buggy drivers because they cannot prevent DMA-capable hardware from accessing arbitrary memory addresses. Many filesystem drivers can be exploited by carefully crafted filesystems [53]. Thus, if Bunker were to automount inserted media, an attacker could compromise the system by inserting a CDROM or USB memory device with a carefully crafted filesystem image.

Bunker addresses such threats by disabling all drivers (including the monitor, mouse, and keyboard) except these four: 1) the network capture card driver, 2) the hard disk driver, 3) the driver for the standard NIC used to enable networking in the open-box VM, and 4) the driver for the virtual interfaces used between the closed-box and open-box VMs. In particular, we were careful to disable external storage device support (i.e. CDROM, USB mass storage) and USB support.

We examined each of these drivers and believe that none can be exploited to gain access to the closed-box. First, the network capture card loads incoming network traffic via one of the drivers left enabled in Domain0. This capture card, a special network monitoring card made by Endace (DAG 4.3GE) [17], cannot be used for two-way communication. Thus, an attacker cannot gain remote access to the closed-box solely through this network interface. The second open communication channel is the SCSI controller driver for our hard disks. This is a generic Linux driver, and we checked the Linux kernel mailing lists to ensure that it had no known bugs. The third open communication channel, the NIC used by the open-box VM, remains in the closed-box VM because Xen's design places all hardware drivers in Domain0. We considered mapping this driver directly into DomainU, but doing so would create challenging security issues related to DMA transfers that are best addressed with specialized hardware support (SecVisor [43] discusses these issues in detail). Instead, we use firewall rules to ensure that all outbound communication on this NIC originates from the open-box VM. As with the SCSI driver, this is a generic Linux gigabit NIC driver, and we verified that it had no known bugs. The final open communication channel is constructed by installing a virtual NIC in both the closed-box and open-box VMs and then building a virtual network between them. Typical for most Xen environments, this configuration permits communication across different Domains. As with the SCSI driver, we checked that it had no known security vulnerabilities.

5.1.3 Attacking the One-Way-Initiation Interface

Upon startup, Bunker firewalls the interface between the open-box VM and the closed-box VM using iptables. The rules used to configure iptables dictate that no connections are allowed unless they originate from the closed-box VM (see Figure 3). We re-used a set of rules from an iptables configuration for firewalling home environments found on the Internet.

5.2 Attacking Hardware

Bunker protects the closed-box VM from hardware attacks by making it safe-on-reboot. If an attacker turns off the machine to tamper with the hardware (e.g. by removing existing hardware or installing new hardware), the sensitive data contained in the closed-box VM is effectively destroyed. This is because the encryption keys and any unencrypted data are only stored in volatile memory (RAM). Therefore, hardware attacks must be mounted while the system is running. Section 5.1.2 discusses how we eliminated all unnecessary drivers from Bunker; this protects Bunker against attacks relying on adding new system devices, such as USB devices.

Another class of hardware attacks is one in which the attacker attempts to extract sensitive data (e.g., the encryption keys) from RAM. Such attacks can be mounted in many ways. A recent project demonstrated that the contents of today's RAM modules may remain readable even minutes after the system has been powered off [21]. Bunker is vulnerable to such attacks: an attacker could try to extract the encryption keys from memory by removing the RAM modules from the tracing machine and placing them into one configured to run key-searching software over memory on bootup [21]. Another approach is to attach a bus monitor to observe traffic on the memory bus. Preventing RAM-based attacks requires specialized hardware, which we discuss below. Yet another way is to attach a specialized device, such as certain Firewire devices, that can initiate DMA transfers without any support from software running on the host [37, 14]. Preventing this attack requires either 1) disabling the Firewire controller or 2) support from an IOMMU to limit which memory regions can be accessed by Firewire devices.

Secure Co-processors Can Prevent Hardware Attacks: A secure co-processor contains a CPU packaged with a moderate amount of non-volatile memory enclosed in a tamper-resistant casing [44]. A secure co-processor would let Bunker store the encryption and anonymization keys, the unencrypted trace data and the code in a secure environment. It also allows the code to be executed within the secure environment.

Trusted Platform Modules (TPMs) Cannot Prevent Hardware Attacks: Unfortunately, the use of TPMs would not significantly help Bunker survive hardware attacks. The limited storage and execution capabilities of a TPM cannot fully protect encryption keys and other sensitive data from an adversary with physical access [21]. This is because symmetric encryption and decryption are not performed directly by the TPM; these operations are still handled by the system's CPU. Therefore, the encryption keys must be exposed to the OS and stored in RAM, making them subject to the attack types mentioned above.

5.3 Attacking the Trace Analysis Software

An attacker could inject carefully crafted network traffic to exploit a vulnerability in the trace analysis software, such as a buffer overflow. Because this software does not run as root, such attacks cannot disable the narrow interfaces of the closed-box; the attacker needs root privileges to alter the OS drivers or the iptable's firewall rules. Nevertheless, such an attack could obtain access to sensitive data, skip the anonymization step, and send captured data directly to the open-box VM through the one-way-initiation interface.

While possible, such attacks are challenging to mount in practice for two reasons. First, Bunker's trace analysis software combines C (e.g., tcpdump plus a TCP/IP reconstruction library, which is a Linux 2.0 networking stack running in user-space), Python, and Perl. The C code is well-known and well-tested, making it less likely to have bugs that can be remotely exploited by injecting network traffic. Bunker's application-level parsing code is written in Python and Perl, two languages that are resistant to buffer overflows. In contrast, online anonymizers write all their parsing code in unmanaged languages (e.g., C or C++) in which it is much harder to handle code errors and bugs.

Second, a successful attack would send sensitive data to the open-box VM. The attacker must then find a way to extract the data from the open-box VM. To mitigate this possibility, we firewall the open-box's NIC to reject any traffic unless it originates from our own private network. Thus, to be successful, an attacker must not only find an exploitable bug in the trace analysis code but must also compromise the open-box VM through an attack that originates from our private network.

6 Operational Issues

At boot time, Bunker's bootloader asks the user to choose between two configurations: an ordinary one and a restricted one. The ordinary configuration loads a typical Xen environment with all drivers enabled. We use this environment only to prepare a tracing experiment and to configure Bunker; we never gather traces in it because it offers no privacy benefits. To initiate a tracing experiment, we boot into the restricted environment. When booting into this environment, Bunker's display and keyboard freeze because no drivers are being loaded. In this configuration, we use the open NIC to log in to the open-box VM where we can monitor the anonymized traces received through the one-way-initiation interface. These traces also contain meta-data about the health of the closed-box VM, including a variety of counters (such as packets received, packets lost, usage of memory, and amount of free space on the encrypted disk).

Network studies often need traces that span weeks, months, or even years. The closed-box nature of Bunker and its long-term use raise the possibility of the following operational attack: an intruder gains physical access to Bunker, reboots it, and sets it up with a fake restricted environment that behaves like Bunker's restricted environment but uses encryption and anonymization keys known to the intruder. This attack could remain undetected by network operators. From the outside, Bunker seems to have gathered network traces continuously.

To prevent this attack, Bunker could generate a public/private key-pair upon starting the closed-box VM. The public key would be shared with the network operator who saves an offline copy, while the private key would never be released from the closed-box VM. To verify that Bunker's code has not been replaced, the closed-box VM would periodically send a heartbeat message through the one-way-initiation interface to the open-box. The heartbeat message would contain the experiment's start time, the current time, and additional counters, all signed with the private key to let network operators verify that Bunker's original closed-box remains the one currently running. This prevention mechanism is not currently implemented.

7 Evaluation

This section presents a three-pronged evaluation of Bunker. First, we measure the performance overhead introduced by virtualization and encryption. Second, we evaluate Bunker's software engineering benefits when compared to online tracing tools. Third, we conduct an experiment to show Bunker's fault handling benefits.

7.1 Performance Overhead

To evaluate the performance overhead of virtualization and encryption, we ran *tcpdump* (i.e., Bunker's online component) to capture all traffic traversing a gigabit link and store it to disk. We measured the highest rate of traffic tcpdump can capture with no packet losses under four configurations: standalone, running in a Xen VM, running on top of an encrypted disk with *dm-crypt* [41], and running on top of an encrypted disk in a Xen VM.

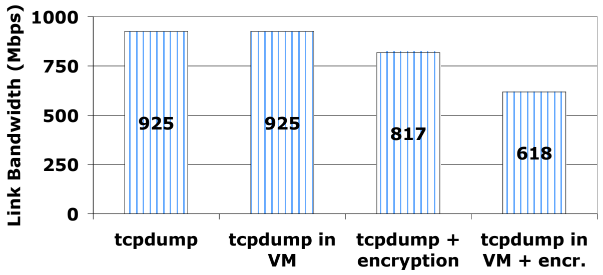


Figure 6. ***Performance overhead of virtualization and encryption:*** *We measured the rate of traffic that tcpdump can capture on our machine with no packet losses under four configurations: standalone, running in a Xen VM, running on top of an encrypted file system, and running on top of an encrypted file system in a Xen VM. All output captured by tcpdump was written to the disk.*

Our tracing host is a dual Intel Xeon 3.0GHz with 4 GB of RAM, six 150 GB SCSI hard-disk drives, and a DAG 4.3GE capture card. We ran Linux Debian 4.0 (etch), kernel version 2.6.18-4 and attached the tracer to a dedicated Dell PowerConnect 2724 gigabit switch with two other commodity PCs attached. One PC sent constant bit-rate (CBR) traffic at a configurable rate to the other; the switch was configured to mirror all traffic to our tracing host. We verified that no packets were being dropped by the switch.

Figure 6 shows the results of this experiment. The first bar shows that we capture 925 Mbps when running tcpdump on the bare machine with no isolation. The limiting factor in this case is the rate at which our commodity PCs can exchange CBR traffic; even after fine tuning, they can exchange no more than 925 Mbps on our gigabit link. The second bar shows that running tcpdump inside the closed-box VM has no measurable effect on the capture rate because the limiting factor remains our traffic injection rate. When we use the Linux dm-crypt module for encryption, however, the capture rate drops to 817 Mbps even when running on the bare hardware: the CPU becomes the bottleneck when running the encryption module. Combining both virtualization and encryption shows a further drop in the capture rate, to 618 Mbps. Once the CPU is fully utilized by the encryption module, the additional virtualization costs become apparent.

Our implementation of Bunker can trace network traffic of up to 618 Mbps with no packet loss. This is sufficiently fast for the tracing scenario that our university requires. While the costs of encryption and virtualization are not negligible, we believe that these overheads will decrease over time as Linux and Xen incorporate further optimizations to their block-level encryption and virtualization software. At the same time, CPU manufacturers have started to incorporate hardware acceleration for AES encryption (i.e., similar to what dm-crypt uses) [46].

7.2 Software Engineering Benefits

As previously discussed, Bunker offers significant software engineering benefits over online network tracing systems. Figure 7 shows the number of lines of code for three network tracing systems that perform HTTP parsing, all developed by this paper's authors. The first two systems trace HTTP traffic at line speeds. The first system was developed from scratch by two graduate students over the course of one year. The second system was developed by one graduate student in nine months; this system was built on top of CoMo, a packet-level tracing system developed by Intel Research [22]. Bunker is the third system; it was developed by one student in two months. As Figure 7 shows, Bunker's codebase is an order of magnitude smaller than the others. Moreover, we wrote only about one fifth of Bunker's code; the remainder was re-used from libraries.

Bunker's smaller and simpler codebase comes at a cost in terms of its offline component's performance. Figure 8 shows the time elapsed for Bunker's online and offline components to process a 5 minute trace of HTTP traffic. The trace contains 4.5 million requests, or about 15,000 requests per second, that we generated using *httpperf*. In practice, very few traces contain that many HTTP requests per second. While the online component runs only tcpdump storing data to the disk, the offline component performs TCP/IP reconstruction, parses HTTP, and records the HTTP headers before copying the trace to the open-box VM. The offline component spends 20 minutes and 28 seconds processing this trace. Clearly, Bunker's ease of development comes at the cost of performance, as we did not optimize the HTTP parser at all. The privacy guarantees of our isolated environment grant us the luxury of re-using existing software components even though they do not meet the performance demands of online tracing.

7.3 Fault Handling Evaluation

In addition to supporting fast development of different tracing experiments, Bunker handles bugs in the tracing software robustly. Upon encountering a bug, Bunker marks the offending flow as "erroneous" and continues processing traffic without having to restart. To illustrate the benefits of this fault handling approach, we performed the following experiment. We used Bunker on a Saturday to gather a 20 hour trace of the HTTP traffic our university exchanges with the Internet. This trace contained over 5.2 million HTTP flows. We artificially

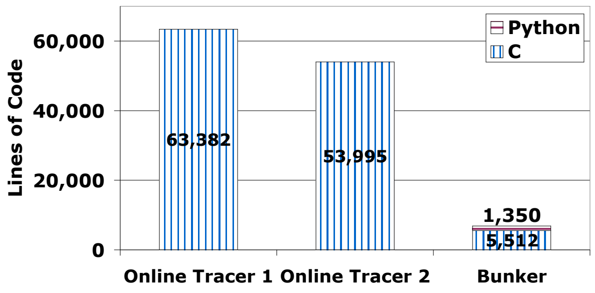


Figure 7. ***Lines of Code in three systems for gathering HTTP traces:*** *The first system was developed from scratch by two graduate students in one year. The second system, an extension of CoMo [22], was developed by one graduate student in nine months; we included CoMo's codebase when counting the size of this system's codebase. The third system, Bunker, was developed by one student in two months.*

injected a parsing bug in one packet out of 100,000 (corresponding to a parsing error rate of 0.001%). Upon encountering this bug, Bunker stops parsing the erroneous HTTP flow and continues with the remaining flows. We compare Bunker to an online tracer that would crash upon encountering a bug and immediately restart. This would result in the online tracer dropping all concurrent flows (we refer to this as "collateral damage"). This experiment assumes an idealized version of an online tracer that restarts instantly; in practice, it takes tens of seconds to restart an online tracer's environment losing even more ongoing flows. Figure 9 illustrates the difference in the fraction of flows affected. While our bug is encountered in only 0.08% of the flows, it affects an additional 31.72% of the flows for an online tracing system. Not one of these additional flows is affected by the bug when Bunker performs the tracing.

8 Legal Background

This section presents legal background concerning the issuing of subpoenas for network traces in the U.S. and Canada and discusses legal issues inherent in designing and deploying data-hiding tracing platforms[2].

8.1 Issuing Subpoenas for Data Traces

U.S. law has two sets of requirements for obtaining a data trace that depend on when the data was gathered. For data traces gathered in the past 180 days, the government needs a *mere subpoena*. Such subpoenas are obtained from a federal or state court with jurisdiction over the offense under investigation. Based on our conversations with legal experts, obtaining a subpoena is relatively simple in the context of a lawsuit. A defendant

[2]Any mistakes in our characterization of the U.S. or Canadian legal systems are the sole responsibility of the authors and not the lawyers we consulted during this research project.

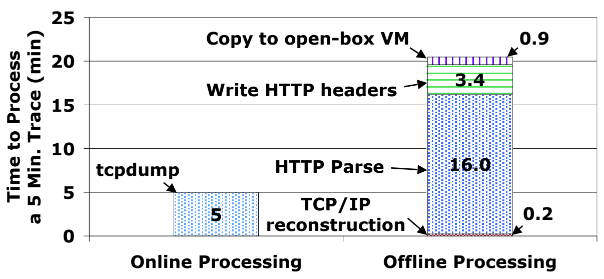


Figure 8. ***Online vs. Offline processing speed:*** *The time spent processing a five minute HTTP trace by Bunker's online and offline components, respectively.*

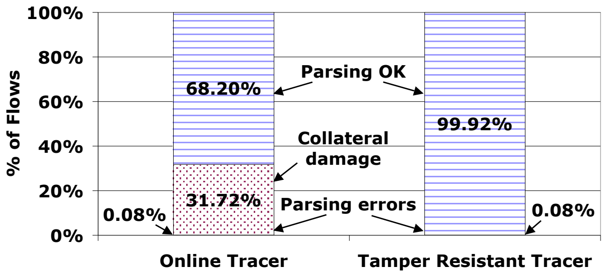


Figure 9. ***Fraction of flows affected by a bug in an online tracer versus in Bunker:*** *A bug crashing an online tracer affects all flows running concurrent with the crash. Instead, Bunker handles bugs using exceptions affecting only the flows that triggered the bug.*

(e.g., the ISP) could try to quash the subpoena if compliance would be unreasonable or oppressive.

For data gathered more than 180 days earlier, a government entity needs a warrant under Title 18 United States Code 2703(d) from a federal or state court with appropriate jurisdiction. The government needs to present "specific and articulable facts showing that there are reasonable grounds to believe that the contents of a wire or electronic communication, or the records or other information sought, are relevant and material to an ongoing criminal investigation." The defendant can quash the subpoena if the information requested is "unusually voluminous in nature" or compliance would cause undue burden. Based on our discussions with legal experts, the court would issue such a warrant if it determines that the data is relevant and not duplicative of information already held by the government entity.

In Canada, a subpoena is sufficient to obtain a data trace regardless of the data's age. In 2000, the Canadian government passed the Personal Information Protection and Electronic Documents Act (PIPEDA) [33], which enhances the users' rights to privacy for their data held by private companies such as ISPs. However, Section 7(3)(c.1) of PIPEDA indicates that ISPs must disclose personal information (including data traces) if they are served with a subpoena or even an "order made by a court, person or body with jurisdiction to compel production of information". In a recent case, a major Canadian ISP released personal information to the local police based on a letter that stated that "the request was done under the authority of PIPEDA" [32]. A judge subsequently found that prior authorization for this information should have been obtained, and the ISP should not have disclosed this information. This case illustrates the complexity of the legal issues ISPs face when they store personal information (e.g., raw network traces).

8.2 Developing Data-Hiding Technology

In our discussions with legal experts, we investigated whether it is legal to develop and deploy a data-hiding network tracing infrastructure (such as Bunker). While there is no clear answer to this question without legal precedent, we learned that the way to evaluate this question is to consider the purpose and potential uses for the technology in question. In general, it is legal to deploy a technology that has many legitimate uses but could also enable certain illegitimate uses. Clearly, technologies whose primary use is to enable or encourage users to evade the law are not legal. A useful example to illustrate this distinction is encryption technology. While encryption can certainly be used to enable illegal activities, its many legitimate uses make development and deployment of encryption technologies legal. In the context of network tracing, protecting users' privacy against accidental loss or mismanagement of the trace data is a legitimate purpose.

9 Related Work

Bunker draws on previous work in network tracing systems, data anonymizing techniques, and virtual machine usage for securing systems. We summarize this previous work and then we describe two systems built to protect access to sensitive data, such as network traces.

9.1 Network Tracing Systems

One of the earliest network tracing systems was Httpdump [51], a tcpdump extension that constructs a log of HTTP requests and responses. Windmill [30] developed a custom packet filter that facilitates the building of specific network analysis applications; it delivers captured packets to multiple filters using dynamic code generation. BLT [18], a network tracing system developed specifically to study HTTP traffic, supports continuous online network monitoring. BLT does not use online anonymization; instead, it records raw packets directly to disk. More recently, CoMo [22] was designed to allow independent parties to run multiple ongoing trace analysis modules by isolating them from each other. With CoMo, anonymization, whether online or offline, must be implemented by each module's owner. Unlike these systems, Bunker's design was motivated by the need to protect the privacy of network users.

9.2 Anonymization Techniques

Xu et al. [52] implemented a prefix-preserving anonymization scheme for IP addresses, i.e., addresses with the same IP prefix share the same prefix after anonymization. Pang et al. [35] designed a high-level language for specifying anonymization policies, allowing researchers to write short policy scripts to express trace transformations. Recent work has shown that traces can still leak private information even after they are anonymized [34], prompting the research community to propose a set of guidelines and etiquette for sharing data traces [1]. Bunker's goal is to create a tracing system that makes it easy to develop trace analysis software while ensuring that no raw data can be exposed from the closed-box VM. Bunker does not protect against faulty anonymization policies, nor does it ensure that anonymized data cannot be subject to the types of attacks described in [34].

9.3 Using VMs for Making Systems Secure

An active research area is designing virtual machine architectures that are secure in the face of attacks. Several solutions have been proposed, including: using tamper-resistant hardware [28, 20]; designing VMMs that are small enough for formal verification [25, 40]; using programming language techniques to provide memory safety and control-flow integrity in commodity OS'es [26, 12]; and using hardware memory protection to provide code integrity [43]. While these systems attempt to secure a general purpose commodity OS, Bunker was designed only to secure tracing software. As a result, its interfaces are simple and narrow.

9.4 Protecting Access to Sensitive Data

Packet Vault [3] is a network tracing system that captures packets, encrypts them, and writes them to a CD. A newer system design tailored for writing the encrypted traces to tape appears in [2]. Packet Vault creates a permanent record of all network traffic traversing a link. Its threat model differs from Bunker's in that there is no attempt to secure the system against physical attacks.

Armored Data Vault [24] is a system that implements access control to previously collected network traces, by using a secure co-processor to enforce security in the face of malicious attackers. Like Bunker, network traces are encrypted before being stored. The encryption key and any raw data are stored inside the secure co-processor. Bunker's design differs from Armored Data Vault's in three important ways. First, Bunker's goal is limited to trace anonymization only and not to implementing access control policies; this lets us use simple, off-the-shelf anonymization code to minimize the likelihood of bugs present in the system. Second, Bunker destroys the raw data as soon as it is anonymized; the Armored Data Vault stores its raw traces permanently while enforcing the data access policy. Finally, Bunker uses commodity hardware that can run unmodified off-the-shelf software. Instead, the authors of the Armored Data Vault had to port their code to accommodate the specifics of the secure co-processor, a process that required effort and affected the system's performance [24].

10 Conclusions

This paper presents Bunker, a network tracing architecture that combines the performance and software engineering benefits of offline anonymization with the privacy offered by online anonymization. Bunker uses a closed-box and safe-on-reboot architecture to protect raw trace data against a large class of security attacks, including physical attacks to the system. In addition to its security benefits, our architecture improves ease of development: using Bunker, one graduate student implemented a network tracing system for gathering anonymized traces of Hotmail e-mail in less than two months.

Our evaluation shows that Bunker has adequate performance. We show that Bunker's codebase is an order of magnitude smaller than previous network tracing systems that perform online anonymization. Because most of its data processing is performed offline, Bunker also handles faults more gracefully than previous systems.

Acknowledgments: We are grateful to Gianluca Iannaccone and Intel Cambridge for donating the hardware for Bunker. We thank John Dunagan, Steve Gribble, Steve Hand (our shepherd), Srikanth Kandula, Andy Warfield, and the anonymous reviewers for their insightful feedback.

References

[1] M. Allman and V. Paxson. Issues and etiquette concerning use of shared measurement data. In *Proceedings of the Internet Measurement Conference (IMC)*, San Diego, CA, October 2007.

[2] C. J. Antonelli, K. Coffman, J. B. Fields, and P. Honeyman. Cryptographic wiretapping at 100 megabits. In *16th SPIE Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Controls*, Orlando, FL, April 2002.

[3] C. J. Antonelli, M. Undy, and P. Honeyman. The packet vault: Secure storage of network data. In *1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA, April 1999.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, pages 164–177, Bolton Landing, NY, 2003.

[5] BBC News. Brown apologises for records loss, November 2007. `http://news.bbc.co.uk/1/hi/uk_politics/7104945.stm`.

[6] S. M. Bellovin. Virtual machines, virtual security. *Communications of the ACM*, 49(10), 2006.

[7] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, Elmau, Germany, May 2001.

[8] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. In *Proc. of the 18th ACM Symposium on Operating System Principles (SOSP)*, pages 73–88, Banff, Canada, October 2001.

[9] M. P. Collins and M. K. Reiter. Finding p2p file-sharing using coarse network behaviors. In *Proceedings of ESORICS*, 2006.

[10] S. Coull, C. Wright, F. Monrose, M. Collins, and M. Reiter. Playing devil's advocate: Inferring sensitive information from anonymized traces. In *Proceedings of the USENIX Security Symposium*, Boston, MA, August 2007.

[11] R. S. Cox, S. D. Gribble, H. M. Levy, and J. G. Hansen. A safety-oriented platform for web applications. In *27th IEEE Symposium on Security and Privacy*, May 2006.

[12] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *21st ACM SOSP*, Stevenson, WA, Oct 2007.

[13] S. Dharmapurikar and V. Paxson. Robust TCP stream reassembly in the presence of adversaries. In *14th USENIX Security Symposium*, Baltimore, MD, July 2005.

[14] M. Dornseif. 0wned by an ipod, 2004. `http://md.hudora.de/presentations/firewire/PacSec2004.pdf`.

[15] N. Duffield, C. Land, and M. Thorup. Charging from sampled network usage. In *Proceedings of Internet Measurement Workshop (IMW)*, San Francisco, CA, November 2001.

[16] Electronic Frontier Foundation. RIAA v. Verizon case archive. `http://www.eff.org/legal/cases/RIAA_v_Verizon/`.

[17] Endace. Dag network monitoring cards – ethernet, 2008. `http://www.endace.com/our-products/dag-network-monitoring-cards/etherne%t`.

[18] A. Feldmann. BLT: Bi-layer tracing of HTTP and TCP/IP. In *9th International World Wide Web Conference*, Amsterdam, Holland, May 2000.

[19] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving traffic demands for operational ip networks: methodology and experience. *IEEE/ACM Transactions on Networking (TON)*, 9(3):265–280, 2001.

[20] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, October 2003.

[21] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the 17th USENIX Security Symposium*, San Jose, CA, July 2008.

[22] G. Iannaccone. CoMo: An open infrastructure for network monitoring – research agenda. Technical report, Intel Research, 2005.

[23] IBM. sHype – Secure Hypervisor. `http://www.research.ibm.com/secure_systems_department/projects/hypervis%or/`.

[24] A. Iliev and S. Smith. Prototyping an armored data vault: Rights management on big brother's computer. In *Privacy-Enhancing Technologies*, San Francisco, CA, April 2002.

[25] K. Kaneda. Tiny virtual machine monitor, 2006. `http://web.yl.is.s.u-tokyo.ac.jp/~kaneda/tvmm/`.

[26] V. Kiriansky, D. Bruening, and S. P. Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[27] A. Lakhina, M. Crovella, and C. Diot. Diagnosing network-wide traffic anomalies. In *Proceedings of 2004 Conference on Applications, Technologies, Architectures, and Protocols for computer communications (SIGCOMM)*, Portland, OR, September 2004.

[28] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *19th ACM SOSP*, Bolton Landing, NY, October 2003.

[29] S. E. Madnick and J. J. Donovan. Application and analysis of the virtual machine approach to information system security and isolation. In *Proc. of the Workshop on Virtual Computer Systems*, Cambridge, MA, 1973.

[30] G. R. Malan and F. Jahanian. An extensible probe architecture for network protocol performance measurement. In *Proceedings of the 1998 Conference on Applications, Technologies, Architectures, and Protocols for computer communications (SIGCOMM)*, Vancouver, BC, September 1998.

[31] J. Mirkovic and P. Reiher. A taxonomy of DDoS attack and defense mechanisms. *ACM SIGCOMM Computer Communications Review*, 34(2):39–53, 2004.

[32] Mondaq. Canada: Disclosure of personal information without consent pursuant to lawful authority, 2007. `http://www.mondaq.com/article.asp?article_id=53904`.

[33] Office of the Privacy Commissioner of Canada. Personal information protection and electronic documents act, 2000. `http://www.privcom.gc.ca/legislation/02_06_01_01_e.asp`.

[34] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *ACM SIGCOMM Computer Communication Review*, 36(1):29–38, 2006.

[35] R. Pang and V. Paxson. A high-level programming environment for packet trace anonymization and transformation. In *9th ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[36] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet background radiation. In *Proc. of IMC)*, Taormina, Italy, October 2004.

[37] P. Panholzer. Physical security attacks on windows vista. Technical report, SEC Consult Unternehmensberatung GmbH, 2008. `http://www.sec-consult.com/fileadmin/Whitepapers/Vista_Physical_Attacks%.pdf`.

[38] K. Papagiannaki, N. Taft, Z.-L. Zhang, and C. Diot. Long-term forecasting of internet backbone traffic: Observations and initial models. In *Proc. of INFOCOM*, San Francisco, CA, April 2003.

[39] N. Provos. Encrypting virtual memory. In *9th USENIX Security Symposium*, Denver, CO, August 2000.

[40] R. Russell. Lguest: The simple x86 hypervisor, 2008. `http://lguest.ozlabs.org/`.

[41] C. Saout. dm-crypt: a device-mapper crypto target, 2007. `http://www.saout.de/misc/dm-crypt/`.

[42] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet content delivery systems. In *Proc. of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, Boston, MA, December 2002.

[43] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity OSes. In *21st ACM SOSP*, October 2007.

[44] J. D. Tygar and B. Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[45] R. Vasudevan and Z. M. Mao. Reval: A tool for real-time evaluation of ddos mitigation strategies. In *Proceedings of the 2006 Usenix Annual Technical Conference*, 2006.

[46] VIA. Via padlock security engine. `http://www.via.com.tw/en/initiatives/padlock/hardware.jsp`.

[47] VMware. VMware VMsafe Security Technology. `http://www.vmware.com/overview/security/vmsafe.html`.

[48] R. Wojtczuk. libNids, 2008. `http://libnids.sourceforge.net/`.

[49] A. Wolman. *Sharing and Caching Characteristics of Internet Content*. PhD thesis, Univ. of Washington, Seattle, WA, 2002.

[50] A. Wolman, G. Voelker, N. Sharma, N. Cardwell, M. Brown, T. Landray, D. Pinnel, A. Karlin, and H. Levy. Organization-based analysis of web-object sharing and caching. In *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems (USITS)*, Boulder, CO, October 1999.

[51] R. Wooster, S. Williams, and P. Brooks. HTTPDUMP network HTTP packet snooper, April 1996. `http://ei.cs.vt.edu/~succeed/96httpdump/`.

[52] J. Xu, J. Fan, M. Ammar, and S. B. Moon. On the design and performance of prefix-preserving IP traffic trace anonymization. In *1st ACM SIGCOMM IMW Workshop*, 2001.

[53] J. Yang, C. Sar, P. Twohey, C. Cadar, and D. Engler. Automatically generating malicious disks using symbolic execution. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 243–257, Oakland, CA, May 2006.

[54] T. Zeller Jr. AOL executive quits after posting of search data. *International Herald Tribune*, August 2006.

Flexible, Wide-Area Storage for Distributed Systems with WheelFS

Jeremy Stribling, Yair Sovran,[†] Irene Zhang, Xavid Pretzer,
Jinyang Li,[†] M. Frans Kaashoek, and Robert Morris
MIT CSAIL [†]*New York University*

Abstract

WheelFS is a wide-area distributed storage system intended to help multi-site applications share data and gain fault tolerance. WheelFS takes the form of a distributed file system with a familiar POSIX interface. Its design allows applications to adjust the tradeoff between prompt visibility of updates from other sites and the ability for sites to operate independently despite failures and long delays. WheelFS allows these adjustments via *semantic cues*, which provide application control over consistency, failure handling, and file and replica placement.

WheelFS is implemented as a user-level file system and is deployed on PlanetLab and Emulab. Three applications (a distributed Web cache, an email service and large file distribution) demonstrate that WheelFS's file system interface simplifies construction of distributed applications by allowing reuse of existing software. These applications would perform poorly with the strict semantics implied by a traditional file system interface, but by providing cues to WheelFS they are able to achieve good performance. Measurements show that applications built on WheelFS deliver comparable performance to services such as CoralCDN and BitTorrent that use specialized wide-area storage systems.

1 Introduction

There is a growing set of Internet-based services that are too big, or too important, to run at a single site. Examples include Web services for e-mail, video and image hosting, and social networking. Splitting such services over multiple sites can increase capacity, improve fault tolerance, and reduce network delays to clients. These services often need storage infrastructure to share data among the sites. This paper explores the use of a new file system specifically designed to be the storage infrastructure for wide-area distributed services.

A wide-area storage system faces a tension between sharing and site independence. The system must support sharing, so that data stored by one site may be retrieved by others. On the other hand, sharing can be dangerous if it leads to the unreachability of one site causing blocking at other sites, since a primary goal of multi-site operation is fault tolerance. The storage system's consistency model affects the sharing/independence tradeoff: stronger forms of consistency usually involve servers or quorums of servers that serialize all storage operations, whose unreliability may force delays at other sites [23]. The storage system's data and meta-data placement decisions also affect site independence, since data placed at a distant site may be slow to fetch or unavailable.

The wide-area file system introduced in this paper, WheelFS, allows application control over the sharing/independence tradeoff, including consistency, failure handling, and replica placement. Each application can choose a tradeoff between performance and consistency, in the style of PRACTI [8] and PADS [9], but in the context of a file system with a POSIX interface.

Central decisions in the design of WheelFS including defining the default behavior, choosing which behaviors applications can control, and finding a simple way for applications to specify those behaviors. By default, WheelFS provides standard file system semantics (close-to-open consistency) and is implemented similarly to previous wide-area file systems (*e.g.*, every file or directory has a primary storage node). Applications can adjust the default semantics and policies with *semantic cues*. The set of cues is small (around 10) and directly addresses the main challenges of wide-area networks (orders of magnitude differences in latency, lower bandwidth between sites than within a site, and transient failures). WheelFS allows the cues to be expressed in the pathname, avoiding any change to the standard POSIX interface. The benefits of WheelFS providing a file system interface are compatibility with existing software and programmer ease-of-use.

A prototype of WheelFS runs on FreeBSD, Linux, and MacOS. The client exports a file system to local applications using FUSE [21]. WheelFS runs on PlanetLab and an emulated wide-area Emulab network.

Several distributed applications run on WheelFS and demonstrate its usefulness, including a distributed Web cache and a multi-site email service. The applications use different cues, showing that the control that cues provide is valuable. All were easy to build by reusing existing software components, with WheelFS for storage instead of a local file system. For example, the Apache caching web proxy can be turned into a distributed, cooperative Web cache by modifying one pathname in a configuration file, specifying that Apache should store cached data in WheelFS with cues to relax consistency. Although the other applications require more changes, the ease of adapting Apache illustrates the value of a file system interface; the extent to which we could reuse non-distributed software in distributed applications came as a surprise [38].

Measurements show that WheelFS offers more scalable performance on PlanetLab than an implementation of NFSv4, and that for applications that use cues to indicate they can tolerate relaxed consistency, WheelFS continues to provide high performance in the face of network and server failures. For example, by using the cues **.EventualConsistency**, **.MaxTime**, and **.Hotspot**, the distributed Web cache quickly reduces the load on the origin Web server, and the system hardly pauses serving pages when WheelFS nodes fail; experiments on PlanetLab show that the WheelFS-based distributed Web cache reduces origin Web server load to zero. Further experiments on Emulab show that WheelFS can offer better file downloads times than BitTorrent [14] by using network coordinates to download from the caches of nearby clients.

The main contributions of this paper are a new file system that assists in the construction of wide-area distributed applications, a set of cues that allows applications to control the file system's consistency and availability tradeoffs, and a demonstration that wide-area applications can achieve good performance and failure behavior by using WheelFS.

The rest of the paper is organized as follows. Sections 2 and 3 outline the goals of WheelFS and its overall design. Section 4 describes WheelFS's cues, and Section 5 presents WheelFS's detailed design. Section 6 illustrates some example applications, Section 7 describes the implementation of WheelFS, and Section 8 measures the performance of WheelFS and the applications. Section 9 discusses related work, and Section 10 concludes.

2 Goals

A wide-area storage system must have a few key properties in order to be practical. It must be a useful building block for larger applications, presenting an easy-to-use interface and shouldering a large fraction of the overall storage management burden. It must allow inter-site access to data when needed, as long as the health of the wide-area network allows. When the site storing some data is not reachable, the storage system must indicate a failure (or find another copy) with relatively low delay, so that a failure at one site does not prevent progress at other sites. Finally, applications may need to control the site(s) at which data are stored in order to achieve fault-tolerance and performance goals.

As an example, consider a distributed Web cache whose primary goal is to reduce the load on the origin servers of popular pages. Each participating site runs a Web proxy and a part of a distributed storage system. When a Web proxy receives a request from a browser, it first checks to see if the storage system has a copy of the requested page. If it does, the proxy reads the page from the storage system (perhaps from another site) and serves it to the browser. If not, the proxy fetches the page from the origin Web server, inserts a copy of it into the storage system (so other proxies can find it), and sends it to the browser.

The Web cache requires some specific properties from the distributed storage system in addition to the general ability to store and retrieve data. A proxy must serve data with low delay, and can consult the origin Web server if it cannot find a cached copy; thus it is preferable for the storage system to indicate "not found" quickly if finding the data would take a long time (due to timeouts). The storage need not be durable or highly fault tolerant, again because proxies can fall back on the origin Web server. The storage system need not be consistent in the sense of guaranteeing to find the latest stored version of document, since HTTP headers allow a proxy to evaluate whether a cached copy is still valid.

Other distributed applications might need different properties in a storage system: they might need to see the latest copy of some data, and be willing to pay a price in high delay, or they may want data to be stored durably, or have specific preferences for which site stores a document. Thus, in order to be a usable component in many different systems, a distributed storage system needs to expose a level of control to the surrounding application.

3 WheelFS Overview

This section gives a brief overview of WheelFS to help the reader follow the design proposed in subsequent sections.

3.1 System Model

WheelFS is intended to be used by distributed applications that run on a collection of sites distributed over the wide-area Internet. All nodes in a WheelFS deployment are either managed by a single administrative entity or multiple cooperating administrative entities. WheelFS's security goals are limited to controlling the set of participating servers and imposing UNIX-like access controls on clients; it does not guard against Byzantine failures in participating servers [6, 26]. We expect servers to be live and reachable most of the time, with occasional failures. Many existing distributed infrastructures fit these assumptions, such as wide-area testbeds (*e.g.*, PlanetLab and RON), collections of data centers spread across the globe (*e.g.*, Amazon's EC2), and federated resources such as Grids.

3.2 System Overview

WheelFS provides a location-independent hierarchy of directories and files with a POSIX file system interface. At any given time, every file or directory object has a single "primary" WheelFS storage server that is responsible for maintaining the latest contents of that object. WheelFS clients, acting on behalf of applications, use the storage servers to retrieve and store data. By default, clients consult the primary whenever they modify an object or need to find the latest version of an object. Accessing a single file could result in communication with several servers, since each subdirectory in the path could be served by a different primary. WheelFS replicates an object's data using primary/backup replication, and a background maintenance process running on each server ensures that data are replicated correctly. Each update to an object increments a version number kept in a separate meta-data structure, co-located with the data.

When a WheelFS client needs to use an object, it must first determine which server is currently the primary for that object. All nodes agree on the assignment of objects to primaries to help implement the default strong consistency. Nodes learn the assignment from a *configuration service*—a replicated state machine running at multiple sites. This service maintains a table that maps each object to one primary and zero or more backup servers. WheelFS nodes cache a copy of this table. Section 5 presents the design of the configuration service.

A WheelFS client reads a file's data in blocks from the file's primary server. The client caches the file's data once read, obtaining a lease on its meta-data (including the version number) from the primary. Clients have the option of reading from other clients' caches, which can be helpful for large and popular files that are rarely updated. WheelFS provides close-to-open consistency by default for files, so that if an application works correctly on a POSIX file system, it will also work correctly on WheelFS.

4 Semantic cues

WheelFS provides semantic cues within the standard POSIX file system API. We believe cues would also be useful in the context of other wide-area storage layers with alternate designs, such as Shark [6] or a wide-area version of BigTable [13]. This section describes how applications specify cues and what effect they have on file system operations.

4.1 Specifying cues

Applications specify cues to WheelFS in pathnames; for example, /wfs/**.Cue**/data refers to /wfs/data with the cue **.Cue**. The main advantage of embedding cues in pathnames is that it keeps the POSIX interface unchanged. This choice allows developers to program using an interface with which they are familiar and to reuse software easily.

One disadvantage of cues is that they may break software that parses pathnames and assumes that a cue is a directory. Another is that links to pathnames that contain cues may trigger unintuitive behavior. We have not encountered examples of these problems.

WheelFS clients process the cue path components locally. A pathname might contain several cues, separated by slashes. WheelFS uses the following rules to combine cues: (1) a cue applies to all files and directories in the pathname appearing after the cue; and (2) cues that are specified later in a pathname may override cues in the same category appearing earlier.

As a preview, a distributed Web cache could be built by running a caching Web proxy at each of a number of sites, sharing cached pages via WheelFS. The proxies could store pages in pathnames such as /wfs/**.MaxTime**=200/url, causing `open()` to fail after 200 ms rather than waiting for an unreachable WheelFS server, indicating to the proxy that it should fetch from the original Web server. See Section 6 for a more sophisticated version of this application.

4.2 Categories

Table 1 lists WheelFS's cues and the categories into which they are grouped. There are four categories: placement, durability, consistency, and large reads. These categories reflect the goals discussed in Section 2. The placement cues allow an application to reduce latency by placing data near where it will be needed. The durability and consistency cues help applications avoid data unavailability and timeout delays caused by transient failures. The large read cues increase throughput when reading large and/or popular files. Table 2 shows which POSIX file system API calls are affected by which of these cues.

Each cue is either *persistent* or *transient*. A persistent cue is permanently associated with the object, and may affect all uses of the object, including references that do not specify the cue. An application associates a persistent cue with an object by specifying the cue when first creating the object. Persistent cues are immutable after object creation. If an application specifies a transient cue in a file system operation, the cue only applies to that operation.

Because these cues correspond to the challenges faced by wide-area applications, we consider this set of cues to be relatively complete. These cues work well for the applications we have considered.

4.3 Placement

Applications can reduce latency by storing data near the clients who are likely to use that data. For example, a wide-area email system may wish to store all of a user's message files at a site near that user.

The **.Site=X** cue indicates the desired site for a newly-created file's primary. The site name can be a simple string, *e.g.* **.Site=westcoast**, or a domain name such as

Cue Category	Cue Name	Type	Meaning (and Tradeoffs)
Placement	**.Site=X**	P	Store files and directories on a server at the site named X.
	.KeepTogether	P	Store all files in a directory subtree on the same set of servers.
	.RepSites=N$_{RS}$	P	Store replicas across N$_{RS}$ different sites.
Durability	**.RepLevel=N$_{RL}$**	P	Keep N$_{RL}$ replicas for a data object.
	.SyncLevel=N$_{SL}$	T	Wait for only N$_{SL}$ replicas to accept a new file or directory version, reducing both durability and delay.
Consistency	**.EventualConsistency**	T*	Use potentially stale cached data, or data from a backup, if the primary does not respond quickly.
	.MaxTime=T	T	Limit any WheelFS remote communication done on behalf of a file system operation to no more than T ms.
Large reads	**.WholeFile**	T	Enable pre-fetching of an entire file upon the first read request.
	.Hotspot	T	Fetch file data from other clients' caches to reduce server load. Fetches multiple blocks in parallel if used with **.WholeFile**.

Table 1: Semantic cues. A cue can be either **P**ersistent or **T**ransient (*Section 4.5 discusses a caveat for **.EventualConsistency**).

Cue	open	close	read	write	stat	mkdir	rmdir	link	unlink	readdir	chmod
.S	X					X					
.KT	X					X					
.RS	X	X		X		X	X	X	X		X
.RL	X	X		X		X	X	X	X		X
.SL	X	X		X		X	X	X	X		X
.EC	X	X	X	X	X	X	X	X	X	X	X
.MT	X	X	X	X	X	X	X	X	X	X	X
.WF			X							X	
.H			X								

Table 2: The POSIX file system API calls affected by each cue.

.Site=rice.edu. An administrator configures the correspondence between site names and servers. If the path contains no **.Site** cue, WheelFS uses the local node's site as the file's primary. Use of **random** as the site name will spread newly created files over all sites. If the site indicated by **.Site** is unreachable, or cannot store the file due to storage limitations, WheelFS stores the newly created file at another site, chosen at random. The WheelFS background maintenance process will eventually transfer the misplaced file to the desired site.

The **.KeepTogether** cue indicates that an entire subtree should reside on as few WheelFS nodes as possible. Clustering a set of files can reduce the delay for operations that access multiple files. For example, an email system can store a user's message files on a few nodes to reduce the time required to list all messages.

The **.RepSites=N$_{RS}$** cue indicates how many different sites should have copies of the data. **N$_{RS}$** only has an effect when it is less than the replication level (see Section 4.4), in which case it causes one or more sites to store the data on more than one local server. When possible, WheelFS ensures that the primary's site is one of the sites chosen to have an extra copy. For example, specifying **.RepSites=2** with a replication level of three causes the primary and one backup to be at one site, and another backup to be at a different site. By using **.Site** and **.RepSites**, an application can ensure that a permanently failed primary can be reconstructed at the desired site with only local communication.

4.4 Durability

WheelFS allows applications to express durability preferences with two cues: **.RepLevel=N$_{RL}$** and **.SyncLevel=N$_{SL}$**.

The **.RepLevel=N$_{RL}$** cue causes the primary to store the object on N$_{RL}-1$ backups; by default, N$_{RL}=3$. The WheelFS prototype imposes a maximum of four replicas (see Section 5.2 for the reason for this limit; in a future prototype it will most likely be higher).

The **.SyncLevel=N$_{SL}$** cue causes the primary to wait for acknowledgments of writes from only N$_{SL}$ of the object's replicas before acknowledging the client's request, reducing durability but also reducing delays if some backups are slow or unreachable. By default, N$_{SL}$ = N$_{RL}$.

4.5 Consistency

The **.EventualConsistency** cue allows a client to use an object despite unreachability of the object's primary node, and in some cases the backups as well. For reads and pathname lookups, the cue allows a client to read from a backup if the primary is unavailable, and from the client's local cache if the primary and backups are both unavailable. For writes and filename creation, the cue allows a client to write to a backup if the primary is not available. A consequence of **.EventualConsistency** is that clients may not see each other's updates if they cannot all reliably contact the primary. Many applications such as Web caches and email systems can tolerate eventual consistency without significantly compromising their users' experience, and in return can decrease delays and reduce service unavailability when a primary or its network link are unreliable.

The cue provides eventual consistency in the sense that, in the absence of updates, all replicas of an object will eventually converge to be identical. However, WheelFS does not provide eventual consistency in the rigorous form (*e.g.*, [18]) used by systems like Bayou [39], where all updates, across all objects in the system, are committed in a total order at all replicas. In particular, updates in WheelFS are only eventually consistent with respect to the object they affect, and updates may potentially be lost. For example, if an entry is deleted from a directory under the **.EventualConsistency** cue, it could reappear in the directory later.

When reading files or using directory contents with eventual consistency, a client may have a choice between the contents of its cache, replies from queries to one or more backup servers, and a reply from the primary. A client uses the data with the highest version number that it finds within a time limit. The default time limit is one second, but can be changed with the **.MaxTime=T** cue (in units of milliseconds). If **.MaxTime** is used without eventual consistency, the WheelFS client yields an error if it cannot contact the primary after the indicated time.

The background maintenance process periodically reconciles a primary and its backups so that they eventually contain the same data for each file and directory. The process may need to resolve conflicting versions of objects. For a file, the process chooses arbitrarily among the replicas that have the highest version number; this may cause writes to be lost. For an eventually-consistent directory, it puts the union of files present in the directory's replicas into the reconciled version. If a single filename maps to multiple IDs, the process chooses the one with the smallest ID and renames the other files. Enabling directory merging is the only sense in which the **.EventualConsistency** cue is persistent: if specified at directory creation time, it guides the conflict resolution process. Otherwise its effect is specific to particular references.

4.6 Large reads

WheelFS provides two cues that enable large-file read optimizations: **.WholeFile** and **.Hotspot**. The **.WholeFile** cue instructs WheelFS to pre-fetch the entire file into the client cache. The **.Hotspot** cue instructs the WheelFS client to read the file from other clients' caches, consulting the file's primary for a list of clients that likely have the data cached. If the application specifies both cues, the client will read data in parallel from other clients' caches.

Unlike the cues described earlier, **.WholeFile** and **.Hotspot** are not strictly necessary: a file system could potentially learn to adopt the right cue by observing application access patterns. We leave such adaptive behavior to future work.

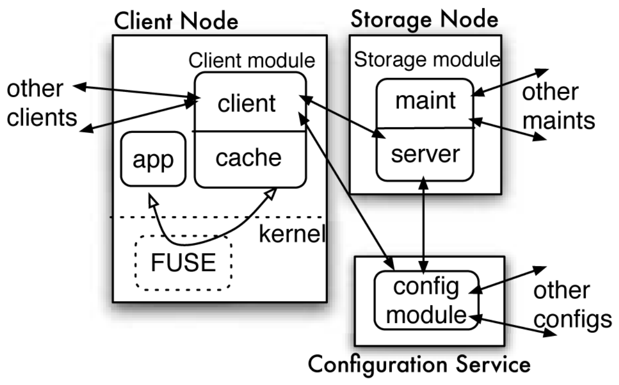


Figure 1: Placement and interaction of WheelFS components.

5 WheelFS Design

WheelFS requires a design flexible enough to follow the various cues applications can supply. This section presents that design, answering the following questions:

- How does WheelFS assign storage responsibility for data objects among participating servers? (Section 5.2)
- How does WheelFS ensure an application's desired level of durability for its data? (Section 5.3)
- How does WheelFS provide close-to-open consistency in the face of concurrent file access and failures, and how does it relax consistency to improve availability? (Section 5.4)
- How does WheelFS permit peer-to-peer communication to take advantage of nearby cached data? (Section 5.5)
- How does WheelFS authenticate users and perform access control? (Section 5.6)

5.1 Components

A WheelFS deployment (see Figure 1) consists of clients and servers; a single host often plays both roles. The WheelFS client software uses FUSE [21] to present the distributed file system to local applications, typically in `/wfs`. All clients in a given deployment present the same file system tree in `/wfs`. A WheelFS client communicates with WheelFS servers in order to look up file names, create files, get directory listings, and read and write files. Each client keeps a local cache of file and directory contents.

The configuration service runs independently on a small set of wide-area nodes. Clients and servers communicate with the service to learn the set of servers and which files and directories are assigned to which servers, as explained in the next section.

5.2 Data storage assignment

WheelFS servers store file and directory objects. Each object is internally named using a unique numeric ID. A file object contains opaque file data and a directory object contains a list of name-to-object-ID mappings for the directory contents. WheelFS partitions the object ID space into 2^S slices using the first S bits of the object ID.

The configuration service maintains a *slice table* that lists, for each slice currently in use, a *replication policy* governing the slice's data placement, and a *replica list* of servers currently responsible for storing the objects in that slice. A replication policy for a slice indicates from which site it must choose the slice's primary (**.Site**), and from how many distinct sites (**.RepSites**) it must choose how many backups (**.RepLevel**). The replica list contains the current primary for a slice, and $N_{RL}-1$ backups.

Because each unique replication policy requires a unique slice identifier, the choice of S limits the maximum allowable number of replicas in a policy. In our current implementation S is fairly small (12 bits), and so to conserve slice identifiers it limits the maximum number of replicas to four.

5.2.1 Configuration service

The configuration service is a replicated state machine, and uses Paxos [25] to elect a new master whenever its membership changes. Only the master can update the slice table; it forwards updates to the other members. A WheelFS node is initially configured to know of at least one configuration service member, and contacts it to learn the full list of members and which is the master.

The configuration service exports a lock interface to WheelFS servers, inspired by Chubby [11]. Through this interface, servers can `acquire`, `renew`, and `release` locks on particular slices, or `fetch` a copy of the current slice table. A slice's lock grants the exclusive right to be a primary for that slice, and the right to specify the slice's backups and (for a new slice) its replication policy. A lock automatically expires after L seconds unless renewed. The configuration service makes no decisions about slice policy or replicas. Section 5.3 explains how WheelFS servers use the configuration service to recover after the failure of a slice's primary or backups.

Clients and servers periodically fetch and cache the slice table from the configuration service master. A client uses the slice table to identify which servers should be contacted for an object in a given slice. If a client encounters an object ID for which its cached slice table does not list a corresponding slice, the client fetches a new table. A server uses the the slice table to find other servers that store the same slice so that it can synchronize with them.

Servers try to always have at least one slice locked, to guarantee they appear in the table of currently locked slices; if the maintenance process notices that the server holds no locks, it will acquire the lock for a new slice. This allows any connected node to determine the current membership of the system by taking the union of the replica lists of all slices.

5.2.2 Placing a new file or directory

When a client creates a new file or directory, it uses the placement and durability cues specified by the application to construct an appropriate replication policy. If **.KeepTogether** is present, it sets the primary site of the policy to be the primary site of the object's parent directory's slice. Next the client checks the slice table to see if an existing slice matches the policy; if so, the client contacts the primary replica for that slice. If not, it forwards the request to a random server at the site specified by the **.Site** cue.

When a server receives a request asking it to create a new file or directory, it constructs a replication policy as above, and sets its own site to be the primary site for the policy. If it does not yet have a lock on a slice matching the policy, it generates a new, randomly-generated slice identifier and constructs a replica list for that slice, choosing from the servers listed in the slice table. The server then acquires a lock on this new slice from the configuration service, sending along the replication policy and the replica list. Once it has a lock on an appropriate slice, it generates an object ID for the new object, setting the first S bits to be the slice ID and all other bits to random values. The server returns the new ID to the client, and the client then instructs the object's parent directory's primary to add a new entry for the object. Other clients that learn about this new object ID from its entry in the parent directory can use the first S bits of the ID to find the primary for the slice and access the object.

5.2.3 Write-local policy

The default data placement policy in WheelFS is to *write locally*, *i.e.*, use a local server as the primary of a newly created file (and thus also store one copy of the contents locally). This policy works best if each client also runs a WheelFS server. The policy allows writes of large non-replicated files at the speed of the local disk, and allows such files to be written at one site and read at another with just one trip across the wide-area network.

Modifying an existing file is not always fast, because the file's primary might be far away. Applications desiring fast writes should store output in unique new files, so that the local server will be able to create a new object ID in a slice for which it is the primary. Existing software often works this way; for example, the Apache caching proxy stores a cached Web page in a unique file named after the page's URL.

An ideal default placement policy would make decisions based on server loads across the entire system; for example, if the local server is nearing its storage capacity but a neighbor server at the same site is underloaded, WheelFS might prefer writing the file to the neighbor rather than the local disk (*e.g.*, as in Porcupine [31]). Developing such a strategy is future work; for now, applications can use cues to control where data are stored.

5.3 Primary/backup replication

WheelFS uses primary/backup replication to manage replicated objects. The slice assignment designates, for each ID slice, a primary and a number of backup servers. When a client needs to read or modify an object, by default it communicates with the primary. For a file, a modification is logically an entire new version of the file contents; for a directory, a modification affects just one entry. The primary forwards each update to the backups, after which it writes the update to its disk and waits for the write to complete. The primary then waits for replies from $N_{SL}-1$ backups, indicating that those backups have also written the update to their disks. Finally, the primary replies to the client. For each object, the primary executes operations one at a time.

After being granted the lock on a slice initially, the WheelFS server must renew it periodically; if the lock expires, another server may acquire it to become the primary for the slice. Since the configuration service only grants the lock on a slice to one server at a time, WheelFS ensures that only one server will act as a primary for a slice at any given time. The slice lock time L is a compromise: short lock times lead to fast reconfiguration, while long lock times allow servers to operate despite the temporary unreachability of the configuration service.

In order to detect failure of a primary or backup, a server pings all other replicas of its slices every five minutes. If a primary decides that one of its backups is unreachable, it chooses a new replica from the same site as the old replica if possible, otherwise from a random site. The primary will transfer the slice's data to this new replica (blocking new updates), and then renew its lock on that slice along with a request to add the new replica to the replica list in place of the old one.

If a backup decides the primary is unreachable, it will attempt to acquire the lock on the slice from the configuration service; one of the backups will get the lock once the original primary's lock expires. The new primary checks with the backups to make sure that it didn't miss any object updates (*e.g.*, because $N_{SL}<N_{RL}$ during a recent update, and thus not all backups are guaranteed to have committed that update).

A primary's maintenance process periodically checks that the replicas associated with each slice match the slice's policy; if not, it will attempt to recruit new replicas at the appropriate sites. If the current primary wishes to recruit a new primary at the slice's correct primary site (*e.g.*, a server that had originally been the slice's primary but crashed and rejoined), it will release its lock on the slice, and directly contact the chosen server, instructing it to acquire the lock for the slice.

5.4 Consistency

By default, WheelFS provides close-to-open consistency: if one application instance writes a file and waits for `close()` to return, and then a second application instance `open()`s and reads the file, the second application will see the effects of the first application's writes. The reason WheelFS provides close-to-open consistency by default is that many applications expect it.

The WheelFS client has a write-through cache for file blocks, for positive and negative directory entries (enabling faster pathname lookups), and for directory and file meta-data. A client must acquire an *object lease* from an object's primary before it uses cached meta-data. Before the primary executes any update to an object, it must invalidate all leases or wait for them to expire. This step may be time-consuming if many clients hold leases on an object.

Clients buffer file writes locally to improve performance. When an application calls `close()`, the client sends all outstanding writes to the primary, and waits for the primary to acknowledge them before allowing `close()` to return. Servers maintain a version number for each file object, which they increment after each `close()` and after each change to the object's meta-data.

When an application `open()`s a file and then reads it, the WheelFS client must decide whether the cached copy of the file (if any) is still valid. The client uses cached file data if the object version number of the cached data is the same as the object's current version number. If the client has an unexpired object lease for the object's meta-data, it can use its cached meta-data for the object to find the current version number. Otherwise it must contact the primary to ask for a new lease, and for current meta-data. If the version number of the cached data is not current, the client fetches new file data from the primary.

By default, WheelFS provides similar consistency for directory operations: after the return of an application system call that modifies a directory (links or unlinks a file or subdirectory), applications on other clients are guaranteed to see the modification. WheelFS clients implement this consistency by sending directory updates to the directory object's primary, and by ensuring via lease or explicit check with the primary that cached directory contents are up to date. Cross-directory rename operations in WheelFS are not atomic with respect to failures. If a crash occurs at the wrong moment, the result may be a link to the moved file in both the source and destination directories.

The downside to close-to-open consistency is that if a primary is not reachable, all operations that consult the primary will delay until it revives or a new primary takes over. The **.EventualConsistency** cue allows WheelFS to avoid these delays by using potentially stale data from backups or local caches when the primary does not respond, and by sending updates to backups. This can result in inconsistent replicas, which the maintenance process resolves in the manner described in Section 4.5, leading eventually to identical images at all replicas. Without the **.EventualConsistency** cue, a server will reject operations on objects for which it is not the primary.

Applications can specify timeouts on a per-object basis using the **.MaxTime=T** cue. This adds a timeout of T ms to every operation performed at a server. Without **.EventualConsistency**, a client will return a failure to the application if the primary does not respond within T ms; with **.EventualConsistency**, clients contact backup servers once the timeout occurs. In future work we hope to explore how to best divide this timeout when a single file system operation might involve contacting several servers (*e.g.*, a create requires talking to the parent directory's primary and the new object's primary, which could differ).

5.5 Large reads

If the application specifies **.WholeFile** when reading a file, the client will pre-fetch the entire file into its cache. If the application uses **.WholeFile** when reading directory contents, WheelFS will pre-fetch the meta-data for all of the directory's entries, so that subsequent lookups can be serviced from the cache.

To implement the **.Hotspot** cue, a file's primary maintains a soft-state list of clients that have recently cached blocks of the file, including which blocks they have cached. A client that reads a file with **.Hotspot** asks the server for entries from the list that are near the client; the server chooses the entries using Vivaldi coordinates [15]. The client uses the list to fetch each block from a nearby cached copy, and informs the primary of successfully fetched blocks.

If the application reads a file with both **.WholeFile** and **.Hotspot**, the client will issue block fetches in parallel to multiple other clients. It pre-fetches blocks in a random order so that clients can use each others' caches even if they start reading at the same time [6].

5.6 Security

WheelFS enforces three main security properties. First, a given WheelFS deployment ensures that only authorized hosts participate as servers. Second, WheelFS ensures that requests come only from users authorized to use the deployment. Third, WheelFS enforces user-based permissions on requests from clients. WheelFS assumes that authorized servers behave correctly. A misbehaving client can act as any user that has authenticated themselves to WheelFS from that client, but can only do things for which those users have permission.

All communication takes place through authenticated SSH channels. Each authorized server has a public/private key pair which it uses to prove its identity. A central administrator maintains a list of all legitimate server public keys in a deployment, and distributes that list to every server and client. Servers only exchange inter-server traffic with hosts authenticated with a key on the list, and clients only send requests to (and use responses from) authentic servers.

Each authorized user has a public/private key pair; WheelFS uses SSH's existing key management support. Before a user can use WheelFS on a particular client, the user must reveal his or her private key to the client. The list of authorized user public keys is distributed to all servers and clients as a file in WheelFS. A server accepts only client connections signed by an authorized user key. A server checks that the authenticated user for a request has appropriate permissions for the file or directory being manipulated—each object has an associated access control list in its meta-data. A client dedicated to a particular distributed application stores its "user" private key on its local disk.

Clients check data received from other clients against server-supplied SHA-256 checksums to prevent clients from tricking each other into accepting unauthorized modifications. A client will not supply data from its cache to another client whose authorized user does not have read permissions.

There are several planned improvements to this security setup. One is an automated mechanism for propagating changes to the set of server public keys, which currently need to be distributed manually. Another is to allow the use of SSH Agent forwarding to allow users to connect securely without storing private keys on client hosts, which would increase the security of highly privileged keys in the case where a client is compromised.

6 Applications

WheelFS is designed to help the construction of wide-area distributed applications, by shouldering a significant part of the burden of managing fault tolerance, consistency, and sharing of data among sites. This section evaluates how well WheelFS fulfills that goal by describing four applications that have been built using it.

All-Pairs-Pings. All-Pairs-Pings [37] monitors the network delays among a set of hosts. Figure 2 shows a simple version of All-Pairs-Pings built from a shell script and WheelFS, to be invoked by each host's `cron` every few minutes. The script pings the other hosts and puts the results in a file whose name contains the local host name

```
FILE=`date +%s`.`hostname`.dat
D=/wfs/ping
BIN=$D/bin/.EventualConsistency/
.MaxTime=5000/.HotSpot/.WholeFile
DATA=$D/.EventualConsistency/dat
mkdir -p $DATA/`hostname`
cd $DATA/`hostname`
xargs -n1 $BIN/ping -c 10 <
 $D/nodes > /tmp/$FILE
cp /tmp/$FILE $FILE
rm /tmp/$FILE
if [ `hostname` = "node1" ]; then
 mkdir -p $D/res
 $BIN/process * > $D/res/`date +%s`.o
fi
```

Figure 2: A shell script implementation of All-Pairs-Pings using WheelFS.

and the current time. After each set of pings, a coordinator host ("node1") reads all the files, creates a summary using the program `process` (not shown), and writes the output to a results directory.

This example shows that WheelFS can help keep simple distributed tasks easy to write, while protecting the tasks from failures of remote nodes. WheelFS stores each host's output on the host's own WheelFS server, so that hosts can record ping output even when the network is broken. WheelFS automatically collects data files from hosts that reappear after a period of separation. Finally, WheelFS provides each host with the required binaries and scripts and the latest host list file. Use of WheelFS in this script eliminates much of the complexity of a previous All-Pairs-Pings program, which explicitly dealt with moving files among nodes and coping with timeouts.

Distributed Web cache. This application consists of hosts running Apache 2.2.4 caching proxies (`mod_disk_cache`). The Apache configuration file places the cache file directory on WheelFS:

`/wfs/`**.EventualConsistency/.MaxTime=1000/**
.Hotspot`/cache/`

When the Apache proxy can't find a page in the cache directory on WheelFS, it fetches the page from the origin Web server and writes a copy in the WheelFS directory, as well as serving it to the requesting browser. Other cache nodes will then be able to read the page from WheelFS, reducing the load on the origin Web server. The **.Hotspot** cue copes with popular files, directing the WheelFS clients to fetch from each others' caches to increase total throughput. The **.EventualConsistency** cue allows clients to create and read files even if they cannot contact the primary server. The **.MaxTime** cue instructs WheelFS to return an error if it cannot find a file quickly, causing Apache to fetch the page from the origin Web server. If WheelFS returns an expired version of the file, Apache will notice by checking the HTTP header in the cache file, and it will contact the origin Web server for a fresh copy.

Although this distributed Web cache implementation is fully functional, it does lack features present in other similar systems. For example, CoralCDN uses a hierarchy of caches to avoid overloading any single tracker node when a file is popular.

Mail service. The goal of Wheemail, our WheelFS-based mail service, is to provide high throughput by spreading the work over many sites, and high availability by replicating messages on multiple sites. Wheemail provides SMTP and IMAP service from a set of nodes at these sites. Any node at any site can accept a message via SMTP for any user; in most circumstances a user can fetch mail from the IMAP server on any node.

Each node runs an unmodified sendmail process to accept incoming mail. Sendmail stores each user's messages in a WheelFS directory, one message per file. The separate files help avoid conflicts from concurrent message arrivals. A user's directory has this path:

`/wfs/mail/`**.EventualConsistency**`/`**.Site=X**`/`
.KeepTogether`/`**.RepSites=2**`/`*user*`/Mail/`

Each node runs a Dovecot IMAP server [17] to serve users their messages. A user retrieves mail via a nearby node using a locality-preserving DNS service [20].

The **.EventualConsistency** cue allows a user to read mail via backup servers when the primary for the user's directory is unreachable, and allows incoming mail to be stored even if primary and all backups are down. The **.Site=X** cue indicates that a user's messages should be stored at site X, chosen to be close to the user's usual location to reduce network delays. The **.KeepTogether** cue causes all of a user's messages to be stored on a single replica set, reducing latency for listing the user's messages [31]. Wheemail uses the default replication level of three but uses **.RepSites=2** to keep at least one off-site replica of each mail. To avoid unnecessary replication, Dovecot uses **.RepLevel=1** for much of its internal data.

Wheemail has goals similar to those of Porcupine [31], namely, to provide scalable email storage and retrieval with high availability. Unlike Porcupine, Wheemail runs on a set of wide-area data centers. Replicating emails over multiple sites increases the service's availability when a single site goes down. Porcupine consists of custom-built storage and retrieval components. In contrast, the use of a wide-area file system in Wheemail allows it to reuse existing software like sendmail and Dovecot. Both Porcupine and Wheemail use eventual consistency to increase availability, but Porcupine has a better reconciliation policy as its "deletion record" prevents deleted emails from reappearing.

File Distribution. A set of many WheelFS clients can cooperate to fetch a file efficiently using the large read cues:

`/wfs/`**.WholeFile**`/`**.Hotspot**`/largefile`

Efficient file distribution may be particularly useful for binaries in wide-area experiments, in the spirit of Shark [6] and CoBlitz [29]. Like Shark, WheelFS uses cooperative caching to reduce load on the file server. Shark further reduces the load on the file server by using a distributed index to keep track of cached copies, whereas WheelFS relies on the primary server to track copies. Unlike WheelFS or Shark, CoBlitz is a CDN, so files cannot be directly accessed through a mounted file system. CoBlitz caches and shares data between CDN nodes rather than between clients.

7 Implementation

The WheelFS prototype consists of 19,000 lines of C++ code, using pthreads and STL. In addition, the implementation uses a new RPC library (3,800 lines) that implements Vivaldi network coordinates [15].

The WheelFS client uses FUSE's "low level" interface to get access to FUSE identifiers, which it translates into WheelFS-wide unique object IDs. The WheelFS cache layer in the client buffers writes in memory and caches file blocks in memory and on disk.

Permissions, access control, and secure SSH connections are implemented. Distribution of public keys through WheelFS is not yet implemented.

8 Evaluation

This section demonstrates the following points about the performance and behavior of WheelFS:

- For some storage workloads common in distributed applications, WheelFS offers more scalable performance than an implementation of NFSv4.
- WheelFS achieves reasonable performance under a range of real applications running on a large, wide-area testbed, as well as on a controlled testbed using an emulated network.
- WheelFS provides high performance despite network and server failures for applications that indicate via cues that they can tolerate relaxed consistency.
- WheelFS offers data placement options that allow applications to place data near the users of that data, without the need for special application logic.
- WheelFS offers client-to-client read options that help counteract wide-area bandwidth constraints.
- WheelFS offers an interface on which it is quick and easy to build real distributed applications.

8.1 Experimental setup

All scenarios use WheelFS configured with 64 KB blocks, a 100 MB in-memory client LRU block cache supplemented by an unlimited on-disk cache, one minute object leases, a lock time of $L = 2$ minutes, 12-bit slice IDs, 32-bit object IDs, and a default replication level of three (the responsible server plus two replicas), unless stated otherwise. Communication takes place over plain TCP, not SSH, connections. Each WheelFS node runs both a storage server and a client process. The configuration service runs on five nodes distributed across three wide-area sites.

We evaluate our WheelFS prototype on two testbeds: PlanetLab [7] and Emulab [42]. For PlanetLab experiments, we use up to 250 nodes geographically spread across the world at more than 140 sites (we determine the site of a node based on the domain portion of its hostname). These nodes are shared with other researchers and their disks, CPU, and bandwidth are often heavily loaded, showing how WheelFS performs in the wild. These nodes run a Linux 2.6 kernel and FUSE 2.7.3. We run the configuration service on a private set of nodes running at MIT, NYU, and Stanford, to ensure that the replicated state machine can log operations to disk and respond to requests quickly (`fsync()`s on PlanetLab nodes can sometimes take tens of seconds).

For more control over the network topology and host load, we also run experiments on the Emulab [42] testbed. Each Emulab host runs a standard Fedora Core 6 Linux 2.6.22 kernel and FUSE version 2.6.5, and has a 3 GHz CPU. We use a WAN topology consisting of 5 LAN clusters of 3 nodes each. Each LAN cluster has 100 Mbps, sub-millisecond links between each node. Clusters connect to the wide-area network via a single bottleneck link of 6 Mbps, with 100 ms RTTs between clusters.

8.2 Scalability

We first evaluate the scalability of WheelFS on a microbenchmark representing a workload common to distributed applications: many nodes reading data written by other nodes in the system. For example, nodes running a distributed Web cache over a shared storage layer would be reading and serving pages written by other nodes. In this microbenchmark, N clients mount a shared file system containing N directories, either using NFSv4 or WheelFS. Each directory contains ten 1 MB files. The clients are PlanetLab nodes picked at random from the set of nodes that support both mounting both FUSE and NFS file systems. This set spans a variety of nodes distributed across the world, from nodes at well-connected educational institutions to nodes behind limited-upload DSL lines. Each client reads ten random files from the file

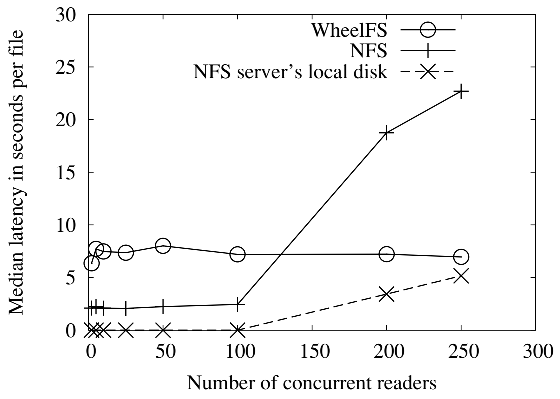


Figure 3: The median time for a set of PlanetLab clients to read a 1 MB file, as a function of the number of concurrently reading nodes. Also plots the median time for a set of local processes to read 1 MB files from the NFS server's local disk through `ext3`.

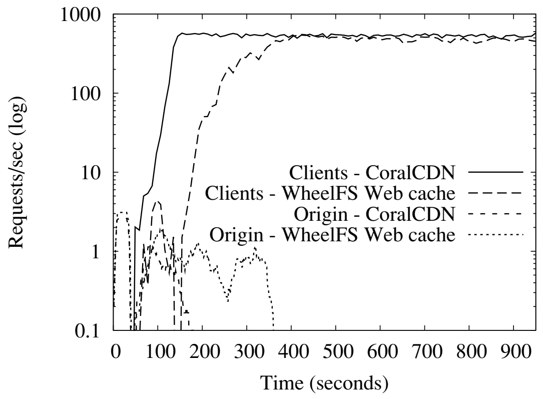


Figure 4: The aggregate client service rate and origin server load for both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

system in sequence, and measures the read latency. The clients all do this at the same time.

For WheelFS, each client also acts as a server, and is the primary for one directory and all files within that directory. WheelFS clients do not read files for which they are the primary, and no file is ever read twice by the same node. The NFS server is a machine at MIT running Debian's nfs-kernel-server version 1.0.10-6 using the default configuration, with a 2.8 GHz CPU and a SCSI hard drive.

Figure 3 shows the median time to read a file as N varies. For WheelFS, a very small fraction of reads fail because not all pairs of PlanetLab nodes can communicate; these reads are not included in the graph. Each point on the graph is the median of the results of at least one hundred nodes (*e.g.*, a point showing the latency for five concurrent nodes represents the median reported by all nodes across twenty different trials).

Though the NFS server achieves lower latencies when there are few concurrent clients, its latency rises sharply as the number of clients grows. This rise occurs when there are enough clients, and thus files, that the files do not fit in the server's 1GB file cache. Figure 3 also shows results for N concurrent processes on the NFS server, accessing the `ext3` file system directly, showing a similar latency increase after 100 clients. WheelFS latencies are not affected by the number of concurrent clients, since WheelFS spreads files and thus the load across many servers.

8.3 Distributed Web Cache

Performance under normal conditions. These experiments compare the performance of CoralCDN and the WheelFS distributed Web cache (as described in Section 6, except with **.MaxTime=2000** to adapt to PlanetLab's characteristics). The main goal of the cache is to reduce load on target Web servers via caching, and secondarily to provide client browsers with reduced latency and increased availability.

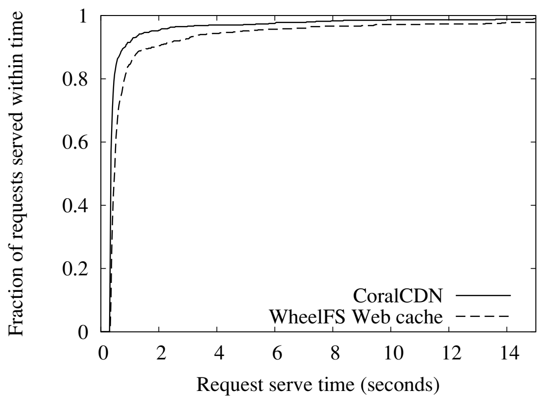


Figure 5: The CDF for the client request latencies of both CoralCDN and the WheelFS-based Web cache, running on PlanetLab.

These experiments use forty nodes from PlanetLab hosted at `.edu` domains, spread across the continental United States. A Web server, located at NYU behind an emulated slow link (shaped using Click [24] to be 400 Kbps and have a 100 ms delay), serves 100 unique 41KB Web pages. Each of the 40 nodes runs a Web proxy. For each proxy node there is another node less than 10 ms away that runs a simulated browser as a Web client. Each Web client requests a sequence of randomly selected pages from the NYU Web server. This experiment, inspired by one in the CoralCDN paper [19], models a flash crowd where a set of files on an under-provisioned server become popular very quickly.

Figures 4 and 5 show the results of these experiments. Figure 4 plots both the total rate at which the proxies send requests to the origin server and the total rate at which the proxies serve Web client requests (the y-axis is a log scale). WheelFS takes about twice as much time as Coral-

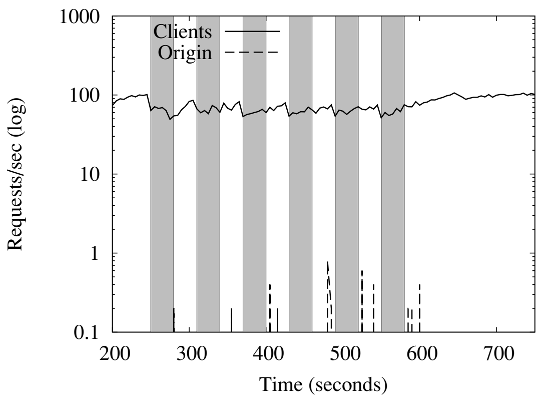


Figure 6: The WheelFS-based Web cache running on Emulab with failures, using the **.EventualConsistency** cue. Gray regions indicate the duration of a failure.

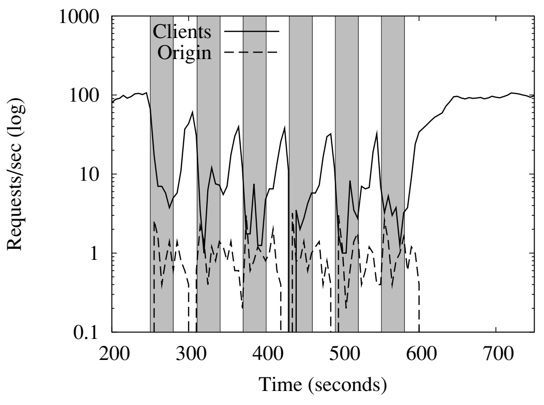


Figure 7: The WheelFS-based Web cache running on Emulab with failures, with close-to-open consistency. Gray regions indicate the duration of a failure.

CDN to reduce the origin load to zero; both reach similar sustained aggregate Web client service rates. Figure 5 plots the cumulative distribution function (CDF) of the request latencies seen by the Web clients. WheelFS has somewhat higher latencies than CoralCDN.

CoralCDN has higher performance because it incorporates many application-specific optimizations, whereas the WheelFS-based cache is built from more general-purpose components. For instance, a CoralCDN proxy pre-declares its intent to download a page, preventing other nodes from downloading the same page; Apache, running on WheelFS, has no such mechanism, so several nodes may download the same page before Apache caches the data in WheelFS. Similar optimizations could be implemented in Apache.

Performance under failures. Wide-area network problems that prevent WheelFS from contacting storage nodes should not translate into long delays; if a proxy cannot quickly fetch a cached page from WheelFS, it should ask the origin Web server. As discussed in Section 6, the cues **.EventualConsistency** and **.MaxTime=1000** yield this behavior, causing `open()` to either find a copy of the desired file or fail in one second. Apache fetches from the origin Web server if the `open()` fails.

To test how failures affect WheelFS application performance, we ran a distributed Web cache experiment on the Emulab topology in Section 8.1, where we could control the network's failure behavior. At each of the five sites there are three WheelFS Web proxies. Each site also has a Web client, which connects to the Web proxies at the same site using a 10 Mbps, 20 ms link, issuing five requests at a time. The origin Web server runs behind a 400 Kbps link, with 150 ms RTTs to the Web proxies.

Figures 6 and 7 compare failure performance of WheelFS with the above cues to failure performance of close-to-open consistency with 1-second timeouts (**.MaxTime=1000**). The y-axes of these graphs are log-scale. Each minute one wide-area link connecting an entire site to the rest of the network fails for thirty seconds and then revives. This failure period is not long enough to cause servers at the failed site to lose their slice locks. Web clients maintain connectivity to the proxies at their local site during failures. For comparison, Figure 8 shows WheelFS's performance on this topology when there are no failures.

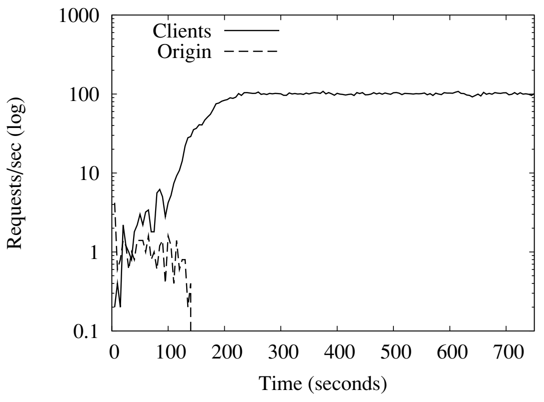


Figure 8: The aggregate client service rate and origin server load for the WheelFS-based Web cache, running on Emulab, without failures.

When a Web client requests a page from a proxy, the proxy must find two pieces of information in order to find a copy of the page (if any) in WheelFS: the object ID to which the page's file name resolves, and the file content for that object ID. The directory information and the file content can be on different WheelFS servers. For each kind of information, if the proxy's WheelFS client has cached the information and has a valid lease, the WheelFS

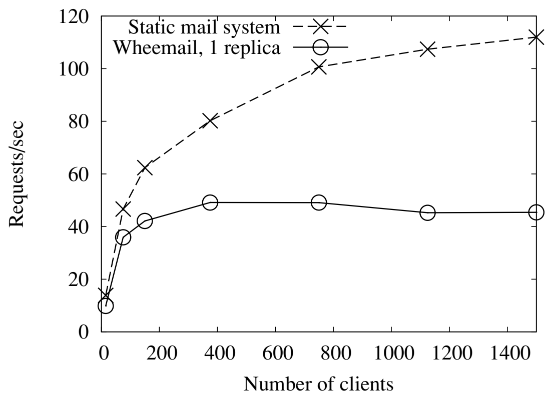


Figure 9: The throughput of Wheemail compared with the static system, on the Emulab testbed.

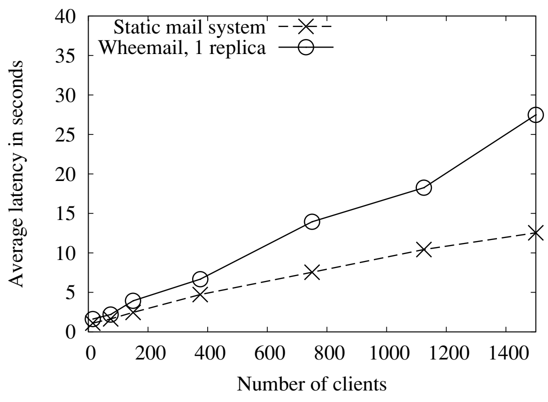


Figure 10: The average latencies of individual SMTP requests, for both Wheemail and the static system, on Emulab.

client need not contact a server. If the WheelFS client doesn't have information with a valid lease, and is using eventual consistency, it tries to fetch the information from the primary; if that fails (after a one-second timeout), the WheelFS client will try fetch from a backup; if that fails, the client will use locally cached information (if any) despite an expired lease; otherwise the `open()` fails and the proxy fetches the page from the origin server. If a WheelFS client using close-to-open consistency does not have cached data with a valid lease, it first tries to contact the primary; if that fails (after timeout), the proxy must fetch the page from the origin Web server.

Figure 6 shows the performance of the WheelFS Web cache with eventual consistency. The graph shows a period of time after the initial cache population. The gray regions indicate when a failure is present. Throughput falls as WheelFS clients encounter timeouts to servers at the failed site, though the service rate remains near 100 requests/sec. The small load spikes at the origin server after a failure reflect requests queued up in the network by the failed site while it is partitioned. Figure 7 shows that with close-to-open consistency, throughput falls significantly during failures, and hits to the origin server increase greatly. This shows that a cooperative Web cache, which does not require strong consistency, can use WheelFS's semantic cues to perform well under wide-area conditions.

8.4 Mail

The Wheemail system described in Section 6 has a number of valuable properties such as the ability to serve and accept a user's mail from any of multiple sites. This section explores the performance cost of those properties by comparing to a traditional mail system that lacks those properties.

IMAP and SMTP are stressful file system benchmarks. For example, an IMAP server reading a Maildir-formatted inbox and finding no new messages generates over 600 FUSE operations. These primarily consist of lookups on directory and file names, but also include more than 30 directory operations (creates/links/unlinks/renames), more than 30 small writes, and a few small reads. A single SMTP mail delivery generates over 60 FUSE operations, again consisting mostly of lookups.

In this experiment we use the Emulab network topology described in Section 8.1 with 5 sites. Each site has a 1 Mbps link to a wide-area network that connects all the sites. Each site has three server nodes that each run a WheelFS server, a WheelFS client, an SMTP server, and an IMAP server. Each site also has three client nodes, each of which runs multiple load-generation threads. A load-generation thread produces a sequence of SMTP and IMAP requests as fast as it can. 90% of requests are SMTP and 10% are IMAP. User mailbox directories are randomly and evenly distributed across sites. The load-generation threads pick users and message sizes with probabilities from distributions derived from SMTP and IMAP logs of servers at NYU; there are 47699 users, and the average message size is 6.9 KB. We measure throughput in requests/second, with an increasing number of concurrent client threads.

When measuring WheelFS, a load-generating thread at a given site only generates requests from users whose mail is stored at that site (the user's "home" site), and connects only to IMAP and SMTP servers at the local site. Thus an IMAP request can be handled entirely within a home site, and does not generate any wide-area traffic (during this experiment, each node has cached directory lookup information for the mailboxes of all users at its site). A load-generating thread generates mail to random users, connecting to a SMTP server at the same site; that server writes the messages to the user's directory in WheelFS, which is likely to reside at a different site. In this experiment, user mailbox directories are not replicated.

We compare against a "static" mail system in which users are partitioned over the 15 server nodes, with the

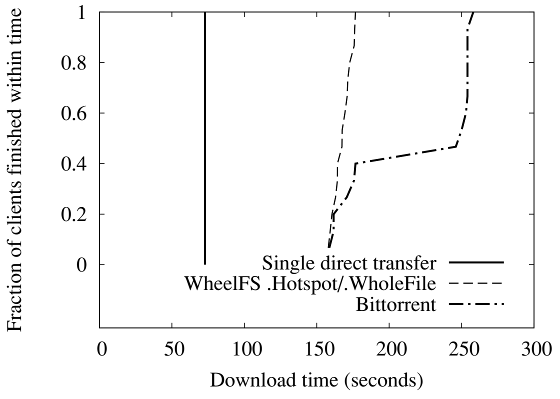


Figure 11: CDF of client download times of a 50 MB file using BitTorrent and WheelFS with the **.Hotspot** and **.WholeFile** cues, running on Emulab. Also shown is the time for a single client to download 50 MB directly using `ttcp`.

SMTP and IMAP servers on each server node storing mail on a local disk file system. The load-generator threads at each site only generate IMAP requests for users at the same site, so IMAP traffic never crosses the wide area network. When sending mail, a load-generating client picks a random recipient, looks up that user's home server, and makes an SMTP connection to that server, often across the wide-area network.

Figure 9 shows the aggregate number of requests served by the entire system per second. The static system can sustain 112 requests per second. Each site's 1 Mbps wide-area link is the bottleneck: since 90% of the requests are SMTP (message with an average size 6.85 KB), and 80% of those go over the wide area, the system as a whole is sending 4.3 Mbps across a total link capacity of 5 Mbps, with the remaining wide-area bandwidth being used by the SMTP and TCP protocols.

Wheemail achieves up to 50 requests per second, 45% of the static system's performance. Again the 1 Mbps WAN links are the bottleneck: for each SMTP request, WheelFS must send 11 wide-area RPCs to the target user's mailbox site, adding an overhead of about 40% to the size of the mail message, in addition to the continuous background traffic generated by the maintenance process, slice lock renewal, Vivaldi coordinate measurement, and occasional lease invalidations.

Figure 10 shows the average latencies of individual SMTP requests for Wheemail and the static system, as the number of clients varies. Wheemail's latencies are higher than those of the static system by nearly 60%, attributable to traffic overhead generated by WheelFS.

Though the static system outperforms Wheemail for this benchmark, Wheemail provides many desirable properties that the static system lacks. Wheemail transparently redirects a receiver's mail to its home site, regardless of where the SMTP connection occurred; additional storage can be added to the system without major manual reconfiguration; and Wheemail can be configured to offer tolerance to site failures, all without any special logic having to be built into the mail system itself.

Application	**LoC**	**Reuses**
CDN	1	Apache+mod_disk_cache
Mail service	4	Sendmail+Procmail+Dovecot
File distribution	N/A	Built-in to WheelFS
All-Pairs-Pings	13	N/A

Table 3: Number of lines of changes to adapt applications to use WheelFS.

8.5 File distribution

Our file distribution experiments use a WheelFS network consisting of 15 nodes, spread over five LAN clusters connected by the emulated wide-area network described in Section 8.1. Nodes attempt to read a 50 MB file simultaneously (initially located at an originating, 16^{th} WheelFS node that is in its own cluster) using the **.Hotspot** and **.WholeFile** cues. For comparison, we also fetch the file using BitTorrent [14] (the Fedora Core distribution of version 4.4.0-5). We configured BitTorrent to allow unlimited uploads and to use 64 KB blocks like WheelFS (in this test, BitTorrent performs strictly worse with its usual default of 256 KB blocks).

Figure 11 shows the CDF of the download times, under WheelFS and BitTorrent, as well as the time for a single direct transfer of 50 MB between two wide-area nodes (73 seconds). WheelFS's median download time is 168 seconds, showing that WheelFS's implementation of cooperative reading is better than BitTorrent's: BitTorrent clients have a median download time of 249 seconds. The improvement is due to WheelFS clients fetching from nearby nodes according to Vivaldi coordinates; BitTorrent does not use a locality mechanism. Of course, both solutions offer far better download times than 15 simultaneous direct transfers from a single node, which in this setup has a median download time of 892 seconds.

8.6 Implementation ease

Table 3 shows the number of new or modified lines of code (LoC) we had to write for each application (excluding WheelFS itself). Table 3 demonstrates that developers can benefit from a POSIX file system interface and cues to build wide-area applications with ease.

9 Related Work

There is a humbling amount of past work on distributed file systems, wide-area storage in general and the tradeoffs of availability and consistency. PRACTI [8] is a recently-proposed framework for building storage systems with arbitrary consistency guarantees (as in TACT [43]). Like PRACTI, WheelFS maintains flexibility by separating policies from mechanisms, but it has a different goal. While PRACTI and its recent extension PADS [9] are designed to simplify the development of new storage or file systems, WheelFS itself is a flexible file system designed to simplify the construction of distributed applications. As a result, WheelFS's cues are motivated by the specific needs of applications (such as the **.Site** cue) while PRACTI's primitives aim at covering the entire spectrum of design tradeoffs (*e.g.*, strong consistency for operations spanning multiple data objects, which WheelFS does not support).

Most distributed file systems are designed to support a workload generated by desktop users (*e.g.*, NFS [33], AFS [34], Farsite [2], xFS [5], Frangipani [12], Ivy [27]). They usually provide a consistent view of data, while sometimes allowing for disconnected operation (*e.g.*, Coda [35] and BlueFS [28]). Cluster file systems such as GFS [22] and Ceph [41] have demonstrated that a distributed file system can dramatically simplify the construction of distributed applications within a large cluster with good performance. Extending the success of cluster file systems to the wide-area environment continues to be difficult due to the tradeoffs necessary to combat wide-area network challenges. Similarly, Sinfonia [3] offers highly-scalable cluster storage for infrastructure applications, and allows some degree of inter-object consistency via lightweight transactions. However, it targets storage at the level of individual pieces of data, rather than files and directories like WheelFS, and uses protocols like two-phase commit that are costly in the wide area. Shark [6] shares with WheelFS the goal of allowing client-to-client data sharing, though its use of a centralized server limits its scalability for applications in which nodes often operate on independent data.

Successful wide-area storage systems generally exploit application-specific knowledge to make decisions about tradeoffs in the wide-area environment. As a result, many wide-area applications include their own storage layers [4, 14, 19, 31] or adapt an existing system [29, 40]. Unfortunately, most existing storage systems, even more general ones like OceanStore/Pond [30] or S3 [1], are only suitable for a limited range of applications and still require a large amount of code to use. DHTs are a popular form of general wide-area storage, but, while DHTs all offer a similar interface, they differ widely in implementation. For example, UsenetDHT [36] and CoralCDN [19] both use a DHT, but their DHTs differ in many details and are not interchangeable.

Some wide-area storage systems offer configuration options in order to make them suitable for a larger range of applications. Amazon's Dynamo [16] works across multiple data centers and provides developers with two knobs: the number of replicas to read or to write, in order to control durability, availability and consistency tradeoffs. By contrast, WheelFS's cues are at a higher level (*e.g.*, eventual consistency versus close-to-open consistency). Total Recall [10] offers a per-object flexible storage API and uses a primary/backup architecture like WheelFS, but assumes no network partitions, focuses mostly on availability controls, and targets a more dynamic environment. Bayou [39] and Pangaea [32] provide eventual consistency by default while the latter also allows the use of a "red button" to wait for the acknowledgment of updates from all replicas explicitly. Like Pangaea and Dynamo, WheelFS provides flexible consistency tradeoffs. Additionally, WheelFS also provides controls in other categories (such as data placement, large reads) to suit the needs of a variety of applications.

10 Conclusion

Applications that distribute data across multiple sites have varied consistency, durability, and availability needs. A shared storage system able to meet this diverse set of needs would ideally provide applications a flexible and practical interface, and handle applications' storage needs without sacrificing much performance when compared to a specialized solution. This paper describes WheelFS, a wide-area storage system with a traditional POSIX interface augmented by cues that allow distributed applications to control consistency and fault-tolerance tradeoffs.

WheelFS offers a small set of cues in four categories (placement, durability, consistency, and large reads), which we have found to work well for many common distributed workloads. We have used a WheelFS prototype as a building block in a variety of distributed applications, and evaluation results show that it meets the needs of these applications while permitting significant code reuse of their existing, non-distributed counterparts. We hope to make an implementation of WheelFS available to developers in the near future.

Acknowledgments

This research was supported by NSF grant CNS-0720644 and by Microsoft Research Asia and Tsinghua University. We thank the many people who have improved this work through discussion, testbed support, reviews, and skepticism: our shepherd Jeff Dean, the anonymous reviewers, the members of the PDOS research group at MIT CSAIL, Sapan Bhatia, Russ Cox, Frank Dabek, Marc Fiuczynski, Michael Freedman, Jeremy Kepner, Jay Lepreau, Jinyuan Li, David Mazières, Michael Puskar and NYU ITS, Emil Sit, Michael Walfish, Daniel Hokka Zakrisson, and Nickolai Zeldovich.

References

[1] Amazon Simple Storage System. `http://aws.amazon.com/s3/`.

[2] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken, R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th OSDI* (Dec. 2002).

[3] Aguilera, M. K., Merchant, A., Shah, M., Veitch, A., and Karamanolis, C. Sinfonia: A new paradigm for building scalable distributed systems. In *Proceedings of the 21st SOSP* (Oct. 2007).

[4] Allcock, W., Bresnahan, J., Kettimuthu, R., Link, M., Dumitrescu, C., Raicu, I., and Foster, I. The Globus striped GridFTP framework and server. In *Proceedings of the 2005 Super Computing* (Nov. 2005).

[5] Anderson, T. E., Dahlin, M. D., Neefe, J. M., Patterson, D. A., Roselli, D. S., and Wang, R. Y. Serverless network file systems. In *Proceedings of the 15th SOSP* (Dec. 1995).

[6] Annapureddy, S., Freedman, M. J., and Mazières, D. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd NSDI* (May 2005).

[7] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. Operating systems support for planetary-scale network services. In *Proceedings of the 1st NSDI* (Mar. 2004).

[8] Belaramani, N., Dahlin, M., Gao, L., Nayate, A., Venkataramani, A., Yalagandula, P., and Zheng, J. PRACTI replication. In *Proceedings of the 3rd NSDI* (2006).

[9] Belaramani, N., Zheng, J., Nayate, A., Soulé, R., Dahlin, M., and Grimm, R. PADS: A policy architecture for building distributed storage systems. In *Proceedings of the 6th NSDI* (Apr. 2009).

[10] Bhagwan, R., Tati, K., Cheng, Y.-C., Savage, S., and Voelker, G. M. Total Recall: System support for automated availability management. In *Proceedings of the 1st NSDI* (Mar. 2004).

[11] Burrows, M. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th OSDI* (Nov. 2006).

[12] C. Thekkath, T. Mann, E. L. Frangipani: A scalable distributed file system. In *Proceedings of the 16th SOSP*.

[13] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th OSDI* (Nov. 2006).

[14] Cohen, B. Incentives build robustness in BitTorrent. In *Proceedings of the Workshop on Economics of Peer-to-Peer Systems* (June 2003).

[15] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. A decentralized network coordinate system. In *Proceedings of the 2004 SIGCOMM* (2004).

[16] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st SOSP* (Oct. 2007).

[17] Dovecot IMAP server. `http://www.dovecot.org/`.

[18] Fekete, A., Gupta, D., Luchangco, V., Lynch, N., and Schvartsman, A. Eventually-serializable data services. *Theoretical Computer Science* (June 1999).

[19] Freedman, M. J., Freudenthal, E., and Mazières, D. Democratizing content publication with Coral. In *Proceedings of the 1st NSDI* (Mar. 2004).

[20] Freedman, M. J., Lakshminarayanan, K., and Mazières, D. OASIS: Anycast for any service. In *Proceedings of the 3rd NSDI* (May 2006).

[21] Filesystem in user space. `http://fuse.sourceforge.net/`.

[22] Ghemawat, S., Gobioff, H., and Leung, S.-T. The Google file system. In *Proceedings of the 19th SOSP* (Dec. 2003).

[23] Gilbert, S., and Lynch, N. Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services. In *ACM SIGACT News* (June 2002), vol. 33.

[24] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Trans. on Computer Systems* (Aug. 2000).

[25] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (1998), 133–169.

[26] Li, J., Krohn, M., Mazières, D., and Shasha, D. Secure Untrusted data Repository (SUNDR). In *Proceedings of the 6th OSDI* (Dec. 2004).

[27] Muthitacharoen, A., Morris, R., Gil, T., and Chen, B. Ivy: A read/write peer-to-peer file system. In *Proceedings of the 5th OSDI* (2002).

[28] Nightingale, E. B., and Flinn, J. Energy-efficiency and storage flexibility in the Blue File System. In *Proceedings of the 6th OSDI* (Dec. 2004).

[29] Park, K., and Pai, V. S. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd NSDI* (May 2006).

[30] Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., and Kubiatowicz, J. Pond: The OceanStore prototype. In *Proceedings of the 2nd FAST* (Mar. 2003).

[31] Saito, Y., Bershad, B., and Levy, H. Manageability, availability and performance in Porcupine: A highly scalable internet mail service. *ACM Transactions of Computer Systems* (2000).

[32] Saito, Y., Karamonolis, C., Karlsson, M., and Mahalingam, M. Taming aggressive replication in the Pangaea wide-area file system. In *Proceedings of the 5th OSDI* (2002).

[33] Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. Design and implementation of the Sun Network Filesystem. In *Proceedings of the Summer 1985 USENIX* (June 1985).

[34] Satyanarayanan, M., Howard, J., Nichols, D., Sidebotham, R., Spector, A., and West, M. The ITC distributed file system: Principles and design. In *Proceedings of the 10th SOSP* (1985).

[35] Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. on Comp. 4*, 39 (Apr 1990), 447–459.

[36] Sit, E., Morris, R., and Kaashoek, M. F. UsenetDHT: A low-overhead design for Usenet. In *Usenix NSDI* (2008).

[37] Stribling, J. PlanetLab All-Pairs-Pings. `http://pdos.csail.mit.edu/~strib/pl_app/`.

[38] Stribling, J., Sit, E., Kaashoek, M. F., Li, J., and Morris, R. Don't give up on distributed file systems. In *Proceedings of the 6th IPTPS* (2007).

[39] Terry, D., Theimer, M., Petersen, K., Demers, A., Spreitzer, M., and Hauser, C. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th SOSP* (1995).

[40] von Behren, J. R., Czerwinski, S., Joseph, A. D., Brewer, E. A., and Kubiatowicz, J. Ninjamail: the design of a high-performance clustered, distributed e-mail system. In *Proceedings of the ICPP '00* (2000).

[41] Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D. E., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th OSDI* (Nov. 2006).

[42] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th OSDI* (Dec. 2002).

[43] Yu, H., and Vahdat, A. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM TOCS 20*, 3 (Aug. 2002), 239–282.

PADS: A Policy Architecture for Distributed Storage Systems

Nalini Belaramani*, Jiandan Zheng§, Amol Nayate†, Robert Soulé‡,
Mike Dahlin*, Robert Grimm‡

*The University of Texas at Austin §Amazon.com Inc. †IBM TJ Watson Research
‡New York University

Abstract

This paper presents PADS, a *policy architecture* for building distributed storage systems. A policy architecture has two aspects. First, a common set of mechanisms that allow new systems to be implemented simply by defining new policies. Second, a structure for how policies, themselves, should be specified. In the case of distributed storage systems, PADS defines a *data plane* that provides a fixed set of mechanisms for storing and transmitting data and maintaining consistency information. PADS requires a designer to define a *control plane policy* that specifies the system-specific policy for orchestrating flows of data among nodes. PADS then divides control plane policy into two parts: *routing policy* and *blocking policy*. The PADS prototype defines a concise interface between the data and control planes, it provides a declarative language for specifying routing policy, and it defines a simple interface for specifying blocking policy. We find that PADS greatly reduces the effort to design, implement, and modify distributed storage systems. In particular, by using PADS we were able to quickly construct a dozen significant distributed storage systems spanning a large portion of the design space using just a few dozen policy rules to define each system.

1 Introduction

Our goal is to make it easy for system designers to construct new distributed storage systems. Distributed storage systems need to deal with a wide range of heterogeneity in terms of devices with diverse capabilities (e.g., phones, set-top-boxes, laptops, servers), workloads (e.g., streaming media, interactive web services, private storage, widespread sharing, demand caching, preloading), connectivity (e.g., wired, wireless, disruption tolerant), and environments (e.g., mobile networks, wide area networks, developing regions). To cope with these varying demands, new systems are developed [12, 14, 19, 21, 22, 30], each making design choices that balance performance, resource usage, consistency, and availability. Because these tradeoffs are fundamental [7, 16, 34], we do not expect the emergence of a single "hero" distributed storage system to serve all situations and end the need for new systems.

This paper presents PADS, a *policy architecture* that simplifies the development of distributed storage systems. A policy architecture has two aspects.

First, a policy architecture defines a common set of mechanisms and allows new systems to be implemented simply by defining new policies. PADS casts its mechanisms as part of a *data plane* and policies as part of a *control plane*. The data plane encapsulates a set of common mechanisms that handle the details of storing and transmitting data and maintaining consistency information. System designers then build storage systems by specifying a control plane policy that orchestrates data flows among nodes.

Second, a policy architecture defines a framework for specifying policy. In PADS, we separate control plane policy into *routing* and *blocking* policy.

- *Routing policy*: Many of the design choices of distributed storage systems are simply *routing decisions* about data flows between nodes. These decisions provide answers to questions such as: "When and where to send updates?" or "Which node to contact on a read miss?", and they largely determine how a system meets its performance, availability, and resource consumption goals.
- *Blocking policy*: Blocking policy specifies predicates for when nodes must block incoming updates or local read/write requests to maintain system invariants. Blocking is important for meeting consistency and durability goals. For example, a policy might block the completion of a write until the update reaches at least 3 other nodes.

The PADS prototype is an instantiation of this architecture. It provides a concise interface between the control and data planes that is flexible, efficient, and yet simple. For routing policy, designers specify an event-driven program over an API comprising a set of *actions* that set up data flows, a set of *triggers* that expose local node information, and the abstraction of *stored events* that store and retrieve persistent state. To facilitate the specification of event-driven routing, the prototype defines a domain-specific language that allows routing policy to be written as a set of declarative rules. For defining a control plane's blocking policy, PADS defines five *blocking points* in the data plane's processing of read, write,

	Simple Client Server	Full Client Server	Coda [14]	Coda +Coop Cache	TRIP [20]	TRIP +Hier	Tier Store [6]	Tier Store +CC	Chain Repl [32]	Bayou [23]	Bayou +Small Dev	Pangaea [26]
Routing Rules	21	43	31	44	6	6	14	29	75	9	9	75
Blocking Conditions	5	6	5	5	3	3	1	1	4	3	3	1
Topology	Client/ Server	Client/ Server	Client/ Server	Client/ Server	Client/ Server	Tree	Tree	Tree	Chains	Ad-Hoc	Ad-Hoc	Ad-Hoc
Replication	Partial	Partial	Partial	Partial	Full	Full	Partial	Partial	Full	Full	Partial	Partial
Demand caching	√	√	√	√	√	√						√
Prefetching			√	√	√	√	√	√	√	√	√	√
Cooperative caching		√		√		√		√			√	√
Consistency	Sequen-tial	Sequen-tial	Open/ Close	Open/ Close	Sequen-tial	Sequen-tial	Mono. Reads	Mono. Reads	Linear-izablity	Causal	Mono. Reads	Mono. Reads
Callbacks	√	√	√	√	√	√						
Leases		√	√	√								
Inval vs. whole update propagation	Invali-dation	Invali-dation	Invali-dation	Invali-dation	Invali-dation	Invali-dation	Update	Update	Update	Update	Update	Update
Disconnected operation			√	√	√	√	√	√		√	√	√
Crash recovery	√	√	√	√	√	√	√	√	√	√	√	√
Object store interface*	√	√	√	√	√	√	√	√	√	√	√	√
File system interface*		√	√	√	√	√	√	√	√	√	√	√

Fig. 1: Features covered by case-study systems. Each column corresponds to a system implemented on PADS, and the rows list the set of features covered by the implementation. *Note that the original implementations of some systems provide interfaces that differ from the object store or file system interfaces we provide in our prototypes.

and receive-update actions; at each blocking point, a designer specifies *blocking predicates* that indicate when the processing of these actions must block.

Ultimately, the evidence for PADS's usefulness is simple: two students used PADS to construct a dozen distributed storage systems summarized in Figure 1 in a few months. PADS's ability to support these systems (1) provides evidence supporting our high-level approach and (2) suggests that the specific APIs of our PADS prototype adequately capture the key abstractions for building distributed storage systems. Notably, in contrast with the thousands of lines of code it typically takes to construct such a system using standard practice, given the PADS prototype it requires just 6-75 routing rules and a handful of blocking conditions to define each new system with PADS.

Similarly, we find it easy to add significant new features to PADS systems. For example, we add cooperative caching [5] to Coda by adding 13 rules.

This flexibility comes at a modest cost to absolute performance. Microbenchmark performance of an implementation of one system (P-Coda) built on our user-level Java PADS prototype is within ten to fifty percent of the original system (Coda [14]) in most cases and 3.3 times worse in the worst case we measured.

A key issue in interpreting Figure 1 is understanding how complete or realistic these PADS implementations are. The PADS implementations are *not* bug-compatible recreations of every detail of the original systems, but we believe they do capture the overall architecture of these designs by storing approximately the same data on each node, by sending approximately the same data across the same network links, and by enforcing the same consistency and durability semantics; we discuss our definition of *architectural equivalence* in Section 6. We also note that our PADS implementations are sufficiently complete to run file system benchmarks and that they handle important and challenging real world details like configuration files and crash recovery.

2 PADS overview

Separating mechanism from policy is an old idea. As Figure 2 illustrates, PADS does so by defining a data plane that embodies the basic mechanisms needed for storing data, sending and receiving data, and maintaining consistency information. PADS then casts policy as defining a control plane that orchestrates data flow among nodes. This division is useful because it allows the designer to focus on high-level specification of control plane policy rather than on implementation of low-level data storage, bookkeeping, and transmission details.

PADS must therefore specify an interface between the data plane and the control plane that is flexible and efficient so that it can accommodate a wide design space. At the same time, the interface must be simple so that the designer can reason about it. Section 3 and Section 4 detail the interface exposed by the data plane mechanisms to the control plane policy.

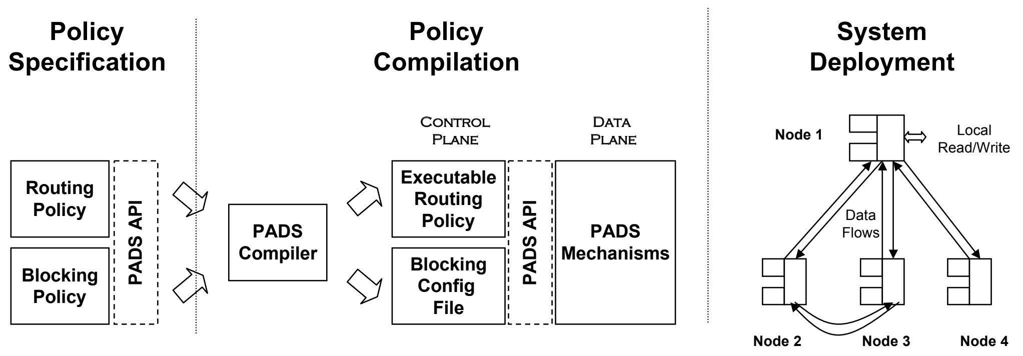


Fig. 2: PADS approach to system development.

To meet these goals and to guide a designer, PADS divides the control policy into a routing policy and a blocking policy. This division is useful because it introduces a separation of concerns for a system designer.

First, a system's trade-offs among performance, availability, and resource consumption goals largely map to routing rules. For example, sending all updates to all nodes provides excellent response time and availability, whereas caching data on demand requires fewer network and storage resources. As described in Section 3, a PADS routing policy is an event-driven program that builds on the data plane mechanisms exposed by the PADS API to set up data flows among nodes in order to transmit and store the desired data at the desired nodes.

Second, a system's durability and consistency constraints are naturally expressed as conditions that must be met when an object is read or updated. For example, the enforcement of a specific consistency semantic might require a read to block until it can return the value of the most recently completed write. As described in Section 4, a PADS blocking policy specifies these requirements as a set of predicates that block access to an object until the predicates are satisfied.

Blocking policy works together with routing policy to enforce the safety constraints and the liveness goals of a system. Blocking policy enforce safety conditions by ensuring that an operation blocks until system invariants are met, whereas routing policy guarantee liveness by ensuring that an operation will eventually unblock—by setting up data flows to ensure the conditions are eventually satisfied.

2.1 Using PADS

As Figure 2 illustrates, in order to build a distributed storage system on PADS, a system designer writes a routing policy and a blocking policy. She writes the routing policy as an event-driven program comprising a set of rules that send or fetch updates among nodes when particular events exposed by the underlying data plane occur. She writes her blocking policy as a list of predicates. She then uses a PADS compiler to translate her routing rules into Java and places the blocking predicates in a configuration file. Finally, she distributes a Java jar file containing PADS's standard data plane mechanisms and her system's control policy to the system's nodes. Once the system is running at each node, users can access locally stored data, and the system synchronizes data among nodes according to the policy.

2.2 Policies vs. goals

A PADS policy is a specific set of directives rather than a statement of a system's high-level goals. Distributed storage design is a creative process and PADS does not attempt to automate it: a designer must still devise a strategy to resolve trade-offs among factors like performance, availability, resource consumption, consistency, and durability. For example, a policy designer might decide on a client-server architecture and specify "When an update occurs at a client, the client should send the update to the server within 30 seconds" rather than stating "Machine X has highly durable storage" and "Data should be durable within 30 seconds of its creation" and then relying on the system to derive a client-server architecture with a 30 second write buffer.

2.3 Scope and limitations

PADS targets distributed storage environments with mobile devices, nodes connected by WAN networks, or nodes in developing regions with limited or intermittent connectivity. In these environments, factors like limited bandwidth, heterogeneous device capabilities, network partitions, or workload properties force interesting tradeoffs among data placement, update propagation, and consistency. Conversely, we do not target environments like well-connected clusters.

Within this scope, there are three design issues for which the current PADS prototype significantly restricts a designer's choices

First, the prototype does not support security specification. Ultimately, our policy architecture should also define flexible security primitives, and providing such primitives is important future work [18].

Second, the prototype exposes an object-store interface for local reads and writes. It does not expose other interfaces such as a file system or a tuple store. We believe that these interfaces are not difficult to incorporate. Indeed, we have implemented an NFS interface over our prototype.

Third, the prototype provides a single mechanism for conflict resolution. Write-write conflicts are detected and logged in a way that is data-preserving and consistent across nodes to support a broad range of application-level resolvers. We implement a simple last writer wins resolution scheme and believe that it is straightforward to extend PADS to support other schemes [14, 31, 13, 28, 6].

3 Routing policy

In PADS, the basic abstraction provided by the data plane is *a subscription*—a unidirectional stream of updates to a specific subset of objects between a pair of nodes. A policy designer controls the data plane's subscriptions to implement the system's routing policy. For example, if a designer wants to implement hierarchical caching, the routing policy would set up subscriptions among nodes to send updates up and to fetch data down the hierarchy. If a designer wants nodes to randomly gossip updates, the routing policy would set up subscriptions between random nodes. If a designer wants mobile nodes to exchange updates when they are in communication range, the routing policy would probe for available neighbors and set up subscriptions at opportune times.

Given this basic approach, the challenge is to define an API that is sufficiently expressive to construct a wide range of systems and yet sufficiently simple to be comprehensible to a designer. As the rest of this section details, PADS provides three sets of primitives for specifying routing policies: (1) a set of 7 *actions* that establish or remove subscriptions to direct communication of specific subsets of data among nodes, (2) a set of 9 *triggers* that expose the status of local operations and information flow, and (3) a set of 5 *stored events* that allow a routing policy to persistently store and access configuration options and information affecting routing decisions in data objects. Consequently, a system's routing policy is specified as an event-driven program that invokes the appropriate actions or accesses stored events based on the triggers received.

In the rest of this section, we discuss details of these PADS primitives and try to provide an intuition for why these few primitives can cover a large part of the design space. We do not claim that these primitives are minimal or that they are the only way to realize this approach. However, they have worked well for us in practice.

3.1 Actions

The basic abstraction provided by a PADS action is simple: *an action sets up a subscription* to route updates

Routing Actions	
Add Inval Sub	srcId, destId, objS, [startTime], LOG|CP|CP+Body
Add Body Sub	srcId, destId, objS, [startTime]
Remove Inval Sub	srcId, destId, objS
Remove Body Sub	srcId, destId, objS
Send Body	srcId, destId, objId, off, len, writerId, time
Assign Seq	objId, off, len, writerId, time
B Action	<policy defined>

Fig. 3: Routing actions provided by PADS. *objId*, *off*, and *len* indicate the object identifier, offset, and length of the update to be sent. *startTime* specifies the logical start time of the subscription. *writerId and time* indicate the logical time of a particular update. The fields for the *B Action* are policy defined.

from one node to another or *removes an established subscription* to stop sending updates. As Figure 3 shows, the subscription establishment API (*Add Inval Sub* and *Add Body Sub*) provides five parameters that allow a designer to control the scope of subscriptions:

- *Selecting the subscription type.* The designer decides whether *invalidations* or *bodies* of updates should be sent. Every update comprises an invalidation and a body. An invalidation indicates that an update of a particular object occurred at a particular instant in logical time. Invalidations aid consistency enforcement by providing a means to quickly notify nodes of updates and to order the system's events. Conversely, a body contains the data for a specific update.
- *Selecting the source and destination nodes.* Since subscriptions are unidirectional streams, the designer indicates the direction of the subscription by specifying the source node (*srcId*) of the updates and the destination node (*destId*) to which the updates should be transmitted.
- *Selecting what data to send.* The designer specifies what data to send by specifying the *objects of interest* for a subscription so that only updates for those objects are sent on the subscription. PADS exports a hierarchical namespace in which objects are identified with unique strings (e.g., /x/y/z) and a group of related objects can be concisely specified. (e.g., /a/b/*).
- *Selecting the logical start time*. The designer specifies a logical *start time* so that the subscription can send all updates that have occurred to the objects of interest from that time. The start time is specified as a partial version vector and is set by default to the receiver's current logical time.
- *Selecting the catch-up method.* If the start time for an invalidation subscription is earlier than the sender's current logical time, the sender has two options: The sender can transmit either a *log* of the updates that have occurred since the start time or a *checkpoint* that includes just the most recent update to each byterange

Local Read/Write Triggers	
Operation block	obj, off, len, blocking_point, failed_predicates
Write	obj, off, len, writerId, time
Delete	obj, writerId, time

Message Arrival Triggers	
Inval arrives	srcId, obj, off, len, writerId, time
Send body success	srcId, obj, off, len, writerId, time
Send body failed	srcId, destId, obj, off, len, writerId, time

Connection Triggers	
Subscription start	srcId, destId, objS, Inval|Body
Subscription caught-up	srcId, destId, objS, Inval
Subscription end	srcId, destId, objS, Reason, Inval|Body

Fig. 4: Routing triggers provided by PADS. *blocking_point* and *failed_predicates* indicate at which point an operation blocked and what predicate failed (refer to Section 4). *Inval | Body* indicate the type of subscription. *Reason* indicates if the subscription ended due to failure or termination.

since the start time. These options have different performance tradeoffs. Sending a log is more efficient when the number of recent changes is small compared to the number of objects covered by the subscription. Conversely, a checkpoint is more efficient if (a) the start time is in the distant past (so the log of events is long) or (b) the subscription set consists of only a few objects (so the size of the checkpoint is small). Note that once a subscription catches up with the sender's current logical time, updates are sent as they arrive, effectively putting all active subscriptions into a mode of continuous, incremental log transfer. For body subscriptions, if the start time of the subscription is earlier than the sender's current time, the sender transmits a checkpoint containing the most recent update to each byterange. The log option is not available for sending bodies. Consequently, the data plane only needs to store the most recent version of each byterange.

In addition to the interface for creating subscriptions (*Add Inval Sub* and *Add Body Sub*), PADS provides *Remove Inval Sub* and *Remove Body Sub* to remove established subscriptions, *Send Body* to send an individual body of an update that occurred at or after the specified time, *Assign Seq* to mark a previous update with a commit sequence number to aid enforcement of consistency [23], and *B Action* to allow the routing policy to send an event to the blocking policy (refer to Section 4). Figure 3 details the full routing actions API.

3.2 Triggers

PADS *triggers* expose to the control plane policy events that occur in the data plane. As Figure 4 details, these events fall into three categories.

- *Local operation* triggers inform the routing policy when an operation blocks because it needs additional information to complete or when a local write or delete occurs.
- *Message receipt* triggers inform the routing policy when an invalidation arrives, when a body arrives, or whether a send body succeeds or fails.
- *Connection* triggers inform the routing policy when subscriptions are successfully established, when a subscription has caused a receiver's state to be caught up with a sender's state (i.e., the subscription has transmitted all updates to the subscription set up to the sender's current time), or when a subscription is removed or fails.

Stored Events	
Write event	objId, eventName, field1, ..., fieldN
Read event	objId
Read and watch event	objId
Stop watch	objId
Delete events	objId

Fig. 5: PADS's stored events interface. *objId* specifies the object in which the events should be stored or read from. *eventName* defines the name of the event to be written and *field** specify the values of fields associated with it.

3.3 Stored events

Many systems need to maintain persistent state to make routing decisions. Supporting this need is challenging both because we want an abstraction that meshes well with our event-driven programming model and because the techniques must handle a wide range of scales. In particular, the abstraction must not only handle simple, global configuration information (e.g., the server identity in a client-server system like Coda [14]), but it must also scale up to per-file information (e.g., which nodes store the gold copies of each object in Pangaea [26].)

To provide a uniform abstraction to address this range of demands, PADS provides *stored events* primitives to store events into a data object in the underlying persistent object store. Figure 5 details the full API for stored events. A *Write Event* stores an event into an object and a *Read Event* causes all events stored in an object to be fed as input to the routing program. The API also includes *Read and Watch* to produce new events whenever they are added to an object, *Stop Watch* to stop producing new events from an object, and *Delete Events* to delete all events in an object.

For example, in a hierarchical information dissemination system, a parent *p* keeps track of what volumes a child subscribes to so that the appropriate subscriptions can be set up. When a child *c* subscribes to a new volume *v*, *p* stores the information in a configuration object */subInfo* by generating a *<write_event, /subInfo, child_sub, p, c, v>* action. When this information is needed, for example on startup or recovery, the parent generates a *<read_event, /subInfo>* action that causes a *<child_sub, p, c, v>* event to be generated for each item stored in the object. The *child_sub* events, in turn, trigger event handlers in the routing policy that re-establish subscriptions.

3.4 Specifying routing policy

A routing policy is specified as an event-driven program that invokes actions when local triggers or stored events are received. PADS provides R/OverLog, a language based on the OverLog routing language [17] and a runtime to simplify writing event-driven policies.[1]

As in OverLog, a R/OverLog program defines a set of *tables* and a set of *rules*. Tables store tuples that represent internal state of the routing program. This state does not need to be persistently stored, but is required for policy execution and can dynamically change. For example, a table might store the ids of currently reachable nodes. Rules are fired when an event occurs and the constraints associated with the rule are met. The input event to a rule can be a trigger injected from the local data plane, a stored event injected from the data plane's persistent state, or an internal event produced by another rule on a local machine or a remote machine. Every rule generates a single event that invokes an action in the data plane, fires another local or remote rule, or is stored in a table as a tuple. For example, the following rule:

```
EVT_clientReadMiss(@S, X, Obj, Off, Len):-
        TRIG_operationBlock(@X, Obj, Off, Len, BPoint,_),
        TBL_serverId(@X, S),
        BPoint == "readNowBlock".
```

specifies that whenever node *X* receives a *operationBlock* trigger informing it of an operation blocked at the *readNowBlock* blocking point, it should produce a new event *clientReadMiss* at server *S*, identified by *serverId* table. This event is populated with the fields from the triggering event and the constraints—the client id (*X*), the data to be read (*obj, off, len*), and the server to contact (*S*). Note that the underscore symbol (_) is a wildcard that matches any list of predicates and the at symbol (@) specifies the node at which the event occurs. A more complete discussion of OverLog language and execution model is available elsewhere [17].

4 Blocking policy

A system's durability and consistency constraints can be naturally expressed as invariants that must hold when an object is accessed. In PADS, the system designer specifies these invariants as a set of predicates that block access to an object until the conditions are satisfied. To that end, PADS (1) defines 5 *blocking points* for which a system designer specifies predicates, (2) provides 4 *built-in conditions* that a designer can use as predicates, and (3) exposes a *B_Action interface* that allows a designer to specify custom conditions based on routing information.

[1] Note that if learning a domain specific language is not one's cup of tea, one can define a (less succinct) policy by writing Java handlers for PADS triggers and stored events to generate PADS actions and stored events.

Predefined Conditions on Local Consistency State	
isValid	Block until node has received the body corresponding to the highest received invalidation for the target object
isComplete	Block until object's consistency state reflects all updates before the node's current logical time
isSequenced	Block until object's total order is established
maxStaleness *nodes, count, t*	Block until all writes up to *(operationStartTime-t)* from *count* nodes in *nodes* have been received.
User Defined Conditions on Local or Distributed State	
B_Action *event-spec*	Block until an event with fields matching *event-spec* is received from routing policy

Fig. 6: Conditions available for defining blocking predicates.

The set of predicates for each blocking point makes up the blocking policy of the system.

4.1 Blocking points

PADS defines five points for which a policy can supply a predicate and a timeout value to block a request until the predicate is satisfied or the timeout is reached. The first three are the most important:

- *ReadNowBlock* blocks a read until it will return data from a moment that satisfies the predicate. Blocking here is useful for ensuring consistency (e.g., *block until a read is guaranteed to return the latest sequenced write.*)
- *WriteEndBlock* blocks a write request after it has updated the local object but before it returns. Blocking here is useful for ensuring consistency (e.g., *block until all previous versions of this data are invalidated*) and durability (e.g., *block here until the update is stored at the server.*)
- *ApplyUpdateBlock* blocks an invalidation received from the network before it is applied to the local data object. Blocking here is useful to increase data availability by allowing a node to continue serving local data, which it might not have been able to if the data had been invalidated. (e.g., *block applying a received invalidation until the corresponding body is received.*)

PADS also provides *WriteBeforeBlock* to block a write before it modifies the underlying data object and *ReadEndBlock* to block a read after it has retrieved data from the data plane but before it returns.

4.2 Blocking conditions

PADS provides a set of predefined conditions, listed in Figure 6, to specify predicates at each blocking point. A blocking predicate can use any combination of these predicates. The first four conditions provide an interface to the consistency bookkeeping information maintained in the data plane on each node.

- *IsValid* requires that the last body received for an object is as new as the last invalidation received for that object. *isValid* is useful for enforcing monotonic coherence on reads[2] and for maximizing availability by ensuring that invalidations received from other nodes are not applied until they can be applied with their corresponding bodies [6, 20].

- *IsComplete* requires that a node receives all invalidations for the target object up to the node's current logical time. *IsComplete* is needed because liveness policies can direct arbitrary subsets of invalidations to a node, so a node may have gaps in its consistency state for some objects. If the predicate for *ReadNowBlock* is set to *isValid* and *isComplete*, reads are guaranteed to see causal consistency.
- *IsSequenced* requires that the most recent write to the target object has been assigned a position in a total order. Policies that want to ensure sequential or stronger consistency can use the *Assign Seq* routing action (see Figure 3) to allow a node to sequence other nodes' writes and specify the *isSequenced* condition as a *ReadNowBlock* predicate to block reads of unsequenced data.
- *MaxStaleness* is useful for bounding real time staleness.

The fifth condition on which a blocking predicate can be based on is *B_Action*. A *B_Action* condition provides an interface with which a routing policy can signal an arbitrary condition to a blocking predicate. An operation waiting for *event-spec* unblocks when the routing rules produce an event whose fields match the specified spec.

Rationale. The first four, built-in consistency bookkeeping primitives exposed by this API were developed because they are simple and inexpensive to maintain within the data plane [2, 35] but they would be complex or expensive to maintain in the control plane. Note that they are primitives, not solutions. For example, to enforce linearizability, one must not only ensure that one reads only sequenced updates (e.g., via blocking at *ReadNowBlock* on *isSequenced*) but also that a write operation blocks until all prior versions of the object have been invalidated (e.g., via blocking at *WriteEndBlock* on, say, the *B_Action allInvalidated* which the routing policy produces by tracking data propagation through the system).

Beyond the four pre-defined conditions, a policy-defined *B_Action* condition is needed for two reasons. The most obvious need is to avoid having to predefine all possible interesting conditions. The other reason for allowing conditions to be met by actions from the event-driven routing policy is that when conditions reflect distributed state, policy designers can exploit knowledge of their system to produce better solutions than a generic implementation of the same condition. For example, in the client-server system we describe in Section 6, a client blocks a write until it is sure that all other clients caching the object have been invalidated. A generic implementation of the condition might have required the client that issued the write to contact all other clients. However, a policy-defined event can take advantage of the client-server topology for a more efficient implementation. The client sets the *writeEndBlock* predicate to a policy-defined *receivedAllAcks* event. Then, when an object is written and other clients receive an invalidation, they send acknowledgements to the server. When the server gathers acknowledgements from all other clients, it generates a *receivedAllAcks* action for the client that issued the write.

[2]Any read on an object will return a version that is equal to or newer than the version that was last read.

5 Constructing P-TierStore

As an example of how to build a system with PADS, we describe our implementation of P-TierStore, a system inspired by TierStore [6]. We choose this example because it is simple and yet exercises most aspects of PADS.

5.1 System goals

TierStore is a distributed object storage system that targets developing regions where networks are bandwidth-constrained and unreliable. Each node reads and writes specific subsets of the data. Since nodes must often operate in disconnected mode, the system prioritizes 100% availability over strong consistency.

5.2 System design

In order to achieve these goals, TierStore employs a hierarchical publish/subscribe system. All nodes are arranged in a tree. To propagate updates up the tree, every node sends all of its updates and its children's updates to its parent. To flood data down the tree, data are partitioned into "publications" and every node subscribes to a set of publications from its parent node covering its own interests and those of its children. For consistency, TierStore only supports single-object monotonic reads coherence.

5.3 Policy specification

In order to construct P-TierStore, we decompose the design into routing policy and blocking policy.

A 14-rule routing policy establishes and maintains the publication aggregation and multicast trees. A full listing of these rules is available elsewhere [3]. In terms of PADS primitives, each connection in the tree is simply an invalidation subscription and a body subscription between a pair of nodes. Every PADS node stores in configuration objects the ID of its parent and the set of publications to subscribe to.

On start up, a node uses stored events to read the configuration objects and store the configuration information in R/OverLog tables (4 rules). When it knows of the ID of its parent, it adds subscriptions for every item in the publication set (2 rules). For every child, it adds subscriptions for "/*" to receive all updates from the child (2 rules). If an application decides to subscribe to another publication, it simply writes to the configuration object. When this update occurs, a new stored event is generated and the routing rules add subscriptions for the new publication.

Recovery. If an incoming or an outgoing subscription fails, the node periodically tries to re-establish the connection (2 rules). Crash recovery requires no extra policy rules. When a node crashes and starts up, it simply re-establishes the subscriptions using its local logical time as the subscription's start time. The data plane's subscription mechanisms automatically detect which updates the receiver is missing and send them.

Delay tolerant network (DTN) support. P-TierStore supports DTN environments by allowing one or more mobile PADS nodes to relay information between a parent and a child in a distribution tree. In this configuration, whenever a relay node arrives, a node subscribes to receive any new updates the relay node brings and pushes all new local updates for the parent or child subscription to the relay node (4 rules).

Blocking policy. Blocking policy is simple because TierStore has weak consistency requirements. Since TierStore prefers stale available data to unavailable data, we set the *ApplyUpdateBlock* to *isValid* to avoid applying an invalidation until the corresponding body is received.

TierStore vs. P-TierStore. Publications in TierStore are defined by a container name and depth to include all objects up to that depth from the root of the publication. However, since P-TierStore uses a name hierarchy to define publications (e.g., /publication1/*), all objects under the directory tree become part of the subscription with no limit on depth.

Also, as noted in Section 2.3, PADS provides a single conflict-resolution mechanism, which differs from that of TierStore in some details. Similarly, TierStore provides native support for directory objects, while PADS supports a simple untyped object store interface.

6 Experience and evaluation

Our central thesis is that it is useful to design and build distributed storage systems by specifying a control plane comprising a routing policy and a blocking policy. There is no quantitative way to prove that this approach is good, so we base our evaluation on our experience using the PADS prototype.

Figure 1 conveys the main result of this paper: *using* PADS*, a small team was able to construct a dozen significant systems with a large number of features that cover a large part of the design space.* PADS qualitatively reduced the effort to build these systems and increased our team's capabilities: we do not believe a small team such as ours could have constructed anything approaching this range of systems without PADS.

In the rest of this section, we elaborate on this experience by first discussing the range of systems studied, the development effort needed, and our debugging experience. We then explore the realism of the systems we constructed by examining how PADS handles key system-building problems like configuration, consistency, and crash recovery. Finally, we examine the costs of PADS's generality: what overheads do our PADS implementations pay compared to ideal or hand-crafted implementations?

Approach and environment. The goal of PADS is to help people develop new systems. One way to evaluate PADS would be to construct a new system for a new demanding environment and report on that experience. We choose a different approach—constructing a broad range of existing systems—for three reasons. First, a single system may not cover all of the design choices or test the limits of PADS. Second, it might not be clear how to generalize the experience from building one system to building others. Third, it might be difficult to disentangle the challenges of designing a new system for a new environment from the challenges of realizing a design using PADS.

The PADS prototype uses PRACTI [2, 35] to provide the data plane mechanisms. We implement a R/OverLog to Java compiler using the XTC toolkit [9]. Except where noted, all experiments are carried out on machines with 3GHz Intel Pentium IV Xeon processors, 1GB of memory, and 1Gb/s Ethernet. Machines and network connections are controlled via the Emulab software [33]. For software, we use Fedora Core 8, BEA JRockit JVM Version 27.4.0, and Berkeley DB Java Edition 3.2.23.

6.1 System development on PADS

This section describes the design space we have covered, how the agility of the resulting implementations makes them easy to adapt, the design effort needed to construct a system under PADS, and our experience debugging and analyzing our implementations.

6.1.1 Flexibility

We constructed systems chosen from the literature to cover large part of the design space. We refer to our implementation of each system as P-system (e.g., P-Coda). To provide a sense of the design space covered, we provide a short summary of each of the system's properties below and in Figure 1.

Generic client-server. We construct a simple client-server (P-SCS) and a full featured client-server (P-FCS).

Objects are stored on the server, and clients cache the data from the server on demand. Both systems implement *callbacks* in which the server keeps track of which clients are storing a valid version of an object and sends invalidations to them whenever the object is updated. The difference between P-SCS and P-FCS is that P-SCS assumes full object writes while P-FCS supports partial-object writes and also implements *leases* and *cooperative caching*. Leases [8] increase availability by allowing a server to break a callback for unreachable clients. Cooperative caching [5] allows clients to retrieve data from a nearby client rather than from the server. Both P-SCS and P-FCS enforce *sequential consistency* semantics and ensure durability by making sure that the server always holds the body of the most recently completed write of each object.

Coda [14]. Coda is a client-server system that supports mobile clients. P-Coda includes the client-server protocol and the features described in Kistler et al.'s paper [14]. It does not include server replication features detailed in [27]. Our discussion focuses on P-Coda. P-Coda is similar to P-FCS—it implements callbacks and leases but not cooperative caching; also, it guarantees *open-to-close consistency*[3] instead of sequential consistency. A key feature of Coda is its support for *disconnected operation*—clients can access locally cached data when they are offline and propagate offline updates to the server on reconnection. Every client has a *hoard list* that specifies objects to be periodically fetched from the server

TRIP [20]. TRIP is a distributed storage system for large-scale information dissemination: all updates occur at a server and all reads occur at clients. TRIP uses a self-tuning prefetch algorithm and delays applying invalidations to a client's locally cached data to maximize the amount of data that a client can serve from its local state. TRIP guarantees sequential consistency via a simple algorithm that exploits the constraint that all writes are carried out by a single server.

TierStore [6]. TierStore is described in Section 5.

Chain replication [32]. Chain replication is a server replication protocol that guarantees linearizability and high availability. All the nodes in the system are arranged in a chain. Updates occur at the head and are only considered complete when they have reached the tail.

Bayou [23]. Bayou is a server-replication protocol that focuses on peer-to-peer data sharing. Every node has a local copy of all of the system's data. From time to time, a node picks a peer to exchange updates with via anti-entropy sessions.

[3]Whenever a client opens a file, it always gets the latest version of the file known to the server, and the server is not updated until the file is closed.

Pangaea [26] Pangaea is a peer-to-peer distributed storage system for wide area networks. Pangaea maintains a connected graph across replicas for each object, and it pushes updates along the graph edges. Pangaea maintains three gold replicas for every object to ensure data durability.

Summary of design features. As Figure 1 further details, these systems cover a wide range of design features in a number of key dimensions. For example,

- *Replication:* full replication (Bayou, Chain Replication, and TRIP), partial replication (Coda, Pangaea, P-FCS, and TierStore), demand caching (Coda, Pangaea, and P-FCS),
- *Topology:* structured topologies such as client-server (Coda, P-FCS, and TRIP), hierarchical (TierStore), and chain (Chain Replication); unstructured topologies (Bayou and Pangaea). Invalidation-based (Coda and P-FCS) and update-based (Bayou, TierStore, and TRIP) propagation.
- *Consistency:* monotonic-reads coherence (Pangaea and TierStore), casual (Bayou), sequential (P-FCS and TRIP), and linearizability (Chain Replication); techniques such as callbacks (Coda, P-FCS, and TRIP) and leases (Coda and P-FCS).
- *Availability:* Disconnected operation (Bayou, Coda, TierStore, and TRIP), crash recovery (all), and network reconnection (all).

Goal: Architectural equivalence. We build systems based on the above designs from the literature, but constructing perfect, "bug-compatible" duplicates of the original systems using PADS is not a realistic (or useful) goal. On the other hand, if we were free to pick and choose arbitrary subsets of features to exclude, then the bar for evaluating PADS is too low: we can claim to have built any system by simply excluding any features PADS has difficulty supporting.

Section 2.3 identifies three aspects of system design—security, interface, and conflict resolution—for which PADS provides limited support, and our implementations of the above systems do not attempt to mimic the original designs in these dimensions.

Beyond that, we have attempted to faithfully implement the designs in the papers cited. More precisely, although our implementations certainly differ in some details, we believe we have built systems that are *architecturally equivalent* to the original designs. We define architectural equivalence in terms of three properties:

E1. *Equivalent overhead.* A system's network bandwidth between any pair of nodes and its local storage at any node are within a small constant factor of the target system.

E2. *Equivalent consistency.* The system provides consistency and staleness properties that are at least as strong as the target system's.

E3. *Equivalent local data.* The set of data that may be accessed from the system's local state without network communication is a superset of the set of data that may be accessed from the target system's local state. Notice that this property addresses several factors including latency, availability, and durability.

There is a principled reason for believing that these properties capture something about the essence of a replication system: they highlight how a system resolves the fundamental CAP (Consistency vs. Availability vs. Partition-resilience) [7] and PC (Performance vs. Consistency) [16] trade-offs that any distributed storage system must make.

6.1.2 Agility

As workloads and goals change, a system's requirements also change. We explore how systems build with PADS can be adapted by adding new features. We highlight two cases in particular: our implementation of Bayou and Coda. Even though they are simple examples, they demonstrate that being able to easily adapt a system to send the right data along the right paths can pay big dividends.

P-Bayou small device enhancement. P-Bayou is a server-replication protocol that exchanges updates between pairs of servers via an anti-entropy protocol. Since the protocol propagates updates for the whole data set to every node, P-Bayou cannot efficiently support smaller devices that have limited storage or bandwidth.

It is easy to change P-Bayou to support small devices. In the original P-Bayou design, when anti-entropy is triggered, a node connects to a reachable peer and subscribes to receive invalidations and bodies for all objects using a subscription set "/*". In our small device variation, a node uses stored events to read a list of directories from a per-node configuration file and subscribes only for the listed subdirectories. This change required us to modify two routing rules.

This change raises an issue for the designer. If a small device *C* synchronizes with a first complete server *S*1, it will not receive updates to objects outside of its subscription sets. These omissions will not affect *C* since *C* will not access those objects. However, if *C* later synchronizes with a second complete server *S*2, *S*2 may end up with causal gaps in its update logs due to the missing updates that *C* doesn't subscribe to. The designer has three choices: weaken consistency from causal to per-object

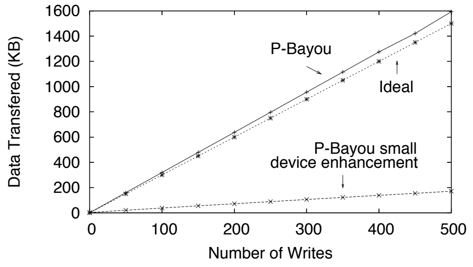


Fig. 7: Anti-Entropy bandwidth on P-Bayou

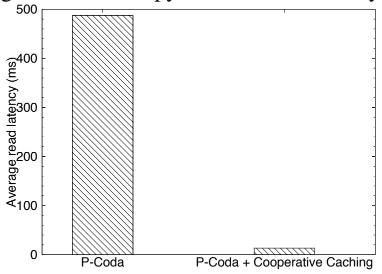


Fig. 8: Average read latency of P-Coda and P-Coda with cooperative caching.

coherence; restrict communication to avoid such situations (e.g., prevent *C* from synchronizing with *S*2); or weaken availability by forcing *S*2 to fill its gaps by talking to another server before allowing local reads of potentially stale objects. We choose the first, so we change the blocking predicate for reads to no longer require the *isComplete* condition. Other designers may make different choices depending on their environment and goals.

Figure 7 examines the bandwidth consumed to synchronize 3KB files in P-Bayou and serves two purposes. First, it demonstrates that the overhead for anti-entropy in P-Bayou is relatively small even for small files compared to an *ideal* Bayou implementation (plotted by counting the bytes of data that must be sent ignoring all metadata overheads.) More importantly, it demonstrates that if a node requires only a fraction (e.g., 10%) of the data, the *small device enhancement*, which allows a node to synchronize a subset of data, greatly reduces the bandwidth required for anti-entropy.

P-Coda and cooperative caching. In P-Coda, on a read miss, a client is restricted to retrieving data from the server. We add cooperative caching to P-Coda by adding 13-rules: 9 to monitor the reachability of nearby nodes, 2 to retrieve data from a nearby client on a read miss, and 2 to fall back to the server if the client cannot satisfy the data request.

Figure 8 shows the difference in read latency for misses on a 1KB file with and without support for cooperative caching. For the experiment, the rount-trip latency between the two clients is 10ms, whereas the round-trip latency between a client and server is almost 500ms. When data can be retrieved from a nearby client, read performance is greatly improved. More importantly, with this new capability, clients can share data even when disconnected from the server.

6.1.3 Ease of development

Each of these systems took a few days to three weeks to construct by one or two graduate students with part time effort. The time includes mapping the original system design to PADS policy primitives, implementation, testing, and debugging. Mapping the design of the original implementation to routing and blocking policy was challenging at first but became progressively easier. Once the design work was done, the implementation did not take long.

Note that routing rules and blocking conditions are extremely simple, low-level building bocks. Each routing rule specifies the conditions under which a single tuple should be produced. R/Overlog lets us specify routing rules succinctly—across all of our systems, each routing rule is from 1 to 3 lines of text. The count of blocking conditions exposes the complexity of the blocking predicates: each blocking predicate is an equation across zero or more blocking condition elements from Figure 6, so the count of at most 10 blocking conditions for a policy indicates that across all of that policy's blocking predicates, a total of 10 conditions were used. As Figure 1 indicates, each system was implemented in fewer than 100 routing rules and fewer than 10 blocking conditions.

6.1.4 Debugging and correctness

Three aspects of PADS help simplify debugging and reasoning about the correctness of PADS systems.

First, the conciseness of PADS policy greatly facilitates analysis, peer review, and refinement of design. It was extremely useful to be able to sit down and walk through an entire design in a one or two hour meeting.

Second, the abstractions themselves divide work in a way that simplifies reasoning about correctness. For example, we find that the separation of policy into routing and blocking helps reduce the risk of consistency bugs. A system's consistency and durability requirements are specified and enforced by simple blocking predicates, so it is not difficult to get them right. We must then design our routing policy to deliver sufficient data to a node to eventually satisfy the predicates and ensure liveness.

Third, domain-specific languages can facilitate the use of model checking [4]. As future work, we intend to implement a translator from R/Overlog to Promela [1] so that policies can be model checked to test the correctness of a system's implementation.

6.2 Realism

When building a distributed storage system, a system designer needs to address issues that arise in practical deployments such as configuration options, local crash recovery, distributed crash recovery, and maintaining consistency and durability despite crashes and network failures. PADS makes it easy to tackle these issues for three reasons.

First, since the stored events primitive allows routing policies to access local objects, policies can store and retrieve configuration and routing options on-the-fly. For example, in P-TierStore, a nodes stores in a configuration object the publications it wishes to access. In P-Pangaea, the parent directory object of each object stores the list of nodes from which to fetch the object on a read miss.

Second, for consistency and crash recovery, the underlying subscription mechanisms insulate the designer from low-level details. Upon recovery, local mechanisms first reconstruct local state from persistent logs. Also, PADS's subscription primitives abstract away many challenging details of resynchronizing node state. Notably, these mechanisms track consistency state even across crashes that could introduce gaps in the sequences of invalidations sent between nodes. As a result, crash recovery in most systems simply entails restoring lost subscriptions and letting the underlying mechanisms ensure that the local state reflects any updates that were missed.

Third, blocking predicates greatly simplify maintaining consistency during crashes. If there is a crash and the required consistency semantics cannot be guaranteed, the system will simply block access to "unsafe" data. On recovery, once the subscriptions have been restored and the predicates are satisfied, the data become accessible again.

In each of the PADS systems we constructed, we implemented support for these practical concerns. Due to space limitations we focus this discussion on the behaviour of two systems under failure: the full featured client server system (P-FCS) and TierStore (P-TierStore). Both are client-server based systems, but they have very different consistency guarantees. We demonstrate the systems are able to provide their corresponding consistency guarantees despite failures.

Consistency, durability, and crash recovery in P-FCS and P-TierStore Our experiment uses one server and two clients. To highlight the interactions, we add a 50ms delay on the network links between the clients and the server. Client C1 repeatedly reads an object and then sleeps for 500ms, and Client C2 repeatedly writes increasing values to the object and sleeps for 2000ms. We plot the start time, finish time, and value of each operation.

Figure 9 illustrates behavior of P-FCS under failures. P-FCS guarantees sequential consistency by maintaining per-object callbacks [11], maintaining object leases [8], and blocking the completion of a write until the server has stored the write and invalidated all other client

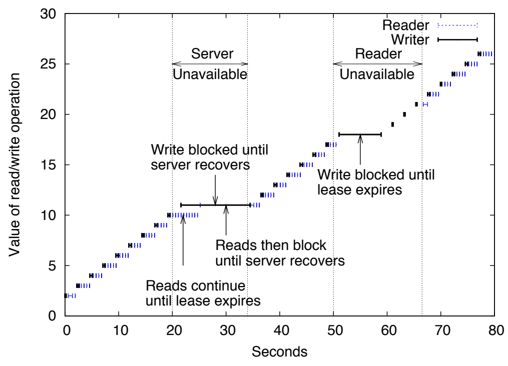


Fig. 9: Demonstration of full client-server system, P-FCS, under failures. The x axis shows time and the y axis shows the value of each read or write operation.

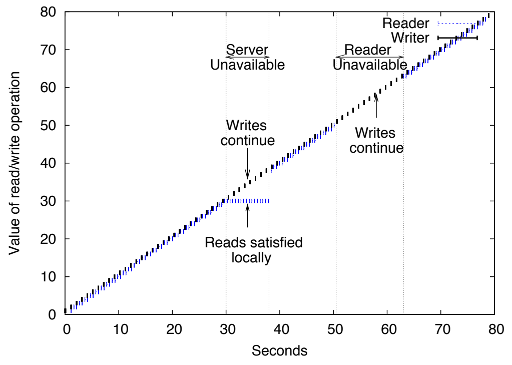


Fig. 10: Demonstration of TierStore under a workload similar to that in Figure 9.

caches. We configure the system with a 10 second lease timeout. During the first 20 seconds of the experiment, as the figure indicates, sequential consistency is enforced. We kill (kill -9) the server process 20 seconds into the experiment and restart it 10 seconds later. While the server is down, writes block immediately but reads continue until the lease expires after which reads block as well. When we restart the server, it recovers its local state and then resumes processing requests. Both reads and writes resume shortly after the server restarts, and the subscription reestablishment and blocking policy ensure that consistency is maintained.

We kill the reader, C1, at 50 seconds and restart it 15 seconds later. Initially, writes block, but as soon as the lease expires, writes proceed. When the reader restarts, reads resume as well.

Figure 10 illustrates a similar scenario using P-TierStore. P-TierStore enforces monotonic reads coherence rather than sequential consistency, and it propagates updates via subscriptions when the network is available. As a result, all reads and writes complete locally and without blocking despite failures. During periods of no failures, the reader receives updates quickly and reads return recent values. However, if the server is unavailable,

	Ideal	**PADS Prototype**
Subscription setup		
Inval Subscription with LOG catch-up	$O(N_{SSPrevUpdates})$	$O(N_{nodes} + N_{SSPrevUpdates})$
Inval Subscription with CP from time=0	$O(N_{SSObj})$	$O(N_{SSObj})$
Inval Subscription with CP from time=VV	$O(N_{SSObjUpd})$	$O(N_{nodes} + N_{SSObjUpd})$
Body Subscription	$O(N_{SSObjUpd})$	$O(N_{SSObjUpd})$
Transmitting updates		
Inval Subscription	$O(N_{SSNewUpdates})$	$O(N_{SSNewUpdates})$
Body Subscription	$O(N_{SSNewUpdates})$	$O(N_{SSNewUpdates})$

Fig. 11: Network overheads of primitives. Here, N_{nodes} is the number of nodes. N_{SSObj} is the number of objects in the subscription set. $N_{SSPrevUpdates}$ and $N_{SSObjUpd}$ are the number of updates that occurred and the number objects in the subscription set that were modified from a subscription start time to the current logical time. $N_{SSNewUpdates}$ is the number of updates to the subscription set that occur after the subscription has caught up to the sender's logical time.

writes still progress, and the reads return values that are locally stored even if they are stale.

6.3 Performance

The programming model exposed to designers must have predictable costs. In particular, the volume of data stored and sent over the network should be proportional to the amount of information a node is interested in.

We carry out performance evaluation of PADS in two steps. First, we evaluate the fundamental costs associated with the PADS architecture. In particular, we argue that network overheads of PADS are within reasonable bounds of ideal implementations and highlight when they depart from ideal.

Second, we evaluate the absolute performance of the PADS prototype. We quantify overheads associated with the primitives via micro-benchmarks and compare the performance of two implementations of the same system: the original implementation with the one built over PADS. We find that P-Coda is as much as 3.3 times worse than Coda.

6.3.1 Fundamental overheads and scalability

Figure 11 shows the network cost associated with our prototype's implementation of PADS's primitives and indicates that our costs are close to the ideal of having actual costs be proportional to the amount of new information transferred between nodes. Note that these ideal costs may not be able always be achievable.

There are two ways that PADS sends extra information.

First, during invalidation subscription setup in PADS the sender transmits a version vector indicating the start time of the subscription and catch-up information so that the receiver can determine if the catch-up information introduces gaps in the receiver's consistency state. That

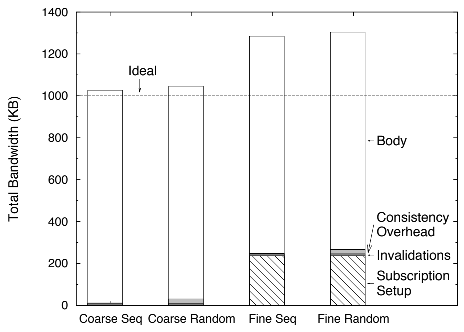


Fig. 12: Network bandwidth cost to synchronize 1000 10KB files, 100 of which are modified.

cost is then amortized over all the updates sent on the connection. Also, this cost can be avoided by starting a subscription at logical time 0 with a checkpoint rather than a log for catching up to the current time. Note, checkpoint catch-up is particularly cheap when interest sets are small.

Second, in order to support flexible consistency, invalidation subscriptions also carry extra information such as imprecise invalidations [2]. Imprecise invalidations summarize updates to objects out of the subscription set and are sent to mark logical gaps in the casual stream of invalidations. The number of imprecise invalidations sent depends on the workload and is never more than the number of invalidations of updates to objects in the subscription set sent. The size of imprecise invalidations depends on the locality of the workload and how compactly the invalidations compress into imprecise invalidations.

Overall, we expect PADS to scale well to systems with large numbers of objects or nodes—subscription sets and imprecise invalidations ensure that the number of records transferred is proportional to amount of data of interest (and not to the overall size of the database), and the per-node overheads associated with the version vectors used to set up some subscriptions can be amortized over all of the updates sent.

6.3.2 Quantifying the constants

We run experiments to investigate the constant factors in the cost model and quantify the overheads associated with subscription setup and flexible consistency. Figure 12 illustrates the synchronization cost for a simple scenario. In this experiment, there are 10,000 objects in the system organized into 10 groups of 1,000 objects each, and each object's size is 10KB. The reader registers to receive invalidations for one of these groups. Then, the writer updates 100 of the objects in each group. Finally, the reader reads all the objects.

We look at four scenarios representing combinations of coarse-grained vs. fine-grained synchronization and of writes with locality vs. random writes. For coarse-grained synchronization, the reader creates a single invalidation subscription and a single body subscription spanning all 1000 objects in the group of interest and receives 100 updated objects. For fine-grained synchronization, the reader creates 1000 invalidation subscriptions, each for one object, and fetches each of the 100 updated bodies. For writes with locality, the writer updates 100 objects in the ith group before updating any in the $i+1$st group. For random writes, the writer intermixes writes to different groups in a random order.

Four things should be noted. First, the synchronization overheads are small compared to the body data transferred. Second, the "extra" overheads associated with PADS subscription setup and flexible consistency over the best case is a small fraction of the total overhead in all cases. Third, when writes have locality, the overhead of flexible consistency drops further because larger numbers of invalidations are combined into an imprecise invalidation. Fourth, coarse-grained synchronization has lower overhead than fine-grained synchronization because it avoids per-object subscription setup costs.

Similarly, Figure 7 compares the bandwidth overhead associated with using a PADS system implementation with an ideal implementation. As the figure indicates, the bandwidth to propagate updates is close to ideal implementations. The extra overhead is due to the meta-data sent with each update.

	1KB objects		100KB objects	
	Coda	P-Coda	Coda	P-Coda
Cold read	1.51	4.95 *(3.28)*	11.65	9.10 *(0.78)*
Hot read	0.15	0.23 *(1.53)*	0.38	0.43 *(1.13)*
Connected Write	36.07	47.21 *(1.31)*	49.64	54.75 *(1.10)*
Disconnected Write	17.2	15.50 *(0.88)*	18.56	20.48 *(1.10)*

Fig. 13: Read and write latencies in milliseconds for Coda and P-Coda. The numbers in parantheses indicate factors of overhead. The values are averages of 5 runs.

6.3.3 Absolute Performance

Our goal is to provide sufficient performance to be useful. We compare the performance of a hand-crafted implementation of a system (Coda) that has been in production use for over a decade and a PADS implementation of the same system (P-Coda). We expect to pay some overheads for three reasons. First, PADS is a relatively untuned prototype rather than well-tuned production code. Second, our implementation emphasizes portability and simplicity, so PADS is written in Java and stores data using BerkeleyDB rather than running on bare metal. Third, PADS provides additional functionality such as tracking consistency metadata, some of which may not be required by a particular hand-crafted system.

Figure 13 compares the client-side read and write latencies under Coda and P-Coda. The systems are set up in a two client configuration. To measure the read latencies, client C1 has a collection of 1,000 objects and Client C2 has none. For cold reads, Client C2 randomly selects 100 objects to read. Each read fetches the object from the server and establishes a callback for the object. C2 re-reads those objects to measure the hot-read latency. To measure the connected write latency, both C1 and C2 initially store the same collection of 1,000 objects. C2 selects 100 objects to write. The write will cause the server to store the update and break a callback with C1 before the write completes at C2. Disconnected writes are measured by disconnecting C2 from the server and writing to 100 randomly selected objects.

The performance of PADS's implementation is comparable to hand-crafted C implementation in most cases and is at most 3 times worse in the worst case we measured.

7 Related work

PADS and PRACTI. We use a modified version of PRACTI [2, 35] as the data plane for PADS. Writing a new policy in PADS differs from constructing a system using PRACTI alone for three reasons.

1. PADS *adds key abstractions* not present in PRACTI such as the separation of routing policy from blocking policy, stored events, and commit actions.
2. PADS *significantly changes* abstractions from those provided in PRACTI. We distilled the interface between mechanism and policy to the handful of calls in Figures 3, 4, and 5, and we changed the underlying protocols and mechanisms to meet the needs of the data plane required by PADS. For example, where the original PRACTI protocol provides the abstraction of *connections* between nodes, each of which carries one subscription, PADS provides the more lightweight abstraction of *subscriptions* which forced us to redesign the protocol to multiplex subscriptions onto a single connection between a pair of nodes in order to efficiently support fine-grained subscriptions and dynamic addition of new items to a subscription. Similarly, where PRACTI provides the abstraction of *bound invalidations* to make sure that bodies and updates propagate together, PADS provides the more flexible *blocking predicates*, and where PRACTI hard-coded several mechanisms to track the progress of updates through the system, PADS simply triggers the routing policy and lets the routing policy handle whatever notifications are needed.
3. PADS provides *R/OverLog* which has proven to be a convenient way to design about, write, and debug routing policies.

The whole is more important than the parts. Building systems with PADS is much simpler than without. In some cases this is because PADS provides abstractions not present in PRACTI. In others, it is "merely" because PADS provides a better way of thinking about the problem.

R/OverLog and OverLog R/OverLog extends OverLog [17] by (1) adding type information to events, (2) providing an interface to pass *triggers*, *actions*, and *stored events* as tuples between PADS and the R/OverLog program, and (3) restricting the syntax slightly to allow us to implement a R/OverLog-to-Java compiler that produces executables that are more stable and faster than programs under the more general P2 [17] runtime system.

Other frameworks. A number of other efforts have defined frameworks for constructing distributed storage systems for different environments. Deceit [29] focuses on distributed storage across a well-connected cluster of servers. Stackable file systems [10] seek to provide a way to add features and compose file systems, but it focuses on adding features to local file systems.

Some systems, such as Cimbiosys [24], distribute data among nodes not based on object identifiers or file names, but rather on content-based filters. We see no fundamental barriers to incorporating filters in PADS to identify sets of related objects. This would allow system designers to set up subscriptions and maintain consistency state in terms of filters rather than object-name prefixes.

PADS follows in the footsteps of efforts to define runtime systems or domain-specific languages to ease the construction of routing [17], overlay [25], cache consistency protocols [4], and routers [15].

8 Conclusion

Our goal is to allow developers to quickly build new distributed storage systems. This paper presents PADS, a policy architecture that allows developers to construct systems by specifying policy without worrying about complex low-level implementation details. Our experience has led us to make two conclusions: First, the approach of constructing a system in terms of a routing policy and a blocking policy over a data plane greatly reduces development time. Second, the range of systems implemented with the small number of primitives exposed by the API suggest that the primitives adequately capture the key abstractions for building distributed storage systems.

Acknowledgements

The authors would like to thank the anonymous reviewers whose comments and suggestions have helped shape this paper. We would also like to thank Petros Maniatis and Amin Vahdat for their valuable insights in the early drafts of this paper. Finally, we would like to thank our shepherd, Bruce Maggs. This material is based upon work supported by the National Science Foundation under Grants No. IIS-0537252 and CNS-0448349 and by the Center of Information Assurance and Security at University of Texas at Austin.

References

[1] http://spinroot.com/spin/whatispin.html.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proc NSDI*, May 2006.

[3] N. Belaramani, J. Zheng, A. Nayate, R. Soule, M. Dahlin, and R. Grimm. PADS: A Policy Architecture for Distributed Storage Systems. Technical Report TR-09-08, U. of Texas at Austin, Feb. 2009.

[4] S. Chandra, M. Dahlin, B. Richards, R. Wang, T. Anderson, and J. Larus. Experience with a Language for Writing Coherence Protocols. In *USENIX Conf. on Domain-Specific Lang.*, Oct. 1997.

[5] M. Dahlin, R. Wang, T. Anderson, and D. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proc. OSDI*, pages 267–280, Nov. 1994.

[6] M. Demmer, B. Du, and E. Brewer. TierStore: a distributed storage system for challenged networks. In *Proc. FAST*, Feb. 2008.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of Consistent, Available, Partition-tolerant web services. In *ACM SIGACT News, 33(2)*, Jun 2002.

[8] C. Gray and D. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. In *SOSP*, pages 202–210, 1989.

[9] R. Grimm. Better extensibility through modular syntax. In *Proc. PLDI*, pages 38–51, June 2006.

[10] J. Heidemann and G. Popek. File-system development with stackable layers. *ACM TOCS*, 12(1):58–89, Feb. 1994.

[11] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and Performance in a Distributed File System. *ACM TOCS*, 6(1):51–81, Feb. 1988.

[12] A. Karypidis and S. Lalis. Omnistore: A system for ubiquitous personal storage management. In *PERCOM*, pages 136–147. IEEE CS Press, 2006.

[13] A. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube aproach to the reconciliation of divergent replicas. In *PODC*, 2001.

[14] J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM TOCS*, 10(1):3–25, Feb. 1992.

[15] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The Click modular router. *ACM TOCS*, 18(3):263–297, Aug. 2000.

[16] R. Lipton and J. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton, 1988.

[17] B. Loo, T. Condie, J. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *SOSP*, Oct. 2005.

[18] P. Mahajan, S. Lee, J. Zheng, L. Alvisi, and M. Dahlin. Astro: Autonomous and trustworthy data sharing. Technical Report TR-08-24, The University of Texas at Austin, Oct. 2008.

[19] D. Malkhi and D. Terry. Concise version vectors in WinFS. In *Symp. on Distr. Comp. (DISC)*, 2005.

[20] A. Nayate, M. Dahlin, and A. Iyengar. Transparent information dissemination. In *Proc. Middleware*, Oct. 2004.

[21] E. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proc. OSDI*, Dec. 2004.

[22] N.Tolia, M. Kozuch, and M. Satyanarayanan. Integrating portable and distributed storage. In *Proc. FAST*, pages 227–238, 2004.

[23] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *SOSP*, Oct. 1997.

[24] V. Ramasubramanian, T. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. Technical report, Microsoft Research, 2008.

[25] A. Rodriguez, C. Killian, S. Bhat, D. Kostic, and A. Vahdat. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *Proc NSDI*, 2004.

[26] Y. Saito, C. Karamanolis, M. Karlsson, and M. Mahalingam. Taming aggressive replication in the Pangaea wide-area file system. In *Proc. OSDI*, Dec. 2002.

[27] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for distributed workstation environments. *IEEE Trans. Computers*, 39(4), 1990.

[28] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. OPODIS*, Dec. 2004.

[29] A. Siegel, K. Birman, and K. Marzullo. Deceit: A flexible distributed file system. Corenell TR 89-1042, 1989.

[30] S. Sobti, N. Garg, F. Zheng, J. Lai, E. Ziskind, A. Krishnamurthy, and R. Y. Wang. Segank: a distributed mobile storage system. In *Proc. FAST*, pages 239–252. USENIX Association, 2004.

[31] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*, Dec. 1995.

[32] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *Proc. OSDI*, Dec. 2004.

[33] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[34] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. In *SOSP*, 2001.

[35] J. Zheng, N. Belaramani, and M. Dahlin. Pheme: Synchronizing replicas in diverse environments. Technical Report TR-09-07, U. of Texas at Austin, Feb. 2009.

Sora: High Performance Software Radio Using General Purpose Multi-core Processors

Kun Tan[†] Jiansong Zhang[†] Ji Fang [‡] He Liu [§] Yusheng Ye[§]
Shen Wang[§] Yongguang Zhang[†] Haitao Wu[†] Wei Wang[†] Geoffrey M. Voelker[♮]

[†]*Microsoft Research Asia, Beijing, China* [§] *Tsinghua University, Beijing, China*
[‡] *Beijing Jiaotong University, Beijing, China* [♮] *UCSD, La Jolla, USA*

Abstract

This paper presents Sora, a fully programmable software radio platform on commodity PC architectures. Sora combines the performance and fidelity of hardware SDR platforms with the programmability and flexibility of general-purpose processor (GPP) SDR platforms. Sora uses both hardware and software techniques to address the challenges of using PC architectures for high-speed SDR. The Sora hardware components consist of a radio front-end for reception and transmission, and a radio control board for high-throughput, low-latency data transfer between radio and host memories. Sora makes extensive use of features of contemporary processor architectures to accelerate wireless protocol processing and satisfy protocol timing requirements, including using dedicated CPU cores, large low-latency caches to store lookup tables, and SIMD processor extensions for highly efficient physical layer processing on GPPs. Using the Sora platform, we have developed a demonstration radio system called SoftWiFi. SoftWiFi seamlessly interoperates with commercial 802.11a/b/g NICs, and achieves equivalent performance as commercial NICs at each modulation.

1 Introduction

Software defined radio (SDR) holds the promise of fully programmable wireless communication systems, effectively supplanting current technologies which have the lowest communication layers implemented primarily in fixed, custom hardware circuits. Realizing the promise of SDR in practice, however, has presented developers with a dilemma.

Many current SDR platforms are based on either programmable hardware such as field programmable gate arrays (FPGAs) [6, 11] or embedded digital signal processors (DSPs) [5, 13]. Such hardware platforms can meet the processing and timing requirements of modern high-speed wireless protocols, but programming FPGAs and specialized DSPs are difficult tasks. Developers have to learn how to program to each particular embedded architecture, often without the support of a rich development environment of programming and debugging tools. Hardware platforms can also be expensive; the WARP [6] educational price, for example, is over US$9,750.

In contrast, SDR platforms based on general-purpose processor (GPP) architectures, such as commodity PCs, have the opposite set of tradeoffs. Developers program to a familiar architecture and environment using sophisticated tools, and radio front-end boards for interfacing with a PC are relatively inexpensive. However, since PC hardware and software have not been designed for wireless signal processing, existing GPP-based SDR platforms can achieve only limited performance [1, 22]. For example, the popular GNU Radio platform [1] achieves only a few Kbps throughput on an 8MHz channel [21], whereas modern high-speed wireless protocols like 802.11 support multiple Mbps data rates on a much wider 20MHz channel [7]. These constraints prevent developers from using such platforms to achieve the full fidelity of state-of-the-art wireless protocols while using standard operating systems and applications in a real environment.

In this paper we present Sora, a fully programmable software radio platform that provides the benefits of both SDR approaches, thereby resolving the SDR platform dilemma for developers. With Sora, developers can implement and experiment with high-speed wireless protocol stacks, *e.g.*, IEEE 802.11a/b/g, using commodity general-purpose PCs. Developers program in familiar programming environments with powerful tools on standard operating systems. Software radios implemented on Sora appear like any other network device, and users can run unmodified applications on their software radios with the same performance as commodity hardware wireless devices.

An implementation of high-speed wireless protocols on general-purpose PC architectures must overcome a number of challenges that stem from existing hardware interfaces and software architectures. First, transferring high-fidelity digital waveform samples into PC memory for processing requires very high bus throughput. Existing GPP platforms like GNU Radio use USB 2.0 or

This work was performed when Ji Fang, He Liu, Yusheng Ye, and Shen Wang were visiting students and Geoffrey M. Voelker was a visiting researcher at Microsoft Research Asia.

Gigabit Ethernet [1], which cannot satisfy this requirement for high-speed wireless protocols. Second, physical layer (PHY) signal processing has very high computational requirements for generating information bits from waveforms, and vice versa, particularly at high modulation rates; indeed, back-of-the-envelope calculations for processing requirements on GPPs have instead motivated specialized hardware approaches in the past [17, 19]. Lastly, wireless PHY and media access control (MAC) protocols have low-latency real-time deadlines that must be met for correct operation. For example, the 802.11 MAC protocol requires precise timing control and ACK response latency on the order of tens of microseconds. Existing software architectures on the PC cannot consistently meet this timing requirement.

Sora uses both hardware and software techniques to address the challenges of using PC architectures for high-speed SDR. First, we have developed a new, inexpensive radio control board (RCB) with a radio front-end for transmission and reception. The RCB bridges an RF front-end with PC memory over the high-speed and low-latency PCIe bus [8]. With this bus standard, the RCB can support 16.7Gbps (x8 mode) throughput with sub-microsecond latency, which together satisfies the throughput and timing requirements of modern wireless protocols while performing all digital signal processing on host CPU and memory.

Second, to meet PHY processing requirements, Sora makes full use of various features of widely adopted multi-core architectures in existing GPPs. The Sora software architecture also explicitly supports streamlined processing that enables components of the signal processing pipeline to efficiently span multiple cores. Further, we change the conventional implementation of PHY components to extensively take advantage of lookup tables (LUTs), trading off computation for memory. These LUTs substantially reduce the computational requirements of PHY processing, while at the same time taking advantage of the large, low-latency caches on modern GPPs. Finally, Sora uses the SIMD (Single Instruction Multiple Data) extensions in existing processors to further accelerate PHY processing. With these optimizations, Sora can fully support the complete digital processing of 802.11b modulation rates on just one core, and 802.11a/g on two cores.

Lastly, to meet the real-time requirements of high-speed wireless protocols, Sora provides a new kernel service, *core dedication*, which allocates processor cores exclusively for real-time SDR tasks. We demonstrate that it is a simple yet crucial abstraction that guarantees the computational resources and precise timing control necessary for SDR on a GPP.

We have developed a demonstration radio system, SoftWiFi, based on the Sora platform. SoftWiFi currently supports the full suite of 802.11a/b/g modulation rates, seamlessly interoperates with commercial 802.11 NICs, and achieves equivalent performance as commercial NICs at each modulation.

In summary, the contributions of this paper are: (1) the design and implementation of the Sora platform and its high-performance PHY processing library; (2) the design and implementation of the SoftWiFi radio system that can interoperate with commercial wireless NICs using 802.11a/b/g standards; and (3) the evaluation of Sora and SoftWiFi on a commodity multi-core PC. To the best of our knowledge, Sora is the first SDR platform that enables users to develop high-speed wireless implementations, such as the IEEE 802.11a/b/g PHY and MAC, entirely in software on a standard PC architecture.

The rest of the paper is organized as follows. Section 2 provides background on wireless communication systems. We then present the Sora architecture in Section 3, and we discuss our approach for addressing the challenges of building an SDR platform on a GPP system in Section 4. We then describe the implementation of the Sora platform in Section 5. Section 6 presents the design and implementation of SoftWiFi, a fully functional software WiFi radio based on Sora, and we evaluate its performance in Section 7. Finally, Section 9 describes related work and Section 10 concludes.

2 Background and Requirements

In this section, we briefly review the physical layer (PHY) and media access (MAC) components of typical wireless communication systems. Although different wireless technologies may have subtle differences among one another, they generally follow similar designs and share many common algorithms. In this section, we use the IEEE 802.11a/b/g standards to exemplify characteristics of wireless PHY and MAC components as well as the challenges of implementing them in software.

2.1 Wireless PHY

The role of the PHY layer is to convert information bits into a radio waveform, or vice versa. At the transmitter side, the wireless PHY component first *modulates* the message (*i.e.*, a packet or a MAC frame) into a time sequence of *baseband signals*. Baseband signals are then passed to the radio front-end, where they are multiplied by a high frequency *carrier* and transmitted into the wireless channel. At the receiver side, the radio front-end detects signals in the channel and extracts the baseband signal by removing the high-frequency carrier. The extracted baseband signal is then fed into the receiver's PHY layer to be *demodulated* into the original message.

Advanced communication systems (*e.g.*, IEEE 802.11a/b/g, as shown in Figure 1) contain multiple

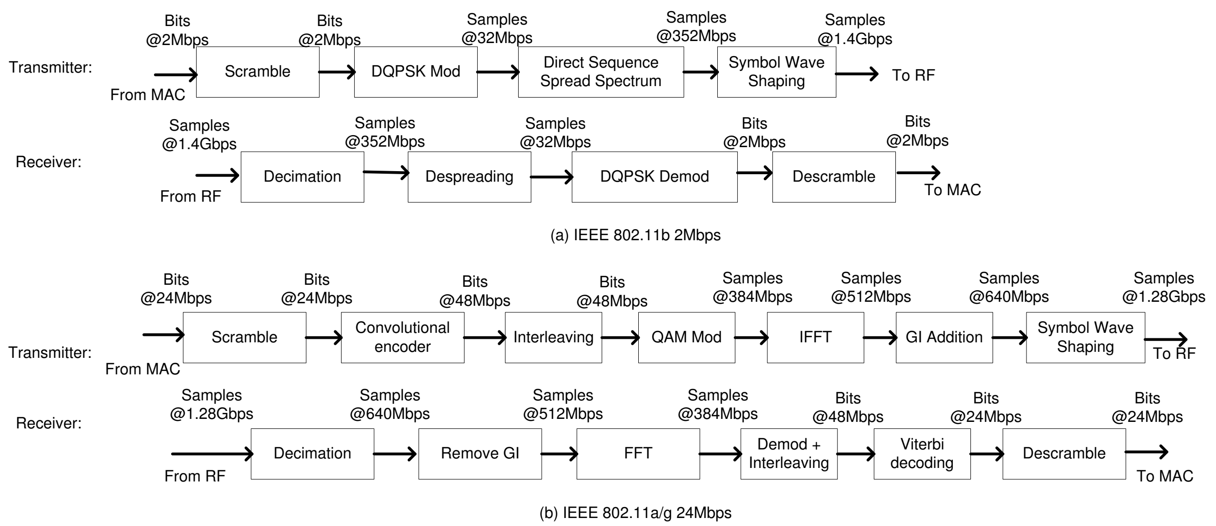


Figure 1: PHY operations of IEEE 802.11a/b/g transceiver.

functional blocks in their PHY components. These functional blocks are pipelined with one another. Data are streamed through these blocks sequentially, but with different data types and sizes. As illustrated in Figure 1, different blocks may consume or produce different types of data in different rates arranged in small data blocks. For example, in 802.11b, the scrambler may consume and produce one bit, while DQPSK modulation maps each two-bit data block onto a complex symbol which uses two 16-bit numbers to represent the in-phase and quadrature (I/Q) components.

Each PHY block performs a fixed amount of computation on every transmitted or received bit. When the data rate is high, *e.g.*, 11Mbps for 802.11b and 54Mbps for 802.11a/g, PHY processing blocks consume a significant amount of computational power. Based on the model in [19], we estimate that a direct implementation of 802.11b may require 10Gops while 802.11a/g needs at least 40Gops. These requirements are very demanding for software processing in GPPs.

PHY processing blocks directly operate on the digital waveforms after modulation on the transmitter side and before demodulation on the receiver side. Therefore, high-throughput interfaces are needed to connect these processing blocks as well as to connect the PHY and radio front-end. The required throughput linearly scales with the bandwidth of the baseband signal. For example, the channel bandwidth is 20MHz in 802.11a. It requires a data rate of at least 20M complex samples per second to represent the waveform [14]. These complex samples normally require 16-bit quantization for both I and Q components to provide sufficient fidelity, translating into 32 bits per sample, or 640Mbps for the full 20MHz channel. Over-sampling, a technique widely used for better performance [12], doubles the requirement to 1.28Gbps to move data between the RF frond-end and PHY blocks for one 802.11a channel.

2.2 Wireless MAC

The wireless channel is a resource shared by all transceivers operating on the same spectrum. As simultaneously transmitting neighbors may interfere with each other, various MAC protocols have been developed to coordinate their transmissions in wireless networks to avoid collisions.

Most modern MAC protocols, such as 802.11, require timely responses to critical events. For example, 802.11 adopts a CSMA (Carrier-Sense Multiple Access) MAC protocol to coordinate transmissions [7]. Transmitters are required to sense the channel before starting their transmission, and channel access is only allowed when no energy is sensed, *i.e.*, the channel is free. The latency between *sense* and *access* should be as small as possible. Otherwise, the sensing result could be outdated and inaccurate. Another example is the link-layer retransmission mechanisms in wireless protocols, which may require an immediate acknowledgement (ACK) to be returned in a limited time window.

Commercial standards like IEEE 802.11 mandate a response latency within tens of microseconds, which is challenging to achieve in software on a general purpose PC with a general purpose OS.

2.3 Software Radio Requirements

Given the above discussion, we summarize the requirements for implementing a software radio system on a general PC platform:

High system throughput. The interfaces between the radio front-end and PHY as well as between some PHY processing blocks must possess sufficiently high

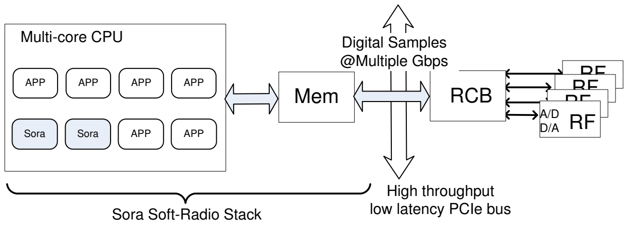


Figure 2: Sora system architecture. All PHY and MAC execute in software on a commodity multi-core CPU.

throughput to transfer high-fidelity digital waveforms. To support a 20MHz channel for 802.11, the interfaces must sustain at least 1.28Gbps. Conventional interfaces like USB 2.0 ($\leq$ 480Mbps) or Gigabit Ethernet ($\leq$ 1Gbps) cannot meet this requirement [1].

Intensive computation. High-speed wireless protocols require substantial computational power for their PHY processing. Such computational requirements also increase proportionally with communication speed. Unfortunately, techniques used in conventional PHY hardware or embedded DSPs do not directly carry over to GPP architectures. Thus, we require new software techniques to accelerate high-speed signal processing on GPPs. With the advent of many-core GPP architectures [9], it is now reasonable to dedicate computational power solely to signal processing. But, it is still challenging to build a software architecture to efficiently exploit the full capability of multiple cores.

Real-time enforcement. Wireless protocols have multiple *real-time deadlines* that need to be met. Consequently, not only is processing throughput a critical requirement, but the processing latency needs to meet response deadlines. Some MAC protocols also require precise timing control at the granularity of microseconds to ensure certain actions occur at exactly pre-scheduled time points. Meeting such real-time deadlines on a general PC architecture is a non-trivial challenge: time sharing operation systems may not respond to an event in a timely manner, and bus interfaces, such as Gigabit Ethernet, could introduce indefinite delays far more than a few μs. Therefore, meeting these real-time requirements requires new mechanisms on GPPs.

3 Architecture

We have developed a high-performance software radio platform called Sora that addresses these challenges. It is based on a commodity general-purpose PC architecture. For flexibility and programmability, we push as much communication functionality as possible into software, while keeping hardware additions as simple and generic as possible. Figure 2 illustrates the overall system architecture.

3.1 Hardware Components

The hardware components in the Sora architecture are a new radio control board (RCB) with an interchangeable radio front-end (RF front-end). The radio front-end is a hardware module that receives and/or transmits radio signals through an antenna. In the Sora architecture, the RF front-end represents the well-defined interface between the digital and analog domains. It contains analog-to-digital (A/D) and digital-to-analog (D/A) converters, and necessary circuitry for radio transmission. During receiving, the RF front-end acquires an analog waveform from the antenna, possibly down-converts it to a lower frequency, and then digitizes it into discrete samples before transferring them to the RCB. During transmitting, the RF front-end accepts a synchronous stream of software-generated digital samples and synthesizes the corresponding analog waveform before emitting it using the antenna. Since all signal processing is done in software, the RF front-end design can be rather generic. It can be implemented in a self-contained module with a standard interface to the RCB. Multiple wireless technologies defined on the same frequency band can use the same RF front-end hardware, and the RCB can connect to different RF front-ends designed for different frequency bands.

The RCB is a new PC interface board for establishing a high-throughput, low-latency path for transferring high-fidelity digital signals between the RF front-end and PC memory. To achieve the required system throughput discussed in Section 2.1, the RCB uses a high-speed, low-latency bus such as PCIe [8]. With a maximum throughput of 64Gbps (PCIe x32) and sub-microsecond latency, it is well-suited for supporting multiple gigabit data rates for wireless signals over a very wide band or over many MIMO channels. Further, the PCIe interface is now common in contemporary commodity PCs.

Another important role of the RCB is to bridge the synchronous data transmission at the RF front-end and the asynchronous processing on the host CPU. The RCB uses various buffers and queues, together with a large on-board memory, to convert between synchronous and asynchronous streams and to smooth out bursty transfers between the RCB and host memory. The large on-board memory further allows caching pre-computed waveforms, adding additional flexibility for software radio processing.

Finally, the RCB provides a low-latency control path for software to control the RF front-end hardware and to ensure it is properly synchronized with the host CPU. Section 5.1 describes our implementation of the RCB in more detail.

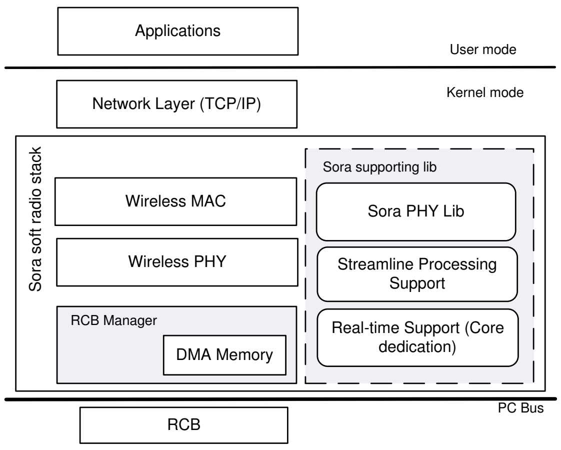


Figure 3: Software architecture of Sora soft-radio stack.

3.2 Sora Software

Figure 3 illustrates Sora's software architecture. The software components in Sora provide necessary system services and programming support for implementing various wireless PHY and MAC protocols in a general-purpose operating system. In addition to facilitating the interaction with the RCB, the Sora soft-radio stack provides a set of techniques to greatly improve the performance of PHY and MAC processing on GPPs. To meet the processing and real-time requirements, these techniques make full use of various common features in existing multi-core CPU architectures, including the extensive use of lookup tables (LUTs), substantial data-parallelism with CPU SIMD extensions, the efficient partitioning of streamlined processing over multiple cores, and exclusive dedication of cores for software radio tasks.

4 High-Performance SDR Processing

In this section we describe the software techniques used by Sora to achieve high-performance SDR processing.

4.1 Efficient PHY processing

In a memory-for-computation tradeoff, Sora relies upon the large-capacity, high-speed cache memory in GPPs to accelerate PHY processing with pre-calculated lookup tables (LUTs). Contemporary modern CPU architectures, such as Intel Core 2, usually have megabytes of L2 cache with a low (10$\sim$20 cycles) access latency. If we pre-calculate LUTs for a large portion of PHY algorithms, we can greatly reduce the computational requirement for on-line processing.

For example, the *soft demapper* algorithm used in demodulation needs to calculate the *confidence level* of each bit contained in an incoming symbol. This task involves rather complex computation proportional to the modulation density. More precisely, it conducts an extensive search for all modulation points in a constellation graph and calculates a ratio between the minimum of Euclidean distances to all points representing one and the minimum of distances to all points representing zero. In this case, we can pre-calculate the confidence levels for all possible incoming symbols based on their I and Q values, and build LUTs to directly map the input symbol to confidence level. Such LUTs are not large. For example, in 802.11a/g with a 54Mbps modulation rate (64-QAM), the size of the LUT for the soft demapper is only 1.5KB.

As we detail later in Section 5.2.1, more than half of the common PHY algorithms can indeed be rewritten with LUTs, each with a speedup from 1.5x to 50x. Since the size of each LUT is sufficiently small, the sum of all LUTs in a processing path can easily fit in the L2 caches of contemporary GPP cores. With *core dedication* (Section 4.3), the possibility of cache collisions is very small. As a result, these LUTs are almost always in caches during PHY processing.

To accelerate PHY processing with data-level parallelism, Sora heavily uses the SIMD extensions in modern GPPs, such as SSE, 3DNow!, and AltiVec. Although these extensions were designed for multimedia and graphics applications, they also match the needs of wireless signal processing very well because many PHY algorithms have fixed computation structures that can easily map to large vector operations. In Appendix A, we show an example of an optimized digital filter implementation using SSE instructions. As our measurements later show, such SIMD extensions substantially speed up PHY processing in Sora.

4.2 Multi-core streamline processing

Even with the above optimizations, a single CPU core may not have sufficient capacity to meet the processing requirements of high-speed wireless communication technologies. As a result, Sora must be able to use more than one core in a multi-core CPU for PHY processing. This multi-core technique should also be scalable because the signal processing algorithms may become increasingly more complex as wireless technologies progress.

As discussed in Section 2, PHY processing typically contains several functional blocks in a pipeline. These blocks differ in processing speed and in input/output data rates and units. A block is only *ready* to execute when it has sufficient input data from the previous block. Therefore, a key issue is how to schedule a functional block on multiple cores when it is ready.

One possible approach is to run multiple PHY pipelines on different cores (Figure 4(a)), and have the scheduler dispatch batches of digital samples to a

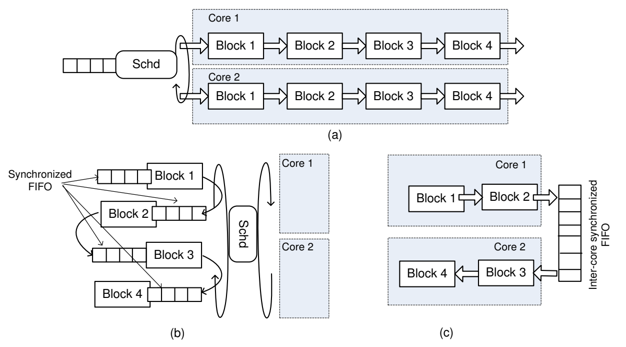


Figure 4: PHY pipeline scheduling: (a) parallel pipelines, (b) dynamic scheduling, (c) static scheduling.

pipeline. This approach, however, does not work well for SDR because wireless communication has strong dependencies in a data stream. For example, in convolutional encoding the output of each bit also depends on the seven preceding bits in the input stream. Without the scheduler knowing all of the data dependencies, it is difficult to produce an efficient schedule.

An alternative scheduling approach is to have only one pipeline and dynamically assign ready blocks to available cores (Figure 4(b)), in a way similar to thread scheduling in a multi-core system. Unfortunately, this approach would introduce prohibitively high overhead. On the one hand, any two adjacent blocks may be scheduled onto two different cores, thereby requiring synchronized FIFO (SFIFO) communication between them. On the other hand, most PHY processing blocks operate on very small data items, *e.g.*, 1–4 bytes each, and the processing only takes a few operations (several to tens of instructions). Such frequent FIFO and synchronization operations are not justifiable for such small computational tasks.

Instead, Sora chooses a static scheduling scheme. This decision is based on the observation that the schedule of each block in a PHY processing pipeline is actually static: the processing pattern of previous blocks can determine whether a subsequent block is ready or not. Sora can thus partition the whole PHY processing pipeline into several sub-pipelines and statically assign them to different cores (Figure 4(c)). Within one sub-pipeline, when a block has accumulated enough data for the next block to be ready, it explicitly schedules the next block. Adjacent sub-pipelines from different blocks are still connected with an SFIFO, but the number of SFIFOs and their overhead are greatly reduced.

4.3 Real-time support

SDR processing is a time-critical task that requires strict guarantees of computational resources and hard real-time deadlines. As an alternative to relying upon the full generality of real-time operating systems, we can achieve real-time guarantees by simply dedicating cores to SDR processing in a multi-core system. Thus, sufficient computational resources can be guaranteed without being affected by other concurrent tasks in the system.

This approach is particularly plausible for SDR. First, wireless communication often requires its PHY to constantly monitor the channel for incoming signals. Therefore, the PHY processing may need to be active all the time. It is much better to always schedule this task on the same core to minimize overhead like cache misses or TLB flushes. Second, previous work on multi-core OSes also suggests that isolating applications into different cores may have better performance compared to symmetric scheduling, since an effective use of cache resources and a reduction in locks can outweigh dedicating cores [10]. Moreover, a *core dedication* mechanism is much easier to implement than a real-time scheduler, sometimes even without modifying an OS kernel. For example, we can simply raise the priority of a kernel thread so that it is pinned on a core and it exclusively runs until termination (Section 5.2.3).

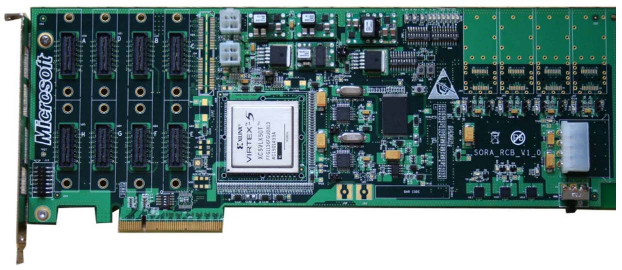


Figure 5: Sora radio control board.

5 Implementation

We have implemented both the hardware and software components of Sora. This section describes our hardware prototype and software stack, and presents microbenchmark evaluations of Sora components.

5.1 Hardware

We have designed and implemented the Sora radio control board (RCB) as shown in Figure 5. It contains a Virtex-5 FPGA, a PCIe-x8 interface, and 256MB of DDR2 SDRAM. The RCB can connect to various RF front-ends. In our experimental prototype, we use a third-party RF front-end, developed by Rice University [6], that is capable of transmitting and receiving a 20MHz channel at 2.4GHz or 5GHz.

Figure 6 illustrates the logical components of the Sora hardware platform. The DMA and PCIe controllers interface with the host and transfer digital samples between the RCB and PC memory. Sora software sends commands and reads RCB states through RCB regis-

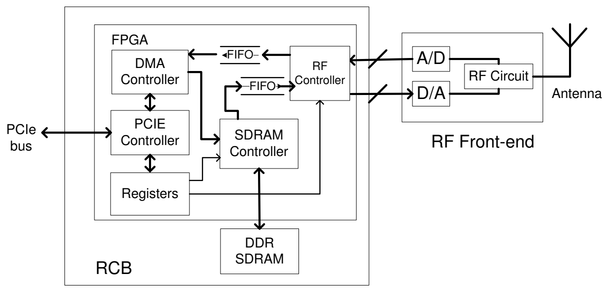


Figure 6: Hardware architecture of RCB and RF.

ters. The RCB uses its on-board SDRAM as well as small FIFOs on the FPGA chip to bridge data streams between the CPU and RF front-end. When receiving, digital signal samples are buffered in on-chip FIFOs and delivered into PC memory when they fit in a DMA burst (128 bytes). When transmitting, the large RCB memory enables Sora software to first write the generated samples onto the RCB, and then trigger transmission with another command to the RCB. This functionality provides flexibility to the Sora software for pre-calculating and storing several waveforms before actually transmitting them, while allowing precise control of the timing of the waveform transmission.

While implementing Sora, we encountered a consistency issue in the interaction between DMA operations and the CPU cache system. When a DMA operation modifies a memory location that has been cached in the L2 cache, it does not invalidate the corresponding cache entry. When the CPU reads that location, it can therefore read an incorrect value from the cache. One naive solution is to disable cached accesses to memory regions used for DMA, but doing so will cause a significant degradation in memory access throughput.

We solve this problem with a *smart-fetch* strategy, enabling Sora to maintain cache coherency with DMA memory without drastically sacrificing throughput. First, Sora organizes DMA memory into small slots, whose size is a multiple of a cache line. Each slot begins with a *descriptor* that contains a flag. The RCB sets the flag after it writes a full slot of data, and cleared after the CPU processes all data in the slot. When the CPU moves to a new slot, it first reads its descriptor, causing a whole cache line to be filled. If the flag is set, the data just fetched is valid and the CPU can continue processing the data. Otherwise, the RCB has not updated this slot with new data. Then, the CPU explicitly flushes the cache line and repeats reading the same location. This next read refills the cache line, loading the most recent data from memory.

5.2 Software

The Sora software is written in C, with some assembly for performance-critical processing. The entire Sora software stack is implemented on Windows XP as a network device driver and it exposes a virtual Ethernet interface to the upper TCP/IP stack. Since any software radio implemented on Sora can appear as a normal network device, all existing network applications can run unmodified on it.

The Sora software currently consists of 23,325 non-blank lines of C code. Of this total, 14,529 lines are for system support, including driver framework, memory management, streamline processing, etc. The remaining 8,796 lines comprise the PHY processing library.

5.2.1 PHY processing library

In the Sora PHY processing library, we extensively exploit the use of look-up tables (LUTs) and SIMD instructions to optimize the performance of PHY algorithms. We have been able to rewrite more than half of the PHY algorithms with LUTs. Some LUTs are straightforward pre-calculations, others require more sophisticated implementations to keep the LUT size small. For the soft-demapper example mentioned earlier, we can greatly reduce the LUT size (*e.g.*, 1.5KB for the 802.11a/g 54Mbps modulation) by exploiting the symmetry of the algorithm. In our SoftWiFi implementation described below, the overall size of the LUTs used in 802.11a/g is around 200KB and 310KB in 802.11b, both of which fit comfortably within the L2 caches of commodity CPUs.

We also heavily use SIMD instructions in coding Sora software. We currently use the SSE2 instruction set designed for Intel CPUs. Since the SSE registers are 128-bit wide while most PHY algorithms require only 8-bit or 16-bit fixed-point operations, one SSE instruction can perform 8 or 16 simultaneous calculations. SSE2 also has rich instruction support for flexible data permutations, and most PHY algorithms, *e.g.*, FFT, FIR Filter and Viterbi, can fit naturally into this SIMD model. For example, the Sora Viterbi decoder uses only 40 cycles to compute the branch metric and select the shortest path for each input. As a result, our Viterbi implementation can handle 802.11a/g at the 54Mbps modulation with only one 2.66GHz CPU core, whereas previous implementations relied on hardware implementations. Note that other GPP architectures, like AMD and PowerPC, have very similar SIMD models and instruction sets; AMD's Enhanced 3DNow!, for instance, includes SSE instructions plus a set of DSP extensions. We expect that our optimization techniques will directly apply to these other GPP architectures as well. In Appendix A, we show a simple example of a functional block using SIMD instruction optimizations.

Table 1 summarizes some key PHY processing algorithms we have implemented in Sora, together with the optimization techniques we have applied. The table also

Algorithm	Configuration	I/O Size (bit)		Optimization Method	Computation Required (Mcycles/sec)		
		Input	Output		Conv. Impl.	Sora Impl.	*Speedup*
IEEE 802.11b							
Scramble	11Mbps	8	8	LUT	96.54	10.82	8.9x
Descramble	11Mbps	8	8	LUT	95.23	5.91	16.1x
Mapping and Spreading	2Mbps, DQPSK	8	44*16*2	LUT	128.59	73.92	1.7x
CCK modulator	5Mbps, CCK	8	8*16*2	LUT	124.93	81.29	1.5x
	11Mbps, CCK	8	8*16*2	LUT	203.96	110.88	1.8x
FIR Filter	16-bit I/Q, 37 taps, 22MSps	16*2*4	16*2*4	SIMD	5,780.34	616.41	9.4x
Decimation	16-bit I/Q, 4x Oversample	16*2*4*4	16*2*4	SIMD	422.45	198.72	2.1x
IEEE 802.11a							
FFT/IFFT	64 points	64*16*2	64*16*2	SIMD	754.11	459.52	1.6x
Conv. Encoder	24Mbps, 1/2 rate	8	16	LUT	406.08	18.15	22.4x
	48Mbps, 2/3 rate	16	24	LUT	688.55	37.21	18.5x
	54Mbps, 3/4 rate	24	32	LUT	712.10	56.23	12.7x
Viterbi	24Mbps, 1/2 rate	8*16	8	SIMD+LUT	68,553.57	1,408.93	48.7x
	48Mbps, 2/3 rate	8*24	16	SIMD+LUT	117,199.6	2,422.04	48.4x
	54Mbps, 3/4 rate	8*32	24	SIMD+LUT	131,017.9	2,573.85	50.9x
Soft demapper	24Mbps, QAM 16	16*2	8*4	LUT	115.05	46.55	2.5x
	54Mbps, QAM 64	16*2	8*6	LUT	255.86	98.75	2.4x
Scramble & Descramble	54Mbps	8	8	LUT	547.86	40.29	13.6x

Table 1: Key algorithms in IEEE 802.11b/a and their performance with conventional and Sora implementations.

compares the performance of a conventional software implementation (*e.g.*, a direct translation from a hardware implementation) and the Sora implementation with the LUT and SIMD optimizations.

5.2.2 Lightweight, synchronized FIFOs

Sora allows different PHY processing blocks to streamline across multiple cores while communicating with one another through shared memory FIFO queues. If two blocks are running on different cores, their access to the shared FIFO must be synchronized. The traditional implementation of a synchronized FIFO uses a *counter* to synchronize the writer and reader, which we refer to as a counter-based FIFO (CBFIFO) and illustrate in Figure 7(a). However, this counter is shared by two processor cores, and every write to the variable by one core will cause a cache miss on the other core. Since both the producer and consumer modify this variable, two cache misses are unavoidable for each datum. It is also quite common to have very fine data granularity in PHY (*e.g.*, 4–16 bytes as summarized in Table 1). Therefore, such cache misses will result in significant overhead when synchronization has to be performed very frequently (*e.g.*, once per micro-second) for such small pieces of data.

```
// producer:
void write_fifo ( DATA_TYPE data ) {
  while (cnt >= q_size); // spin wait
  q[w_tail] = data;
  w_tail = (w_tail+1) % q_size;
  InterlockedIncrement (cnt); // increase cnt by 1
 }
// consumer:
void read_fifo ( DATA_TYPE * pdata ) {
  while (cnt==0); // spin wait
  * pdata = q[r_head];
  r_head = (r_head+1) % q_size;
  InterlockedDecrement(cnt); // decrease cnt by 1
 }
```

(a)

```
// producer:
void write_fifo ( DATA_TYPE data ) {
  while (q[w_tail].flag>0); // spin wait
  q[w_tail].data = data;
  q[w_tail].flag = 1; // occupied
  w_tail = (w_tail+1) % q_size;
}
// consumer:
void read_fifo ( DATA_TYPE * pdata ) {
  while (q[r_head].flag==0); // spin
  *data = q[r_head].data;
  q[r_head].flag = 0; // release
  r_head = (r_head + 1) % q_size;
}
```

(b)

Figure 7: Pseudo-code for synchronized (a) CBFIFOs and (b) CPFIFOs.

In Sora, we implement another synchronized FIFO that removes the sole shared synchronization variable. The idea is to augment each data slot in the FIFO with a header that indicates whether the slot is empty or not. We pad each data slot to be a multiple of a cache line. Thus, the consumer is always chasing the producer in the circular buffer for filled slots, as outlined in Figure 7(b). This chasing-pointer FIFO (CPFIFO) largely mitigates the overhead even for very fine-grained synchronization. If the speed of the producer and consumer is the same and the two pointers are separated by a particular offset (*e.g.*, two cache lines in the Intel architecture), no cache miss will occur during synchronized streaming since the local cache will prefetch the following slots before the actual access. If the producer and the consumer have different processing speeds, *e.g.*, the reader is faster than the writer, then eventually the consumer will wait for the producer to release a slot. In this case, each time the producer writes to a slot, the write will cause a cache miss at the consumer. But the producer will not suffer

Mode	Rx (Gbps)	Tx (Gbps)
PCIe-x4	6.71	6.55
PCIe-x8	12.8	12.3

Table 2: DMA throughput performance of the RCB.

Method	Memory Throughput
Cache Disabled	707.2Mbps
Smart-fetch	10.1Gbps

Table 3: Memory throughput.

a miss since the next free slot will be prefetched into its local cache. Fortunately, such cache misses experienced by the consumer will not cause significant impact on the overall performance of the streamline processing since the consumer is not the bottleneck element.

5.2.3 Real-time support

Sora uses *exclusive threads* (or *ethreads*) to dedicate cores for real-time SDR tasks. Sora implements ethreads without any modification to the kernel code. An ethread is implemented as a kernel-mode thread, and it exploits the *processor affiliation* that is commonly supported in commodity OSes to control on which core it runs. Once the OS has scheduled the ethread on a specified physical core, it will raise its IRQL (interrupt request level) to a level as high as the kernel scheduler, *e.g.*, dispatch_level in Windows. Thus, the ethread takes control of the core and prevents itself from being preempted by other threads.

Running at such an IRQL, however, does not prevent the core from responding to hardware interrupts. Therefore, we also constrain the *interrupt affiliations* of all devices attached to the host. If an ethread is running on one core, all interrupt handlers for installed devices are removed from the core, thus prevent the core from being interrupted by hardware. To ensure the correct operation of the system, Sora always ensures core zero is able to respond to all hardware interrupts. Consequently, Sora only allows ethreads to run on cores whose ID is greater than zero.

5.3 Evaluation

We measure the performance of the Sora implementation with microbenchmark experiments. We perform all measurements on a Dell XPS PC with an Intel Core 2 Quad 2.66GHz CPU (Section 7.1 details the complete hardware configuration).

Throughput and latency. To measure PCIe throughput, we instruct the RCB to read/write a number of descriptors from/to main memory via DMA, and measure the time taken. Table 2 summarizes the results, which agree with the hardware specifications.

To precisely measure PCIe latency, we instruct the RCB to read a memory address in host memory. We measure the time interval between issuing the request and receiving the response data in hardware. Since the *memory read* operation accesses the PCIe bus using a round trip operation, we use half of the measured time to estimate the one-way delay. This one-way delay is $360ns$ with a worst case variation of $4ns$. We also confirm that the RCB hardware itself induces negligible delay except for buffers on the data path. However, such delay is tiny when the buffer is small. For example, the DMA burst size is 128 bytes, which causes only $76ns$ latency in PCIe-x8.

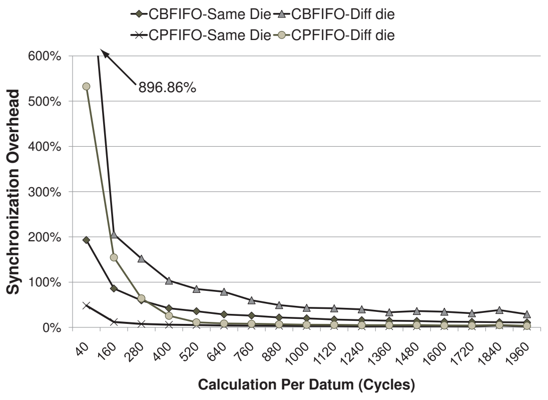


Figure 8: Overhead of synchronized FIFOs.

Table 3 compares measured memory throughput in two different cases. The first row shows the read throughput of uncacheable memory. It is only 707Mbps, which is insufficient for 802.11 processing. The second row shows the performance of the smart-fetch technique. With smart-fetch, the memory throughput is a factor of 14 greater compared to the uncacheable case, and sufficient for supporting high-speed protocol processing. We note, however, that it is still slower than reading from normal cacheable memory without having to be consistent with DMA operations. This reduction is due to the overhead of additional cache-line invalidations.

Synchronized FIFO. To measure the overhead of the synchronized CBFIFO and CPFIFO implementations, we process ten thousand data inputs through the FIFOs first on one core, and then on two cores. We also vary the number of cycles to process each datum to change the ratio of synchronization time with processing time. When processing with two cores, we allocate the same computation to each core. Denote t_1 and t_2 as the completion times of processing on one core and two cores, respectively. We then define the overhead of a synchronized FIFO as $\frac{t_2 - t_1/2}{t_1/2}$.

Figure 8 shows the results of this experiment. The x-axis shows the total processing cycles required for each datum, and the y-axis shows the overhead of the synchronized FIFO. We make following observations from these results. First, partitioning work across cores gives different overheads depending upon whether the cores are on the same die. Two cores on the same die share the same L2 cache, while cores on different dies are connected via a shared front-side bus. Thus, streaming data between functional blocks across cores on the same die has significantly less overhead than streaming between cores on different dies.

Second, the overhead decreases as the computation time per datum increases, as expected. When the computation per datum is very short, the communication overhead between cores dominates. The Intel CPU requires about 10 cycles to access its local L2 cache, and 100 cycles to access a remote cache. Therefore, when there are 40 cycles per datum, the overhead is at least $\frac{10}{20} = 50\%$ when two cores are on one die, and $\frac{100}{20} = 500\%$ when two cores are on different dies. The CPFIFO almost achieves this lower bound. When there is more computation required per datum, however, the data transfer can be overlapped with computation, enabling the overhead to be hidden. Finally, the CBFIFO generally has significantly higher overhead compared to the CPFIFO due to the additional synchronization overhead on the shared variable, which the CPFIFO avoids.

6 Case study: SoftWiFi

To demonstrate the use of Sora, we have developed a fully functional WiFi transceiver on the Sora platform called SoftWiFi. Our SoftWiFi stack supports all IEEE 802.11a/b/g modulations and can communicate seamlessly with commercial WiFi network cards.

Figure 9 illustrates the Sora SoftWiFi implementation. The MAC state machine (SM) is implemented as an ethread. Since 802.11 is a half-duplex radio, the demodulation components can run directly within a MAC SM thread. If a single core is insufficient for all PHY processing (*e.g.*, 802.11a/g), the PHY processing can be partitioned across two ethreads. These two ethreads are streamlined using a CPFIFO. An additional thread, *Snd_thread*, modulates the outgoing frames into waveform samples in the background. These modulated waveforms can be pre-stored in the RCB's memory to facilitate transmission. The *Completion_thread* monitors the *Rcv_buf* and notifies upper software layers of any correctly received frames. This thread also cleans up the *snd* and *rcv* buffers after they are used.

SoftWiFi implements the basic access mode of 802.11. The detailed MAC SM is shown in Figure 10. Normally, the SM is in the *Frame Detection* (FD) state. In that state, the RCB constantly writes samples into the *Rx_buf*. The SM continuously measures the average energy to determine whether the channel is clean or whether there is an incoming frame.

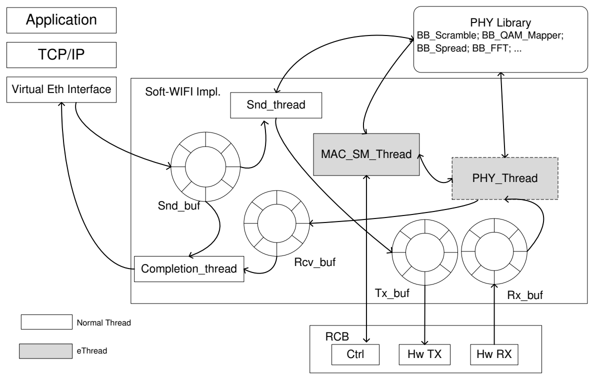


Figure 9: SoftWiFi implementation.

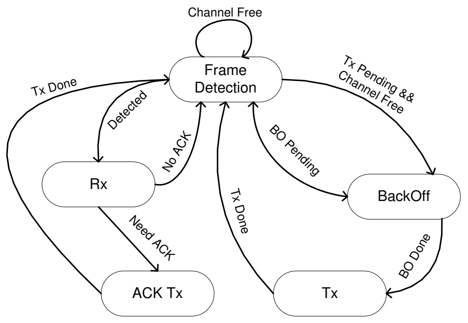


Figure 10: State machine of the SoftWiFi MAC.

The transmission of a frame follows the CSMA mechanism. When there is a pending frame, the SM first needs to check if the energy on the channel is low. If the channel is busy, the transmission should be deferred and a backoff timer started. Each time the channel becomes free, the SM checks if any backoff time remains. If the timer goes to zero, it transmits the frame.

SoftWiFi starts to receive a frame if it detects a high energy in the FD state. In 802.11, it takes three steps in the PHY layer to receive a frame. First, the PHY layer needs to synchronize to the frame, *i.e.*, find the starting point of the frame (timing synchronization) and the frequency offset and phase of the sample stream (carrier synchronization). Synchronization is usually done by correlating the incoming samples with a pre-defined preamble. Subsequently, the PHY layer needs to demodulate the PLCP (Physical Layer Convergence Protocol) header, which is always transmitted using a fixed low-rate modulation mode. The PLCP header contains the length of the frame as well as the modulation mode, possibly a higher rate, of the frame data that follows. Thus, only after successful reception of the PLCP header will the PHY layer know how to demodulate the remainder of the frame.

After successfully receiving a frame, the 802.11 MAC standard requires a station to transmit an ACK frame in a timely manner. For example, 802.11b requires that an

ACK frame be sent with a $10\mu s$ delay. However, this ACK requirement is quite difficult for an SDR implementation to achieve in software on a PC. Both generating and transferring the waveform across the PC bus will cause a latency of several microseconds, and the sum is usually larger than mandated by the standard. Fortunately, an ACK frame generally has a fixed pattern. For example, in 802.11 all data in an ACK frame is fixed except for the sender address of the corresponding data frame. Thus, in SoftWiFi, we can precalculate most of an ACK frame (19 bytes), and update only the address (10 bytes). Further, we can do it early in the processing, immediately after demodulating the MAC header, and without waiting for the end of a frame. We then prestore the waveform into the memory of the RCB. Thus, the time for ACK generation and transferring can overlap with the demodulation of the data frame. After the MAC SM demodulates the entire frame and validates the CRC32 checksum, it instructs the RCB to transmit the ACK, which has already been stored on the RCB. Thus, the latency for ACK transmission is very small.

In rare cases when the incoming data frame is quite small (*e.g.*, the frame contains only a MAC header and zero payload), then SoftWiFi cannot fully overlap ACK generation and the DMA transfer with demodulation to completely hide the latency. In this case, SoftWiFi may fail to send the ACK in time. We address this problem in SoftWiFi by maintaining a cache of previous ACKs in the RCB. With 802.11, all data frames from one node will have exactly the same ACK frame. Thus, we can use pre-allocated memory slots in the RCB to store ACK waveforms for different senders (we currently allocate 64 slots). Now, when demodulating a frame, if the ACK frame is already in the RCB cache, the MAC SM simply instructs the RCB to transmit the pre-cached ACK. With this scheme, SoftWiFi may be late on the first small frame from a sender, effectively dropping the packet from the sender's perspective. But retransmissions, and all subsequent transmissions, will find the appropriate ACK waveform already stored in the RCB cache.

We have implemented and tested the full 802.11a/g/b SoftWiFi tranceivers, which support DSSS (Direct Sequence Spreading: 1 and 2Mbps in 11b), CCK (Complementary Code Keying: 5.5 and 11Mbps in 11b), and OFDM (Orthogonal Frequency Division Multiplexing: 6, 9 and up to 54Mbps in 11a/g). It took one student about one month to develop and test 11b on Sora, and another student one and half months to code and test 11a/g; these efforts also include the time for implementing the corresponding algorithms in the PHY library.

7 Evaluations

In this section we evaluate the end-to-end application performance delivered by Sora. Our goals are to show that Sora interoperates seamlessly with commercial 802.11 devices, and that the Sora SoftWiFi implementation achieves equivalent performance. As a result, we show that Sora can process signals sufficiently fast to achieve full channel utilization, and that it can satisfy all timing requirements of the 802.11 standards with a software implementation on a GPP. We also characterize the CPU utilization of the software processing. In the following, we sometimes use the label 11a/g to present data for both 11a/g, since 11a and 11g have exactly the same OFDM PHY specification.

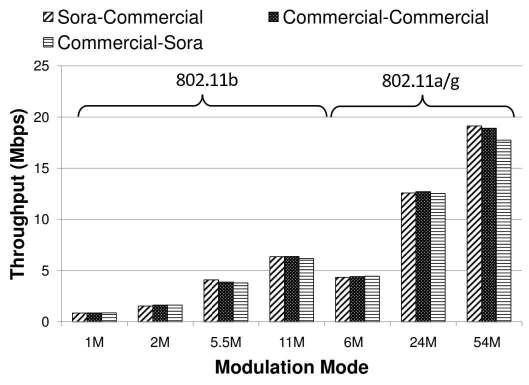


Figure 11: Throughput of Sora when communicating with a commercial WiFi card. *Sora–Commercial* presents the transmission throughput when a Sora node sends data. *Commercial–Sora* presents the throughput when a Sora node receives data. *Commercial–Commercial* presents the throughput when a commercial NIC communicates with another commercial NIC.

7.1 Experimental setup

The experimental setup consists of two high-end Dell XPS PCs (Intel Core 2 Quad 2.66GHz CPU, 4GB DDR2 400MHz SDRAM, and two PCIe-16x slots) and two laptops, all running Window XP. Each Dell PC equips a Sora radio control board (RCB) with an 802.11 RF board (Section 5) and runs Sora and the SoftWiFi implementation. Each CPU core has 32KB instruction and 32KB data L1 caches and a 2MB L2 cache. The Dell laptops use commercial WiFi NICs. We have used several different WiFi NICs in our experiments, including Netgear, Cisco and Intel devices. All give similar results. Thus, we present results just for the Netgear WAG511 device (based on the Atheros AR5212 chipset).

7.2 Throughput

Figure 11 shows the transmitting and receiving throughput of a Sora SoftWiFi node when it communicates with a commercial WiFi NIC. In the "Sora–Commercial" configuration, the Sora node acts as a sender and generates 1400-byte UDP frames and unicast transmits them to a laptop equipped with a commercial NIC. In the "Commercial–Sora" configuration, the Sora node acts as a receiver, and the laptop generates the same workload. The "Commercial–Commercial" configuration shows the throughput when both sender and receiver are commercial NICs. In all configurations, the hosts were at the same distance from each other and experienced very little packet loss. Figure 11 shows the throughput achieved for all configurations with the various modulation modes in 11a/b/g. We show only three selective rates in 11a/g for conciseness. The results are averaged over five runs (the variance was very small).

We make a number of observations from these results. First, the Sora SoftWiFi implementation operates seamlessly with commercial devices, showing that Sora SoftWiFi is protocol compatible. Second, Sora SoftWiFi can achieve similar performance as commercial devices. The throughputs for both configurations are essentially equivalent, demonstrating that SoftWiFi (1) has the processing capability to demodulate all incoming frames at full modulation rates, and (2) it can meet the 802.11 timing constraints for returning ACKs within the delay window required by the standard. We note that the maximal achievable application throughput for 802.11 is less than 80% of the PHY data rate, and the percentage decreases as the PHY data rate increases. This limit is due to the overhead of headers at different layers as well as the MAC overhead to coordinate channel access (*i.e.*, carrier sense, ACKs, and backoff), and is a well-known property of 802.11 performance.

7.3 CPU Utilization

What is the processing cost of onloading all digital signal processing into software on the host? Figure 12 shows the CPU utilization of a Sora SoftWiFi node to support modulation/demodulation at the corresponding rate. We normalize the utilization to the processing capability of one core. For receiving, higher modulation rates require higher CPU utilization due to the increased computational complexity of demodulating the higher rates. We can see that one core of a contemporary multicore CPU can comfortably support all 11b modulation modes. With the 11Mbps rate, Sora SoftWiFi requires roughly 70% of the computational power of one core for real-time SDR processing. However, 802.11a/g PHY processing is more complex than 11b and may require two cores for receive processing. In our software implementation, the Viterbi decoder in 11a/g is the most computationally-intensive component. It alone requires more than 1.4 Gcycles/s at modulation rates higher than 24Mbps (Table 1). Therefore, it is natural to partition the receive pipeline across two cores, with the Viterbi decoder on one core and the remainder on another. With the parallelism enabled by this streamline processing,

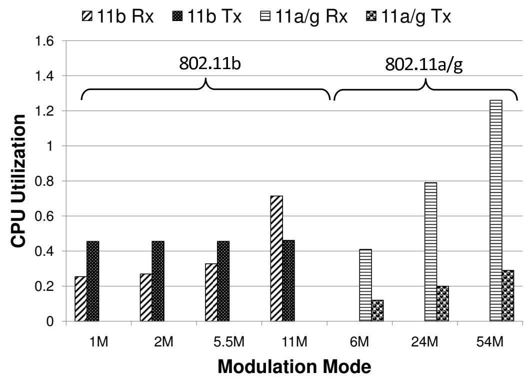


Figure 12: CPU Utilization of Sora.

we reduce the delay to process one 11a/g symbol from $4.8\mu s$ to $3.9\mu s$, meeting the requirement of the standard (*i.e.* $4\mu s$) for 54Mbps. Note that the CPU utilization is not completely linear with the modulation rates in 11b because the 5.5/11Mbps rates use a different modulation scheme than with 1/2Mbps.

The CPU utilization for transmission, however, is generally lower than the receiving case. Note that the utilization is constant for all 11b rates. Since the transmission part of 11b can be optimized effectively with LUTs, for different rates we just use different LUTs. In 11a/g, since all samples need to pass an IFFT, the computation requirements increase as the rate increases.

7.4 Detailed processing costs

The results in Figure 12 presented the overall CPU utilization for a Sora SoftWiFi receiving node. As discussed in Section 6, a complete receiver has a number of stages: frame detection, frame synchronization, and demodulators for both the PLCP header and its data depending on the modulation mode. How does CPU utilization partition across these stages? Figure 13 shows the computational cost for each component for receiving a 1400-byte UDP packet in each modulation mode; again, we show only three representative modulation rates for 11a/g. Frame detection (FD) has the lowest utilization (11% of a 2.66GHz core for 11b and only 3.2% for 11a/g) and is constant across all modulation modes in each standard. Note that frame detection needs to execute even if there is no communication since a frame may arrive at any time. When Sora detects a frame, it uses 29% of a core to synchronize to the start of a frame (SYNC) for 11b, and it uses 20% of a core to synchronize to an 11a/g frame. Then Sora can demodulate the PLCP header, which is always transmitted using the lowest modulation rate. It requires slightly less (27.5%) computation overhead than synchronization for 11b; but it needs much more computation (44%) for 11a. Demodulation of the data (DATA) at the higher rates is the most computationally expensive step in a receiver. It re-

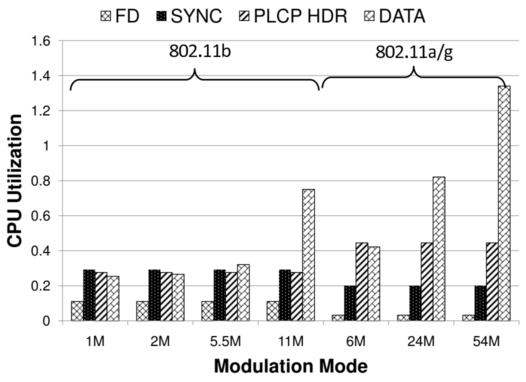


Figure 13: Detailed processing costs in WiFi PHY.

quires 75% of a core at 11Mbps for 11b, and the utilization reaches exceeds one core (134%) for processing at 54Mbps in 11a/g. This result indicates that we need to streamline the processing to at least two cores to support this modulation.

8 Extensions

The flexibility of Sora allows us to develop interesting extensions to current WiFi protocol.

8.1 Jumbo Frames

If the channel conditions are good, transmitting data with larger frames can reduce the overhead of MAC/-PHY headers, preambles and the per frame ACK. However, the maximal frame size of 802.11 is fixed at 2304 bytes. With simple modifications (changes in a few lines), SoftWiFi can transmit and receive jumbo frames with up to 32KB. Figure 14 shows the throughput of sending UDP packets between two Sora SoftWiFi nodes using the jumbo frame optimization across a range of frame sizes (with 11b using the 11Mbps modulation mode). When we increase the frame size from 1KB to 6KB, the end-to-end throughput increase 39% from 5.9Mbps to 8.2Mbps. When we further increase the frame size to 7KB, however, the throughput drops because the frame error rate also increases with the size. So, at some point, the increasing error will offset the gain of reducing the overhead. Note that our default commercial NIC rejects frames larger than 2304 bytes, even if those frames can be successfully demodulated.

In this experiment, we place the antennas close to each other, clearly a best-case scenario. Our goal, though, is not to argue that jumbo frames for 802.11 are necessarily a compelling optimization. Rather, we want to demonstrate that the full programmability offered by Sora makes it both possible and straightforward to explore such “what if” questions on a GPP SDR platform.

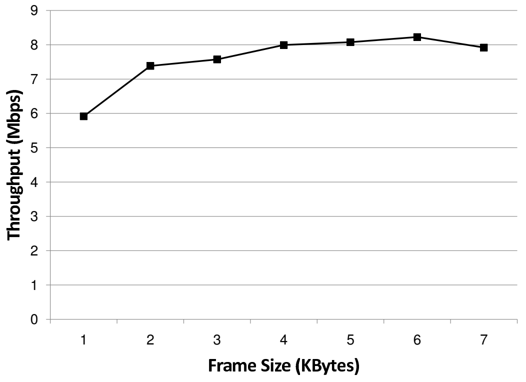


Figure 14: Throughput with Jumbo Frames between two Sora SoftWiFi nodes.

	10ms	50ms	100ms
$\epsilon/\sigma(\mu s)$	0.85/0.5	0.96/0.54	0.98/0.46
Outlier	0.5%	0.4%	0.4%

Table 4: Timing error of Sora in TDMA.

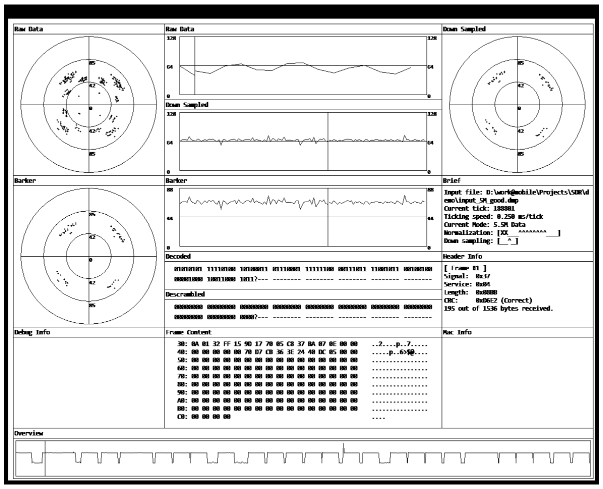
8.2 TDMA MAC

To evaluate the ability of Sora to precisely control the transmission time of a frame, we implemented a simple TDMA MAC that schedules a frame transmission at a predefined time interval. The MAC state machine (SM) runs in an ethread, and it continuously queries a timer to check if the pre-defined amount of time has elapsed. If so, the MAC SM will instruct the RCB to send out a frame. The modification is simple and straightforward with about 20 lines of additional code.

Since our RCB can indicate to SoftWiFi when the transmission completes, and we know the exact size of the frame, we can calculate the exact time when the frame transmits. Table 4 summarizes the results with various scheduling intervals under a heavy load, where we copy files on the local disk, download files from a nearby server, and playback a HD video simultaneously. In the Table, ϵ presents the average error and σ presents the standard deviation of the error. The average error is less than $1\mu s$, which is sufficient for most wireless protocols. We also list outliers, which we define as packet transmissions that occur later than $2\mu s$ from the pre-defined schedule. Previous work has also implemented TDMA MACs on a commodity WiFi NIC [20], but their software architecture results in a timing error of near $100\mu s$.

8.3 Soft Spectrum Analyzer.

It is also easy for Sora to expose all PHY layer information to applications. One application we have found useful is a software spectrum analyzer for WiFi. We have implemented such a simple spectrum analyzer that can graphically display the waveform and modulation points



```
Input file: D:\work@mobile\Projects\SDR\d
emo\input_5M_good.dmp
Current tick: 188801
Ticking speed: 0.250 ms/tick
Current Mode: 5.5M Data
Normalization: [XX___^^^^^^^^___]
Down sampling: [__^_]
```

```
01010101 11110100 10100011 01110001 11111100 00111011 11001011 00100100
00001000 10011000 1011?--- -------- -------- -------- -------- --------
```

```
[ Frame #1 ]
Signal:  0x37
Service: 0x04
Length:  0x0888
CRC:     0xD6E2 (Correct)
195 out of 1536 bytes received.
```

```
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 0000?--- -------- -------- -------- -------- --------
```

```
30: 0A 01 32 FF 15 9D 17 70 05 C8 37 BA 07 0E 00 00    ..2....p..7.....
40: 00 00 00 00 00 70 D7 CB 36 3E 24 40 DC 05 00 00    .....p..6>$@....
50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
A0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
B0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
C0: 00 00 00 00                                        ....
```

Figure 15: Software Spectrum Analyzer built on Sora.

in a constellation graph, as well as the demodulated results, as shown in Figure 15. Commercial spectrum analyzers may have similar functionality and wider sensing spectrum band, but they are also more expensive.

9 Related Work

In this section we discuss various efforts to implement software defined radio functionality and platforms.

Traditionally, device drivers have been the primary software mechanism for changing wireless functionality on general purpose computing systems. For example, the MadWiFi drivers for cards with Atheros chipsets [3], HostAP drivers for Prism chipsets [2], and the rtx200 drivers for RaLink chipsets [4] are popular driver suites for experimenting with 802.11. These drivers typically allow software to control a wide range of 802.11 management tasks and non-time-critical aspects of the MAC protocol, and allow software to access some device hardware state and exercise limited control over device operation (e.g., transmission rate or power). However, they do not allow changes to fundamental aspects of 802.11 like the MAC packet format or any aspects of PHY.

SoftMAC goes one step further to provide a platform for implementing customized MAC protocols using inexpensive commodity 802.11 cards [20]. Based on the MadWiFi drivers and associated open-source hardware abstraction layers, SoftMAC takes advantage of features of the Atheros chipsets to control and disable default low-level MAC behavior. SoftMAC enables greater flexibility in implementing non-standard MAC features, but does not provide a full platform for SDR. With the separation of functionality between driver software and hardware firmware on commodity devices, time critical tasks and PHY processing remain unchangeable on the device.

GNU Radio is a popular software toolkit for building software radios using general purpose computing platforms [1]. It is derived from an earlier system called SpectrumWare [22]. GNU Radio consists of a software library and a hardware platform. Developers implement software radios by composing modular pre-compiled components into processing graphs using python scripts. The default GNU Radio platform is the Universal Software Radio Peripheral (USRP), a configurable FPGA radio board that connects to the host. As with Sora, GNU Radio performs much of the SDR processing on the host itself. Current USRP supports USB2.0 and a new version USRP 2.0 upgrades to Gigabit Ethernet. Such interfaces, though, are not sufficient for high speed wireless protocols in wide bandwidth channels. Existing GNU Radio platforms can only sustain low-speed wireless communication due to both the hardware constraints as well as software processing [21]. As a consequence, users must sacrifice radio performance for its flexibility.

The WARP hardware platform provides a flexible and high-performance software defined radio platform [6]. Based on Xilinx FPGAs and PowerPC cores, WARP allows full control over the PHY and MAC layers and supports customized modulations up to 36 Mbps. A variety of projects have used WARP to experiment with new PHY and MAC features, demonstrating the impact a high-performance SDR platform can provide. KUAR is another SDR development platform [18]. Similar to WARP, KUAR mainly uses Xilinx FPGAs and PowerPC cores for signal processing. But it also contains an embedded PC as the control processor host (CPH), which has a 1.4GHz Pentium M processor. Therefore, it allows some communication systems to be implemented completely in software on CPH. They have demonstrated some GNU Radio applications on KUAR. Sora provides the same flexibility and performance as hardware-based platforms, like WARP, but it also provides a familiar and powerful programming environment with software portability at a lower cost.

The SODA architecture represents another point in the SDR design space [17]. SODA is an application domain-specific multiprocessor for SDR. It is fully programmable and targets a range of radio platforms — four such processors can meet the computational requirements of 802.11a and W-CDMA. Compared to WARP and Sora, as a single-chip implementation it is more appropriate for embedded scenarios. As with WARP, developers must program to a custom architecture to implement SDR functionality.

10 Conclusions

This paper presents Sora, a fully programmable software radio platform on commodity PC architectures. Sora combines the performance and fidelity of hardware SDR platforms with the programmability of GPP-based SDR platforms. Using the Sora platform, we also present the design and implementation of SoftWiFi, a software radio implementation of the 802.11a/b/g protocols. We are planning and implementing additional software radios, such as 3GPP LTE (Long Term Evolution), W-CDMA, and WiMax using the Sora platform. We have started the implementation of 3GPP LTE in cooperation with Beijing University of Posts and Telecommunications, China, and we confirm the programming effort is greatly reduced with Sora. For example, it has taken one student only two weeks to develop the transmission half of LTE PUSCH(Physical Uplink Shared Channel), which can be a multi-month task on a traditional FPGA platform.

The flexibility provided by Sora makes it a convenient platform for experimenting with novel wireless protocols, such as ANC [16] or PPR [15]. Further, being able to utilize multiple cores, Sora can scale to support even more complex PHY algorithms, such as MIMO or SIC (Successive Interference Cancellation) [23].

More broadly, we plan to make Sora available to the wireless networking research community. Currently, we are collaborating with Xi'an Jiao Tong University, China, to design a new MIMO RF module that supports eight channels. We are planning moderate production of the Sora RCB and RF modules for use by other researchers. The estimated cost for Sora hardware is about $2,000 per set (RCB + one RF front-end). We also plan to release the Sora software to the wireless network research community. Our hope is that Sora can substantially contribute to the adoption of SDR for wireless networking experimentation and innovation.

Acknowledgements

The authors would like to thank Xiongfei Cai, Ningyi Xu and Zenlin Xia in the Platform and Devices Center group at MSRA for their essential assistance in the hardware design of the RCB. We also thank Fan Yang and Chunyi Peng in the Wireless Networking (WN) Group at MSRA; in particular we have learned much from their early study on accelerating 802.11a using GPUs. We would also like to thank all members in the WN Group and Zheng Zhang for their support and feedback. The authors also want to thank Songwu Lu, Frans Kaashoek, and MSR colleagues (Victor Bahl, Ranveer Chandra, etc.) for their comments on earlier drafts of this paper.

References

[1] Gnu radio. *http://www.gnu.org/software/gnuradio/*.

[2] HostAP. *http://hostap.epitest.fi/*.

[3] Madwifi. *http://sourceforge.net/projects/madwifi*.

[4] Rt2x00. *http://rt2x00.serialmonkey.com*.

[5] Small form factor sdr development platform. *http://www.xilinx.com/products/devkits/SFF-SDR-DP.htm*.

[6] WARP: Wireless open access research platform. *http://warp.rice.edu/trac*.

[7] *ANSI/IEEE Std 802.11, Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification*. IEEE Press, 1999.

[8] *PCI Express Base 2.0 specification*. PCI-SIG, 2007.

[9] A. Agarwal and M. Levy. Thousand-core chips: the kill rule for multi-core. In *Proceedings of the 44th Annual Conference on Design Automations*, 2007.

[10] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: an operating system for many cores. In *OSDI*, 2008.

[11] M. Cummings and S. Haruyama. FPGA in the software radio. *IEEE Communications Magazine*, 1999.

[12] J. V. de Vegte. *Fundamental of Digital Signal Processing*. Cambridge University Press, 2005.

[13] J. Glossner, E. Hokenek, and M. Moudgill. The sandbridge sandblaster communications processor. In *3rd Workshop on Application Specific Processors*, 2004.

[14] A. Goldsmith. *Wireless Communication*. Cambridge University Press, 2005.

[15] K. Jamieson and H. Balakrishnan. Ppr: Partial packet recovery for wireless networks. In *Proceedings of ACM SIGCOMM 2007*, April 2007.

[16] S. Katti, S. Gollakota, and D. Katabi. Embracing wireless interference: analog network coding. In *Proceedings of ACM SIGCOMM 2007*, pages 397–408. ACM Press, 2007.

[17] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, and T. Mudge. Soda: A low-power architecture for software radio. In *ISCA '06: Proceedings of the 33rd International Symposium on Computer Architecture*, 2006.

[18] G. J. Minden, J. B. Evans, L. Searl, D. DePardo, V. R. Patty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A. M. Wyglinski, and A. Agah. Kuar: A flexible software-defined radio development platform. In *DySpan*, 2007.

[19] J. Neel, P. Robert, and J. Reed. A formal methodology for estimating the feasible processor solution space for a software radio. In *SDR '05: Proceedings of the SDR Technical Conference and Product Exposition*, 2005.

[20] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. Softmac - flexible wireless research platform. In *HotNets 05*, 2005.

[21] T. Schmid, O. Sekkat, and M. B. Srivastava. An experimental study of network performance impact of increased latency in software defined radios. In *WiNETCH07*, 2007.

[22] D. L. Tennenhouse and V. G. Bose. Spectrumware-a software-oriented approach to wireless signal processing. In *MobiCom 95*, 1995.

[23] S. Verdu. *Multiuser Detection*. Cambridge University Press, 1998.

Appendix A: SIMD example for FIR Filter

In this appendix, we show a small example of how to use SSE instructions to optimize the implementation of a FIR (Finite Impulse Response) filter in Sora. FIR filters are widely used in various PHY layers. An n-tap FIR filter is defined as

$$y[t] = \sum_{k=0}^{n-1} c_k \cdot x[t-k],$$

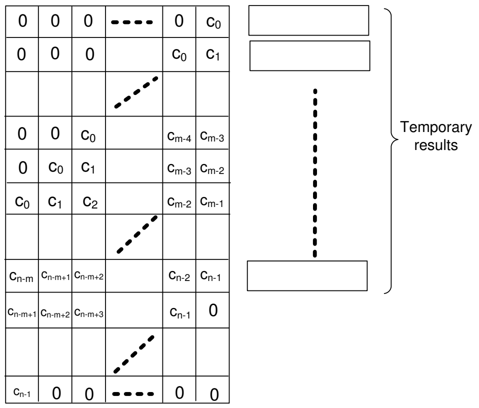


Figure 16: Memory layout of the FIR coefficients.

where $x[.]$ are the input samples, $y[.]$ are the output samples, and c_k are the filter coefficients. With SIMD instructions, we can process multiple samples at the same time. For example, Intel SSE supports a 128-bit packed-vector and each FIR sample takes 16 bits. Therefore, we can perform $m = 8$ calculations simultaneously. To facilitate SSE processing, the data layout in memory should be carefully designed. Figure 16 shows the memory layout of the FIR coefficients. Each row forms a packed-vector containing m components for SIMD operations. The coefficient vector of the FIR filter is replicated in each column in a zig-zag layout. Thus, the total number of rows is $(n + m - 1)$. There are also n temporary variables containing the accumulated sum up to each FIR tap for each sample.

Figure 17 shows the example code. It takes an array of input samples, a coefficient array, and outputs the filtered samples in an output sample buffer. The input contains two separate sample streams, with the even and odd indexed samples representing the *I* and *Q* samples, respectively. The coefficient array is arranged similarly to Figure 16, but with two sets of FIR coefficients for *I* and *Q* samples, respectively.

Each iteration, four *I* and four Q samples are loaded into an SSE register. It multiplies the data in each row and adds the result to the corresponding temporal accumulative sum variable (lines 59–68). A result is output when all taps are calculated for the input samples (lines 18–57). When the input sample stream is long, there are nm samples in the pipeline and m outputs are generated in each iteration. Note that the output samples may not be in the same order as the input — some algorithms do not always require the output to have exactly the same order as the input. A few shuffle instructions can be added to place the output samples in original order if needed.

```
1  int FirSSE ( PSAMPLE  pSrc,
2               PSAMPLE  pOutput,
3               int nSize, // number of complex samples
4               PSHORT    pCoff, // filter coeffs
5               int iTaps, // the highest index of tap (n-1)
6               PSAMPLE pTempBuf, // for temp value store
7             )
8  {
9    _asm {
10     mov esi, pSrc;
11     mov ecx, nSize;
12     mov ebx, pOutput;
13 outerloop:
14     mov edx, pCoff;
15     mov edi, pTempBuf;
16
17     ;// load samples 4-I and 4-Q
18     movdqa xmm0, [esi];
19
20     ; // result_0
21     movdqa xmm4, xmm0;
22     pmullw xmm4, [edx];
23     paddsw xmm4, [edi];
24     ; // result_1
25     movdqa xmm5, xmm0;
26     pmullw xmm5, [edx + 16];
27     paddsw xmm5, [edi + 16];
28     ; // result_2
29     movdqa xmm6, xmm0;
30     pmullw xmm6, [edx + 32];
31     paddsw xmm6, [edi + 32];
32     ; // result_3
33     movdqa xmm7, xmm0;
34     pmullw xmm7, [edx + 48];
35     paddsw xmm7, [edi + 48];
36
37     ; // xmm4, xmm5, xmm6, xmm7 contains output
38     ; // perform shuffle and horizontal additions
39     movdqa xmm1, xmm4;
40     punpckldq xmm1, xmm6;
41     punpckhdq xmm4, xmm6;
42     paddsw xmm4, xmm1;
43
44     movdqa xmm1, xmm5;
45     punpckldq xmm1, xmm7;
46     punpckhdq xmm5, xmm7;
47     paddsw xmm5, xmm1;
48
49     movdqa xmm1, xmm4;
50     punpckldq xmm1, xmm5;
51     punpckhdq xmm4, xmm5;
52     paddsw xmm4, xmm1;
53
54     ; // output
55     ; // additional instructions may be added to
56     ; // adjust the sample orders
57     movdqa [ebx], xmm4;
58
59     ; // update temp buffers
60     mov eax, iTaps;
61 innerloop:
62     movdqa xmm1, xmm0;
63     pmullw xmm1, [edx + 64];
64     paddsw xmm1, [edi + 64];
65     movdqa [edi], xmm1;
66
67     add edx, 16;
68     add edi, 16;
69     dec eax;
70     jnz innerloop;
71
72     ;// advance to next sample group
73     add esi, 16;
74     add ebx, 16;
75     sub ecx, 4;
76     jg outerloop;
77   }
78 }
```

Figure 17: Pseudo-code of SSE optimized FIR Filter.

Enabling MAC Protocol Implementations on Software-Defined Radios

George Nychis, Thibaud Hottelier, Zhuocheng Yang, Srinivasan Seshan, Peter Steenkiste
Carnegie Mellon University

Abstract

Over the past few years a range of new Media Access Control (MAC) protocols have been proposed for wireless networks. This research has been driven by the observation that a single one-size-fits-all MAC protocol cannot meet the needs of diverse wireless deployments and applications. Unfortunately, most MAC functionality has traditionally been implemented on the wireless card for performance reasons, thus, limiting the opportunities for MAC customization. Software-defined radios (SDRs) promise unprecedented flexibility, but their architecture has proven to be a challenge for MAC protocols.

In this paper, we identify a minimum set of core MAC functions that must be implemented close to the radio in a high-latency SDR architecture to enable high performance and efficient MAC implementations. These functions include: precise scheduling in time, carrier sense, backoff, dependent packets, packet recognition, fine-grained radio control, and access to physical layer information. While we focus on an architecture where the bus latency exceeds common MAC interaction times (tens to hundreds of microseconds), other SDR architectures with lower latencies can also benefit from implementing a subset of these functions closer to the radio. We also define an API applicable to all SDR architectures that allows the host to control these functions, providing the necessary flexibility to implement a diverse range of MAC protocols. We show the effectiveness of our *split-functionality* approach through an implementation on the GNU Radio and USRP platforms. Our evaluation based on microbenchmarks and end-to-end network measurements, shows that our design can simultaneously achieve high flexibility and high performance.

1 Introduction

Over the past few years, a range of new Media Access Control (MAC) protocols have been proposed for use in wireless networks. Much of this increased activity has been driven by the observation that a single one-size-fits-all MAC protocol cannot meet the needs of diverse wireless deployments and applications and, thus, MAC protocols need to be specialized (e.g. for use on long-distance links, mesh networks). Unfortunately, the development and deployment of new MAC designs has been slow due to the limited programmability of traditional wireless network interface hardware. The reason is that key MAC functions are implemented on the network interface card (NIC) for performance reasons, which often uses proprietary software and custom hardware, making the MAC hard, if even possible, to modify.

Software-defined radios (SDRs) have been proposed as an attractive alternative. SDRs provide simple hardware that translates signals between the RF and the digital domains. SDRs implement most of the network interface functionality (e.g., the physical layer and link layer) in software and, as a result, they make it feasible for developers to modify this functionality. SDR architectures [19, 6, 17, 20, 9] typically distribute processing of the digitized signals across several processing units – including FPGAs and CPUs located on the SDR device, and the CPU of the host. The platforms differ in the precise nature of the processing units that are provided, how those units are connected, and how computation is distributed across them.

Unfortunately, the high degree of flexibility offered by SDRs does not automatically lead to flexibility in the MAC implementation. The reason is that, in the SDR architecture we are addressing, the use of multiple heterogeneous processing units with interconnecting buses, introduces large delays and jitter into the processing path of packets. Processing, queuing, and bus transfer delays can easily add up to hundreds of microseconds [14]. Unfortunately, the delay limits how quickly the MAC can respond to incoming packets or changes in channel conditions, and the jitter prevents precise control over the timing of packet transmissions. These restrictions severely reduce the performance of many MAC protocols.

This paper presents a set of techniques that makes it possible to implement diverse, high performance MAC protocols that are easy to modify and customize from the host. The key idea is a novel way of splitting core MAC functionality between the host processing unit and pro-

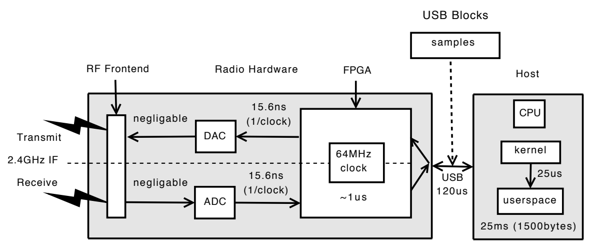


Figure 1: Generic SDR Architecture

cessing units on the hardware (e.g., FPGA). The paper makes the following contributions:

- We identify a set of core MAC functions that must be implemented close to the radio for performance and efficiency reasons.
- We define a *split-functionality* architecture that allows the functions to be implemented near the radio hardware, while maintaining control on the host CPU through an API.
- We present an implementation of our architecture using the GNU Radio [6] and USRP [17] SDR platform. We also use our implementation to characterize the performance-flexibility tradeoffs for key MAC features. For example, our results show three orders of magnitude greater precision for the scheduling of packets and carrier sense, along with a high level of accuracy in fast packet detection.
- Finally, we use our implementation for an end-to-end evaluation of the split-functionality architecture. We show how the system can support diverse high-performance MAC implementations by implementing 802.11-like and Bluetooth-like protocols for experimentation over the air.

The rest of the paper is organized as follows. We discuss current radio architecture and its impact on MAC protocol development in Section 2. In Sections 3 and 4, we explore the core MAC requirements and introduce our *split-functionality* architecture. Section 5 provides details for each component implementation with evaluation results. Finally, we present end-to-end evaluation results, related work, and a summary of our results in Sections 6 through 8.

2 MAC Implementation Choices

A number of different software-defined radio architectures have been developed. One common architecture is shown in Figure 1. The frontend is responsible for converting the signal between the RF domain and an intermediate frequency, and the A/D and D/A components convert the signal between the analog and the digital domain. Physical and higher layer processing of the digitized signal are executed on one or more processing units. Typically, there is at least an FPGA or DSP close to the frontend. The frontend, D/A, A/D, and FPGA are usually placed on a network card that is connected to the host CPU by a standard bus (e.g., USB).

The distribution of functionality across the processing units significantly impacts the radio's performance, flexibility, and ease of reprogramming. To achieve a high level of flexibility and reprogramming, the majority of processing (i.e., modulation) can be placed on the host CPU where the functionality is easy to modify. We refer to this architecture as *host-PHY*. This architecture is exemplified by GNU Radio [6] and the USRP [17], which place the majority of functionality in userspace, shown in Figure 1. For greater performance, processing can be implemented in the radio hardware on the FPGA or DSP. We refer to this architecture as *NIC-PHY*. The WARP platform [20] implements this architecture, placing the PHY and MAC layers on the radio hardware for performance reasons. It is fairly straightforward however, to parameterize PHY layers (e.g. to control the frequency band and coding an modulation options). Thus, it is possible control many aspects of the PHY layer from the host, no matter where it is implemented.

Unfortunately, MAC protocols are less structured and SDRs have fallen short in providing high-performance flexible MAC implementation. The MAC is either implemented near the radio hardware for performance, or near the host for flexibility. We propose a novel split of MAC functionality across the processing units in a *host-PHY* architecture such that we can achieve a high level of performance, while maintaining flexibility at both the MAC and PHY layers. This is especially significant in a *host-PHY* architecture, which has been considered incapable of supporting even core MAC protocol functions (e.g., carrier sense) due to the large processing delays inherent to the architecture [14, 18]. In addition, our design can enable many cross-layer optimizations, such as those proposed between the MAC and PHY layers [5, 8, 7]. Such optimizations have used the *host-PHY* architecture for easy PHY modifications, but given the lack of MAC support, they typically "fake" the MAC layer (e.g., by combining the SDR with a commodity 802.11 NIC to do the MAC processing [5]) or omit it all together [7, 8]. Although our work focuses on a *host-PHY* architecture, several of the components we will present can be applied to a *NIC-PHY* architecture.

In the next section, we explore delay and jitter measurements in the *host-PHY* architecture, which are the major limiting factor on performance of MAC implementations. The measurements are important in understanding the proper split of MAC functionality across the heterogeneous processing units of an SDR.

	Avg	SDev	Min	Max
User–>Kernel (μs)	24	10	22	213
Kernel–>User (μs)	27	89	13	7000
4096 Kernel<–>FPGA (μs)	291	62	204	360
512 Kernel<–>FPGA (μs)	148	35	90	193
GNU Radio<–>FPGA (μs)	612	789	289	9000

Table 1: Kernel level delay measurements.

2.1 Delay Measurements

Schmid et al [14] present delay measurement for SDRs and their impact on MAC functionality in a *host-PHY* architecture. However, they focus on user-level measurements, largely ignoring precise measurement of delays between the kernel and userspace, and kernel and the radio hardware. Such measurements are important, since they can provide insight into whether implementing MAC functions in the kernel is sufficient to overcome the performance problems associated with user level implementations. To obtain precise user and kernel-level measurements, we modified the Linux kernel's USB Request Block (*URB*) and USB Device Filesystem URB (*USBDEVFS_URB*) to include nanosecond precision timestamps taken at various times in the transmission and receive process. All user level timestamps are taken in user space right before or after a URB is submitted (write) or returned (read). At the kernel level, the measurement is taken at the last point in the kernel's USB driver before the DMA write request is generated, or after a DMA read request interrupts the driver. This is as close to the bus transfer timing as possible.

We measured the round trip time between GNU Radio (in user space) and the FPGA using a ping command on a control channel that we implement (Section 4.2). Using the measurements described above, we are also able to identify the sources of the delay by calculating the user to kernel space delay, kernel to user space delay, and round trip time between the kernel and FPGA based on ping. We ran the user process at the highest priority to minimize scheduling delay. We used the default 4096 byte USB transfer block size for all experiments, and then perform an additional kernel to FPGA RTT experiment using a 512 byte transfer block size, the minimum possible, in an attempt to minimize queuing delay.

The results presented in Table 1 are averaged over 1000 experiments. Focusing on the average times, we see the cost of a GNU Radio ping is dominated by the kernel-FPGA roundtrip time (291 out of 612 μs). The user-kernel and kernel-user times are relatively modest (24 and 27 μs). The remaining time (270 μs) is spent in the GNU Radio chain. The high latency of the kernel-FPGA roundtrip time is somewhat surprising, given that the effective measured rate of the USB with the USRP is 32MB/s. The difference between the latencies for 4KB and 512B shed some light on this. The difference in latency is only a factor of two, suggesting that the set up cost for transfers contributes significantly to the delay. The kernel-FPGA time also includes the time it takes for the data to pass through the USRP USB FX2 controller buffers, and to be copied into the FPGA for parsing. The time taken for the data to pass through the USRP USB FX2 controller buffers and copied into the FPGA for parsing also contributes to the kernel-FPGA RTT.

The standard deviations and the min/max values paint a different picture. The user-to-kernel and kernel-FPGA times fall in a fairly narrow range, so they only contribute a limited amount of jitter. The kernel-to-user times however have a very high standard deviation, which results in a high standard deviation for the GNU Radio ping delays. This is clearly the result of process scheduling.

2.2 MAC Design Space

As discussed briefly in Section 2, the processing units in the above SDR architecture have very different properties. Focusing on Figure 1, the host CPU is easy to program and is readily accessible to users and developers. However, the path between the host CPU and the radio front end has both high delay and jitter, as shown by the measurements presented in Section 2.1. The round trip times between the device driver on the host and the FPGA is about 300 μs for 4KB of data, with relatively modest jitter. The roundtrip from GNU Radio is about double, but with significantly more jitter. As a result, a host-based MAC protocol (be it in user space or in the kernel) will not be able to precisely control packet timing, or implement small, precise inter-frame spacings, which will hurt the performance of many MAC protocols. We conclude that, *time critical radio or MAC functions should not be placed on the host CPU*.

Processing close to the radio performed by a FPGA or CPU on the NIC has the opposite properties. It has a low latency path to the frontend (see USRP latencies in Figure 1), making it attractive for delay sensitive functions. Unfortunately, code running on the radio hardware is much harder to change because it is often hardware-specific and requires a more complex development environment. Moreover, history shows that vendors do not provide open access to their NICs, even if they are programmable. Access to the processors on the NIC is restricted to its manufacturer and possibly large customers who can, under license, customize the NIC code. This is of course not a problem for research groups using research platforms, which is why many researchers are moving to software radios, but it is an important consideration for widespread deployment. We conclude that *in order to be widely applicable, the control of flexible MAC implementations should reside on the host*.

Interesting enough, the SDR NIC architecture in Figure 1 is not unlike the architecture of traditional NICs (e.g., 802.11 cards). Today's commodity NICs use analog hardware to perform physical layer processing, but they typically also have a CPU, FPGA, or custom processor. These commodity devices exhibit the same tradeoffs we identified above for software radios: the delay between the processing on the host and the (analog) frontend is substantially higher and less predictable than between the NIC processor and the front end.

Experience with commercial 802.11 cards supports the conclusions we highlighted above. First, time sensitive MAC functions such as sending ACKs are always performed on the NIC, and only functions that are not delay sensitive such as access point association are handled by the host processor. Moreover, although most of the MAC functionality on the NIC is implemented in software, it can only be modified by a small number of vendors (i.e. in practice the NIC is a black box). Researchers have had some success in using commodity cards for MAC research by moving specific MAC functions to the host [13, 16, 10, 15], but the results are often unsatisfactory. The host can only take control over certain functions (e.g. interframe spacings must be longer than 60 microseconds), precision is limited (e.g. cannot eliminate all effects of jitter), and the host implementation is inefficient (as a result of polling) and is susceptible to host loads.

The different properties of the host and NIC processing units means that the placement of MAC functionality will fundamentally affect four key MAC performance metrics, including network performance, flexibility in MAC implementation and runtime control, and ease of development. Unfortunately, as discussed above, these performance goals are in conflict with each other and achieving the highest level for each is not possible. In this paper, we present a split-functionality architecture that implements key MAC functions on the radio hardware, but provides full control to the host. This allows us to simultaneously score very high on all four metrics, and it also allows developers and users to make tradeoffs across the metrics. While developers will always have to make tradeoffs, the negatives associated with specific design choices are significantly reduced in our design. Note that this does not imply that our design can support any arbitrary or even all existing MAC designs. However, we believe that it is capable of supporting most of the critical features of modern MAC designs.

The focus of the paper is on SDR platforms because they provide maximal flexibility in key research areas such as cross-layer MAC and PHY optimization (e.g., [5, 7, 8]). Our evaluation is based on a platform that uses the host-PHY architecture, but is not critical. Even in NIC-PHY architectures that have good support for the MAC on the NIC (e.g., in the form of a general-purpose CPU), it is important to maintain control over the MAC and PHY on the host to ensure easy customization. As a result, the techniques we propose can be useful across the entire spectrum of NIC designs.

3 Core MAC Functions

An ideal wireless protocol platform should support the implementation of well-known MAC protocols as well as novel MAC research designs. A study of current wireless protocols, including WiFi (both Distributed and Point Coordination Function), Zigbee, Bluetooth, and various research protocols shows that they are based on a common, core set of techniques such as contention-based access (CSMA), TDMA, CDMA, and polling. In this section, we identify key core functions that a platform must implement efficiently in order to support a wide range of MAC protocols.

Precise Scheduling in Time: TDMA-based protocols require precise scheduling to ensure that transmissions occur during time slots. Imprecise timing can be tolerated by using long guard periods; however, this degrades performance. Surprisingly, modern contention-based protocols also require precise scheduling to implement inter-frame spacing (i.e. DIFS, SIFS, PIFS), contention windows, back-off periods, etc.

Carrier Sense: Contention-based protocols often use carrier sense to detect other transmissions. Carrier sense may use simple power detection (e.g., using signal strength) or may use actual bit decoding. Network interfaces need to transmit shortly after the channel is detected to be idle. Additional delay increases both the frequency of collision and also the minimum packet size required by the network.

Backoff: When a transmission fails in a contention-based protocol, a backoff mechanism is used to reschedule the transmission under the assumption that the loss was caused by a collision. Backoff is related to precise scheduling, but focuses more closely on fast-rescheduling of a transmission without the full packet transmission process (e.g., modulation).

Fast Packet Recognition: Many MAC performance optimizations could use the ability to quickly detect an incoming packet and identify that it is relevant to the local node in a timely and accurate manner. For example, detecting and identifying an incoming packet before the demodulation procedure can reduce resource use on the processing units and on the bus.

Dependent Packets: Dependent packets are explicit responses to received packets. A typical example is control packets that are associated with data packets, for example for error control (e.g., ACKs) or for improved channel access (e.g., RTS/CTS). Network interfaces need to generate these packets quickly and transmit them with precise time scheduling relative to the previous packet.

Fine-grained Radio Control: Frequency-hopping spread spectrum protocols such as Bluetooth and the recently proposed MAXchop algorithm [11] require fine-grained radio control to rapidly change channels according to a pseudo-random sequence. Similarly, recent designs [1] for minimizing interference require the ability to control transmission power on a per-packet basis.

Access to physical layer information: Many MAC protocol optimizations could benefit from access to radio-level packet information. Examples include using a received signal strength indicator (RSSI) to improve access point handoff decisions and using information on the confidence of each decoded bit to implement partial packet recovery [7].

3.1 Implications

While it is difficult to argue that this (or any) list of core functions is the correct one and is complete, we believe that it is sufficient to implement a broad range of interesting MAC protocols. To provide some degree of confidence in this statement, we describe our implementation of an 802.11-like CSMA protocol and a Bluetooth-like TDMA protocol using our framework in Section 6. As such, this is a reasonable first "toolbox" that MAC protocol developers can extend over time.

4 Split Functionality Architecture

As discussed in Section 2, implementing flexible high-performance MAC protocols is challenging because the high delays and jitter between the host CPU and frontend affects the performance of the core MAC functions described in the previous section. For example, most protocols need either precise scheduling in time or dependent packets. However, the delays inherent in a host MAC implementation in the given SDR architecture would make these functions inefficient or ineffective. In this section, we first review the requirements associated with the core MAC functions identified above, and we then present an architecture that allows us to support high performance MACs while maintaining host control.

4.1 Core Requirements

Implementing the core MAC functions from Section 3 raises three challenges.

Bus delay: The delay introduced by transmission of data over the bus can be constant and predictable, depending on the technology. A constant delay is relatively easy to accommodate in supporting *precision scheduling*, as discussed in Section 5.1. However, the bus delay does impact the performance of *carrier sense*, *dependent packets*, and *fast packet recognition*. The effect of bus latency on performance for SDR NICs is discussed in previous work [14].

Queuing delay: The delay introduced by queues may be smaller than the bus transmission delay but has significant jitter, which makes precision scheduling difficult, if not impossible. The jitter can modify the inter-packet spacing through compression or dispersion as the data is processed in the host and at the ends of the bus. In Section 5.1.2, we present measurements that show that this compression can be so significant in the given architecture that spacing transmissions by under 1ms cannot be achieved reliably using host-CPU based scheduling.

Stream-based architecture of SDRs: The frontend operates on streams of samples, which can make *fine-grained radio control* and *access to physical layer information* from the host ineffective. The reason is that it adds complexity to the interaction between a MAC layer executing on a host CPU (or NIC CPU) and the radio frontend since it is difficult to associate control information or radio information with particular groups of samples (e.g., those belonging to a packet). This problem consists of two components: (1) how to propagate information within the software environment that performs physical and MAC layer processing, and (2) how to propagate the information between the host and the frontend, across the bus and SDR hardware. This first issue is being addressed in the GNU Radio design with the introduction of m-blocks [2], which is briefly discussed in Section 7, but we must address the second issue.

4.2 Overcoming the Limitations

We now present an architecture that overcomes the above limitations. The goal is to allow as much of the protocol to execute on the host as possible to achieve the flexibility and ease of development goals, both of which are important to a wireless platform for protocol development, as identified in Section 2. However, we must ensure that the high latency and jitter between the host and radio frontend does not result in poor performance and limited control, the other two criteria in Section 2. This is done by introducing two architectural features, **per-block meta-data** and a **control channel**, shown in Figure 2. The novelty is not in the two new architectural features, but in how we use them to implement the core MAC functions (Section 3) in such a way that we maintain flexibility, while increasing performance (Section 5). We first discuss both features in more detail.

Per block meta-data: Enabling the association of information with a packet is crucial to the support of nearly

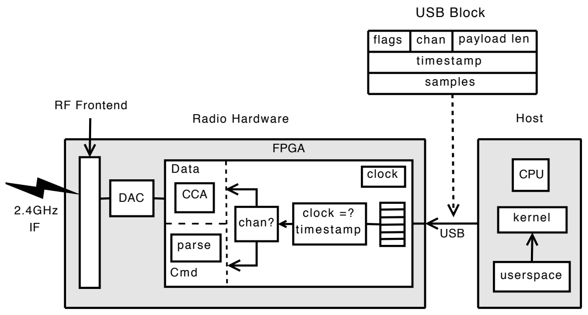


Figure 2: Split SDR architecture.

all of the core requirements in Section 3. Each packet is modulated into blocks of samples, for which we introduce per block meta-data. The meta-data stored in the header includes a timestamp (inbound and outbound), a channel flag (data/control), a payload length, and single bit flags to mark events such as overrun, underrun, or to request specific functions that we implement on the radio hardware. We limit the scope of the meta-data to the minimum needed to support the core requirements, thus minimizing the overhead on the bus.

Control Channel: The *control channel* allows us to implement a rich API between the host and radio hardware and allows for less frequent information to be passed. It consists of control blocks that are interleaved with the data blocks over the same bus. Control blocks carry the same meta-data header as data blocks but have the channel field in the header set to *CONTROL*. The control block payload contains one or more command subblocks. Each subblock specifies the command type, the length of the subblock, and information relevant to the specific command (e.g., a register number). Examples of commands include: reading or writing configuration registers on the SDR device, changing the carrier frequency, and setting the signal sampling rate.

With these two features, we can effectively partition the core MAC functions into a part that runs on the radio hardware close to the radio frontend, and a control part that runs on the host. Of course, meta-data and control channels are used in many contexts. The contribution lies in how we use them to partition the core MAC functions, which is the focus of the next section.

5 Core Component Design and Evaluation

We now examine how the split-functionality approach can be used to implement the core functions described in Section 3. We also evaluate the performance of the implementation of each core function. We focus our discussion on the GNU Radio and USRP platform.

5.1 Precise Scheduling in Time

Precision scheduling needs to be implemented close to the radio to achieve the fine-grained timing required for TDMA, spread spectrum, and contention based protocols. This is especially important when a large amount of jitter exists in the system from multiple stages of queuing and process scheduling, explored in Section 2.1.

For nodes to synchronize to the time of a global reference point, such as a beacon transmission for synchronization to the start of a round in a TDMA protocol, the nodes need to accurately estimate the reference point. Jitter at the transmitter can cause the actual transmission of the beacon to vary from its target time by δ_t, the maximum transmission jitter. Moreover, the estimated time of the beacon transmission as a global reference point will vary by δ_r, the maximum reception jitter. The maximum error is therefore $\delta_t + \delta_r$, which defines the minimum guard time needed by a TDMA protocol. By minimizing δ_t and δ_r, we increase channel capacity.

5.1.1 Precision Scheduling Design

Our delay measurements in Section 2.1 suggest that much of the delay jitter is created near the host. Therefore, the triggering mechanism for packet transmissions should reside beyond the introduction of the jitter. Likewise, to obtain an accurate local time at which a reception occurs, the time should be recorded prior to the introduction of the jitter on the RX path. To enable precision scheduling, we use a free running clock on the radio hardware to coordinate transmission/reception times as follows.

Transmit: To reduce the transmission jitter (δ_t), we insert a timestamp on all sample blocks sent from the host to the radio hardware. When the radio hardware receives the sample block, it waits until the local clock is equal to the *timestamp* value before transmitting the samples. This allows for timing compression or dispersion of data in the system with no effect on the precision scheduling of the transmission. The host must ensure the transmission reaches the radio hardware before the *timestamp* is equal to the hardware clock, else the transmission is discarded. The host is notified on failure, which can be treated as notification to schedule transmissions earlier. To support traditional best-effort streaming, we use a special timestamp value, called *NOW*, to transmit the block immediately.

In practice, the samples for a packet will be fragmented across multiple blocks. To make sure that a single packet's transmission is continuous and that if the packet is dropped all fragments are dropped, we implement *start of packet* and *end of packet* flags in the block headers. The first block carrying the packet will have the *start of packet* flag set and the timestamp for transmis-

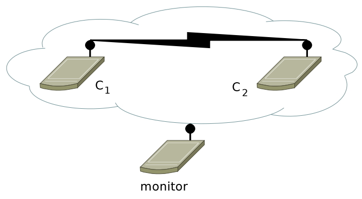


Figure 3: Evaluation setup using 3 USRPs.

sion. All remaining blocks carry a timestamp value of *NOW* to ensure continuous transmission. The hardware detects the last fragment using the *end of packet* flag, and can also report underruns to the host by detecting a gap between fragments.

A common solution to achieve precise transmission spacing from the host is to leave the transmitter enabled at all times and space transmissions with 0 valued samples. This solution is inefficient since it wastes both host CPU cycles and bus bandwidth, and it does not eliminate jitter on the receive side.

Receive: To reduce the receiver jitter (δ_r), the radio hardware *timestamps* all incoming sample blocks with the radio clock time at which the first sample in the block was generated by the ADC. Given that the sampling rate is set by the host, the host knows the exact spacing between samples. It can therefore calculate the exact time at which any sample was received, eliminating δ_r and allowing for full synchronization between transmitter and receiver.

5.1.2 Precision Scheduling Evaluation

To evaluate precision scheduling, we compare the timestamp-based release of packets using the split-functionality approach with a timer-based implementation in GNU Radio and in the kernel. We enable the real-time scheduling mechanism, which sets the GNU Radio processes to the highest priority. Our experiment transmits a frame used as a logical time reference, and then attempts to transmit another frame at a controlled spacing over the air. With no error, the actual spacing over the air is equal to the targeted spacing. We measure the actual spacings achieved using a monitoring node (Figure 3). A USRP on the monitoring node measures the magnitude of received complex samples at 8 megasamples per second, resulting in a precision of 125 nanoseconds. With no transmission jitter (δ_t), the spacing between beacons will exactly match their transmission rate, while any variability in scheduling will affect the spacings. The nodes are connected via coaxial cable to avoid the impact of external signals.

We compare the measured spacing of 50 transmissions with targeting spacings from 100ms to 1μs. Figure 4 shows the host and kernel based implementations to have approximately 1ms and 35μs of error, respectively. The timestamp-based mechanism achieves exact spacing to our monitoring node's precision. Therefore, moving timestamps to the kernel improves accuracy, but the error is still at least an order of magnitude greater than in the split-functionality design. Section 6.1 quantifies the benefits further through the implementation of a Bluetooth-like TDMA protocol. In the evaluation, we also measure δ_r with the split-functionality approach to be within 312ns. The average results show one-sided error, illustrating that compression of data across the bus dominates over dispersion. This is likely due to the multiple stages of buffers, including the buffers on the radio hardware to read the data from the FX2 controller. While dispersion is recorded, it occurs infrequently.

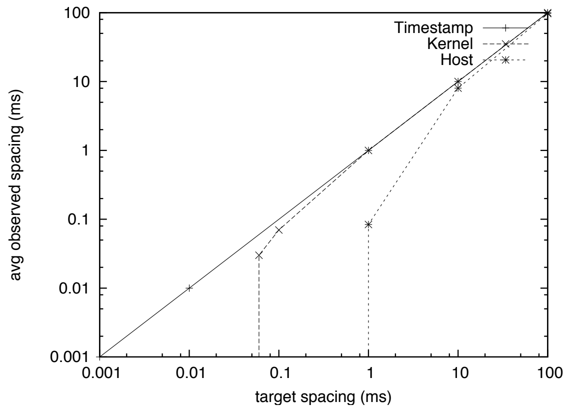


Figure 4: Split-functionality vs. host scheduling.

5.2 Carrier Sense

The performance of carrier sense is crucial to CSMA protocols: the longer it takes to transmit a packet after the channel goes idle, the greater the chance of collision. This turnaround time is referred to as the carrier sense 'blind spot" by Schmid et al. [14]. This blind spot has 4 components: signal propagation delay, the delay between the radio hardware and host for incoming samples, the processing delay involved in carrier detection at the host, and the complete transmission delay once the medium is detected idle at the host; this includes modulation of a packet and transferring the samples to the radio hardware for transmission.

5.2.1 Carrier Sense Design

To significantly reduce the size of the carrier sense blind spot, we must avoid the associated delays by placing the decision at the radio hardware. However, the decision process should be controlled by software running on the host CPU to maintain flexibility. The first assumption we can make is that if carrier sense is to be performed, the host has data to transmit and can modulate it and pass

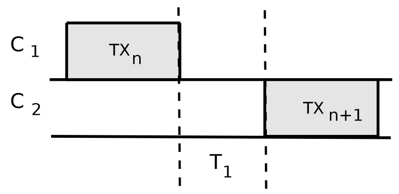


Figure 5: Carrier sense blind spot measurement.

it to the radio hardware to pend on carrier sense. The per block meta-data for the transmission has a single bit flag set to indicate the block should be held until there is no carrier using a locally computed RSSI value. The host can control the carrier sense threshold via the control channel. We use an RSSI value recorded in the radio hardware to implement a simple RSSI threshold carrier sense mechanism.

5.2.2 Carrier Sense Evaluation

We now present an evaluation of the *carrier sense* component in comparison to performing carrier sense at the host. In the host implementation, the received signal strength is estimated from the incoming sample stream and uses thresholds to control outgoing transmissions. We use the evaluation setup in Figure 3, described in Section 5.1.2, to achieve a 125 nanosecond resolution in measuring the archived carrier sense blind spot. The two contending nodes exchange the channel using carrier sense 100 times and we measure the spacing between each transmission, as illustrated in Figure 5. The first contending node, C_1, finishes transmission TX_n, and C_2 takes T_1 time to detect the channel as idle and begin transmission TX_{n+1}. T_1 represents the carrier sense turnaround time, or blind spot.

We plot two example channel exchanges using both implementations in Figure 6. Time is relative in the figure and we align the contending node's end of transmission at time 100. We highlight the gap in both implementations, and present the average gap observed across 100 exchanges: 1.5μs and 1.98ms for the split-functionality and host implementations, respectively. The host based latency could be reduced closer to 1ms, or on the order of tens of microseconds, by splitting the functionality to the USRP device driver, or the kernel, respectively. In our evaluation, the times were recorded at a higher-level block in GNU Radio where a MAC protocol would reside. These measurements illustrate our design's ability to reduce the carrier sense blind spot by *three orders of magnitude*, while maintaining host control on a per-packet basis. This can significantly increase the capacity in the channel by reducing the time it takes to detect it is idle. The host can even control the threshold on a per-packet basis by placing a control packet with a new threshold on the bus before the data packet.

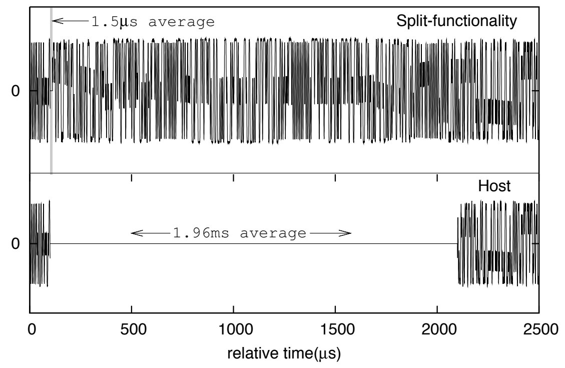


Figure 6: Measured carrier sense blind spots.

5.3 Backoff

In contention based protocols, backoff is used to reduce collisions and increase fairness. Although the technique varies by protocol, a common implementation is to reduce collisions by forcing a transmission delay and to increase fairness by making this delay random. The various delay components in SDRs prevent fine-grained backoff at the host. As shown in Section 5.1, a host backoff of less than 1ms is unachievable and values between 1ms and 100ms would be unpredictable. Therefore, backoff at the host would require a large minimum backoff time, which decreases channel capacity.

Despite our timestamping mechanism achieving microsecond level accuracy (Section 5.1.2), such a mechanism alone is insufficient. If a new backoff time is to be computed once a failure is reported to the MAC on the host, the retransmission would incur at least a radio-to-host RTT after the previous transmission, meaning the minimum backoff in a host implementation is an RTT. The average RTT measured in Section 2.1 was 612μs with a standard deviation was 789μs and a maximum observed value of 9ms. This is insufficient by current protocol standards. Placing the backoff algorithm on the radio hardware would require developers to make low level changes. We therefore explore a split-functionality approach for backoff.

5.3.1 Backoff Design

To enable flexible fine-grained backoff we build upon the precision scheduling mechanism (Section 5.1) to introduce a technique that leaves the backoff algorithm and computations at the host, and the actual transmission delay on the radio hardware. The key observation that enables our technique is that all backoff times, from the initial transmission n_0 to $n_{\text{MAX\_RETRIES}}$, can be pre-calculated by the host. The host calculates the backoff time for transmission n_0, and then assuming failure calculates all remaining backoffs from 1 to *MAX_RETRIES*, including each in the per packet meta-data.

A flag is set in the per block meta-data for the radio hardware to interpret the timestamp value as the maximum number of retries (M), and the first M 32-bit words pre-pended in the data payload to be interpreted as backoff times for each retransmission. Each value is interpreted as a time-to-wait, where the transmission is scheduled at *current_clock+backoff*. Moreover, we implement a control channel command that allows the host to configure the interpretation of a backoff value as an absolute time-to-wait, or a channel idle time-to-wait (most common).

This technique does not affect scheduling of future transmissions, as for example in 802.11 the contention window is reset to the minimum on a successful transmission. This means that the host can fully schedule a transmission and before a success/failure notification is given by the hardware, it can prepare the next transmission and buffer it on the radio hardware.

5.3.2 Backoff Evaluation

Given that the backoff technique uses the precision scheduling mechanism, its accuracy is the same as the precision scheduling mechanism and on the order of microseconds. We also use the backoff technique in our split-functionality 802.11-like protocol evaluation found in Section 6.

5.4 Fast Packet Recognition

Traditional software-defined radios, in the receive state, stream captured samples at some decimated rate between the radio hardware and the host. For many MAC protocols, such as CSMA-style designs, the radio cannot determine when packets for the attached node will arrive. As a result, the radio must remain in the receiving state. The downside to this is that the demodulation process uses significant memory and processor resources despite the fact that incoming packets destined for the radio are infrequent. As such radios become more ubiquitous and common for implementation, resource usage will become increasingly important, especially for energy-constrained devices such as the battery-powered Kansas University Agile Radio [9].

One simple solution would be to send samples when the RSSI is above some threshold. However, this does not filter out transmissions destined to other hosts and external signals. A better solution would be to have the radio hardware look for the packet preamble and the destination address, then transfer a maximum packet size worth of samples to the host after any match. At first glance, it may seem that fast packet recognition

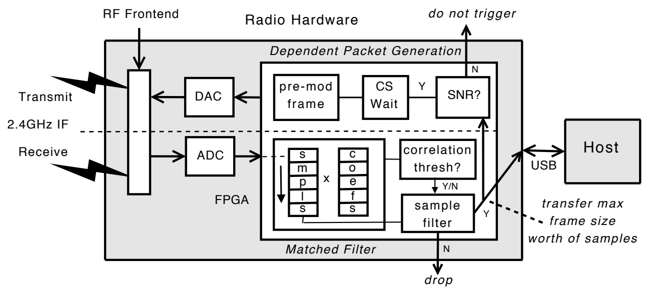


Figure 7: Matched filter & dependent packet design.

is not a "necessary" function for implementing MAC protocols, especially since the CPU and bus bandwidth resource consumption can become insignificant rather quickly (i.e., due to Moore's Law). However, trends in bus delay do not have this same property. As we will discuss further in Section 5.5, the ability to identify packets and process them partially on the SDR hardware is critical to supporting low-latency MAC interactions (e.g., packet/ACK exchanges or RTS/CTS) in a high-latency architecture.

5.4.1 Fast Packet Recognition Design

Our goal is to accurately detect packets at the radio hardware without demodulating the signal (to keep flexibility), for which we perform signal detection. The most relevant work in signal detection comes from the area of radar and sonar system design. From this area, we borrow a well-known technique, called a *matched filter*, to detect incoming packets at the radio hardware without the demodulation stage. For the purpose of design discussion, we refer to the bottom half of Figure 7.

Matched filter: A matched filter is the optimal linear filter that maximizes the output signal to noise ratio for use in correlating a known signal to the unknown received signal. For use in packet detection, the known signal would be the time-reversed complex conjugate of the modulated framing bits. This known signal is stored as the coefficients of the matched filter (Figure 7). The received sample stream is convolved with the coefficients to perform cross-correlation, where the output can be treated as a correlation score between the unknown and known signals. The correlation score is then compared with a threshold to trigger the transfer of samples to the host. The matched filter is flexible to different modulation schemes (e.g., GMSK, PSK, QAM), but requires a Fast Fourier transform for OFDM, given that the symbols are in the frequency domain. This would require an FFT implementation on the radio hardware.

To also detect that the frame is destined to the particular host, two different methods that have mathematically different properties can be used. *Single Stage*: Use a frame format where the destination address is the first field after the framing bits, and use this complete modulated sequence as the matched filter coefficients. *Dual Stages*: detect the framing bits first, then change the coefficients to the modulated destination address. Our implementation uses the single stage approach for simplification. However, a dual stage is more appropriate for monitoring multiple addresses such as a local address and a broadcast address.

5.4.2 Fast Packet Recognition Evaluation

We evaluate the effectiveness of the matched filter at detecting incoming sequences using simulations where we can control the noise level. Results are presented from over the air experiments with the presence of interference, multipath, and fading in Section 5.5.

To evaluate the effectiveness of the matched filter with varying signal quality, we first run experiments with controlled signal-to-noise ratios (SNR) using the GNU Radio software. We introduce additive white Gaussian noise (AWGN) to control the SNR in terms of dB:

$$SNR(dB) = 10 * log_{10} * \frac{Power_{signal}}{Power_{noise}} \tag{1}$$

To introduce noise, we compute the noise power based on the specified *snr* and power in the signal:

$$\begin{aligned} SNR &= 10^{(snr/10)} \\ Power_{signal} &= \left|Signal_{ampl}\right|^2 \\ Power_{noise} &= \frac{Power_{signal}}{SNR} \end{aligned}$$

For evaluation, 1000 frames of 1500 bytes are encoded using the Gaussian minimum-shift keying (GMSK) modulation scheme. These frames are used as the ground truth and mixed with the noise. We require that the matched filter detect the framing bits *and* that the transmission is destined for the attached host using the single-stage scheme (Section 5.4.1). The success rate is defined as the number of detected frames over the total number of frames in the dataset (1000). For comparison, we also include the success rate of the full GMSK decoder. At a high noise level, even the full decoder will fail at detecting the frames. The success rate, as a function of the SNR, is shown in Figure 8. The results show that the matched filter can detect the frames at a much higher success rate than the decoder can, even at low SNR levels where the noise power is greater than the signal power.

Given these results, and further real-world results presented in Section 5.5, we conclude that using the matched filter for detecting relevant packets is accurate enough that the host will never miss an actual frame due to the filter. In fact, the filter triggering samples to the host can been seen from a different perspective as providing further confidence to the host that there is actually

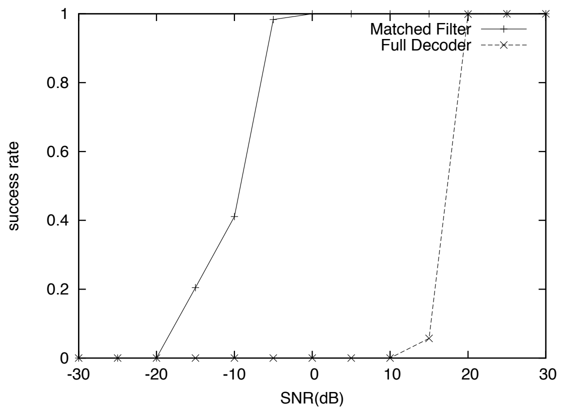


Figure 8: Success rate of the matched filter.

a frame within the sample stream. The host could then perform additional processing in an attempt to decode the frame successfully.

5.5 Dependent Packets

Dependent packets are packets generated in response to another packet (e.g., an ACK or RTS packet). MAC protocols often leave the channel idle during the dependent packet exchanges such as RTS-CTS and data-ACK exchanges. As a result, reducing the turnaround time of such exchanges can significantly increase overall capacity. In a host-based MAC, three sources contribute to the delay associated with dependent packet generation: bus transmission delay, queuing delay, and processing time. In this section, we explore the use of a matched filter along with additional techniques for triggering dependent packet responses on the radio hardware. The technique minimizes processing time by placing the packet detection as close to the radio as possible and avoids bus transmission and queuing delays by triggering a pre-modulated packet stored on the radio hardware.

5.5.1 Decoding Delay at the Host

We begin by quantifying the processing delay associated with host-based dependent packet generation. Note that we have already quantified bus delays in Section 2.1. We measure decode time for various frame sizes at the maximum supported decoding rate of the USRP: 2Mbps. The larger frame sizes would be representative of processing time for data/ACK exchanges, and the smaller frame sizes for RTS/CTS exchanges.

We use two 3.0GHz Pentium 4 machines running GNU Radio with their USRPs transmitting/receiving using the GMSK modulation scheme. Using host based timers, we record the minimum, average, and maximum time to decode 6 different frame sizes seen in Figure 9. The average decoding time is close to the mini-

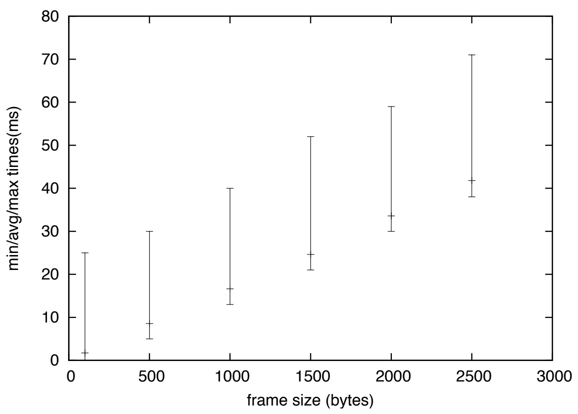


Figure 9: Decode times for various frame sizes.

mum recorded times for each frame size, however, rather large delays can be experienced at each frame size, likely due to the jitter introduced by queuing delays and process scheduling. Therefore, if one were to implement the matched filter at the radio hardware to detect incoming dependent packets and generate responses, anywhere from several milliseconds to 70 milliseconds can be saved solely in host processing.

5.5.2 Generating Fast-Dependent Packets

As an optimization to circumvent the decoding delays described, we develop a mechanism for fast-dependent packet generation in the radio hardware. This is not necessarily limited to *host-PHY* architectures. Although bus delay is reduced in *NIC-PHY* architectures, they typically use slower processors that increases decoding delays. Fast-dependent packet generation has three stages: (1) fast-packet detection of the initiating packet (e.g., RTS), (2) conditionals specific to the protocol that trigger the dependent packet, and (3) transmission of a pre-modulated dependent packet. We discuss stages 2 and 3 in this section. **Stage 1** was detailed in Section 5.4, although it is important to point out that by running multiple matched filters in parallel, it is possible to detect and respond to different initiating packets.

Stage 2: To introduce protocol dependent behavior after stage 1 detects the initiating packet and its end of transmission (the incoming signal drops to the noise floor), protocol developers can introduce a set of conditionals that control when a dependent packet is generated. In our current implementation this must be written in a hardware description language (Verilog), which has primitives similar to those in C/C++ (e.g., if, else, case, etc.). A simple example is the conditional for generating a CTS in Verilog. It checks that the receiver and channel are idle: *if(!receiving && RSSI < carrier_sense_thresh)*.

A more interesting example is the fast-ACK generator developed for our 802.11-like protocol (Section 6.3). We write 3 simple conditional statements around an SNR value. If any of the conditionals *pass* during the transmission, the radio hardware concludes that the host would not have been able to decode the packet, and a fast-ACK should not be triggered. The following are the 3 conditionals, with reasons as to why the fast-ACK should not be generated based on the conditional passing. (1) *if(SNR < lowest_thresh)*: interference throughout the transmission. (2) *if(last_SNR_val - SNR < drop_thresh)*: interference at the tail of the transmission, or fading. (3) *if(SNR - last_SNR_val > increase_thresh)*: interference at the head of the transmission, or multipath. The technique is illustrated in the overall system in Figure 7, where the correlation threshold for a data packet raises a signal which streams the samples to the SNR monitor. The final conditional is to detect the carrier as idle; then the fast-ACK is generated.

Stage 3: To satisfy fast-dependent packet generation, the dependent packet must be pre-modulated and stored on the radio hardware, for which we provide a mechanism on the control channel. Pre-modulation restricts the dependent packet to not contain fields dependent on the initiating packet (e.g., a MAC address). However, it still permits many dependent packets like those in current protocol standards (e.g., ACKs, RTS/CTS). For example, despite 802.11's requirement for a destination address in an ACK packet, we can still develop and evaluate an 802.11-like protocol where senders assume the destination of the ACK based on data transmissions. We remind the reader that a goal of our work is to enable MAC implementations and building blocks for novel MAC designs, not to necessarily support every current protocol to its specification. Future work could be in the development of a technique which extracts part of an incoming signal (e.g., destination address) and then performs additional processing to use this raw signal in a pre-modulated dependent packet. This would essentially enable dynamic fast-dependent packets, without the interaction of the host. We do not explore this in the scope of our work.

Fast-Dependent Packet Evaluation: To illustrate the fast-dependent packet generator, we evaluate an implementation of the fast-ACK generator outlined in the description of *stage 2*. First, we use the control channel to setup a matched filter which detects the framing bits and the attached node's address (satisfying stage 1). Then, we pre-modulate an ACK that uses the broadcast address as the destination address for all active nodes to parse it (satisfying stage 3).

To evaluate the SNR monitoring technique, and further evaluate the matched filter's ability to detect packets in a real world scenario, we use a 2 USRP-node setup in the ISM band for presence of 802.11 and Bluetooth devices, incorporating real world interference in our results. We detected 6 active 802.11 devices within interference range, but ensured that none were within 40 feet of either node. To test in adversarial conditions with multipath interference, the two USRPs were placed in separate rooms with no direct line of sight. The matched filter and fast-ACK technique are enabled at the receiver, for which we transmit 10000 frames to at 1Mbps. These frames are considered the ground truth for the matched filter, which we are trying to determine the accuracy of in detecting the frames. Full decoding of the data packets at the host is used as the ground truth for the fast-ACK generator. If the full decoder successfully decodes the frame, and the SNR monitor triggers a fast-ACK, it is considered success. If the SNR monitor chose to not generate a fast-ACK in this scenario, it is considered failure. An additional failure scenario is triggering a fast-ACK when the host could not decode the frame.

For the 10000 frames transmitted, we find that the matched filter is able to detect the transmissions with 100% success rate, reinforcing the simulation results from Section 5.4.2 with real world signal propagation properties. Of the 10000 frames, 460 transmissions were not decodable. Using the SNR monitoring technique *we detect 457 of the corrupted frames for a failure rate of 0.6%*. Inspection of the 3 misses could not determine the cause of transmission failure. The error rate of not generating an ACK, when one should have been, is 4%.

There are implications to incorrectly generating ACKs, which the MAC can be designed to recover from, or higher layers such as TCP can be relied on. Our evaluation further explores the matched filter's accuracy and illustrates the ability to implement fast-dependent packets. Reducing the error rates seen by our technique is future work, either by improving the SNR monitoring technique, or introducing other fast-ACK techniques. An example for improvement would be detecting multipath during SNR monitoring, which is a property that can reduce decoding probability.

5.6 Access to Physical Layer Information and Fine-grained Radio Control

The underlying radio hardware in an SDR platform has many controls that are not configured by the transmitted sample stream (e.g., transmission frequency and power), and can make many observations that are not easily derived from the input sample stream (e.g., RSSI). We use our control channel between the SDR hardware and host to expose these controls and physical layer information to the MAC protocol implementation. Many existing network interface use similar designs for setting the transmission channel and obtaining RSSI measurements. One key difference is that our interface operates on blocks of samples instead of packets.

Physical Layer Information: Access to physical layer information at all other layers in the processing chain is important for supporting common cross-layer optimizations. This can be seen through recent work where per-bit confidence levels are used to perform partial packet recovery [7]. In our design, information from the SDR can be sent to the host using either the control channel or per block meta-data. We use this mechanism to report RSSI to the host. Note that the host could calculate RSSI using the raw samples, but an RSSI value which takes into account the gain or attenuation in the RF stages is only available at the radio hardware. The control protocol is easily modified to support reporting additional properties, however, developers must reprogram the FPGA to report the desired values.

Radio Control: We implement a set of radio hardware control messages on the control channel (Section 4.2) that can be synchronized with packet transmissions using the timestamp. For example, by placing a control block with a timestamp T before a data packet on the bus, which uses a *NOW* timestamp, the radio will be reconfigured at time T and the data packet will be transmitted immediately after the reconfiguration. This can be used to implement common techniques such as rapid frequency hopping. Unfortunately on the USRP, the daughterboards are tuned directly from the FX2 USB controller using the I^{2}C bus, which has no connection to the FPGA. Therefore, we cannot issue daughterboard commands from the FPGA using the control channel and hardware clock to implement rapid frequency hopping. The USRP2 tunes the daughterboards directly from the FPGA. Therefore, if our design was implemented on the USRP2, unavailable at the time, rapid frequency hopping could be achieved.

6 MAC Evaluation

We now provide end-to-end results for a Bluetooth-like TDMA protocol and 802.11-like CSMA protocol. The protocols use the *split-functionality* design described in Section 5 and we compare their performance with that of full host-based implementations.

6.1 Bluetooth-like TDMA Protocol

To illustrate the effectiveness of the overall system design, we implement a tightly timed Bluetooth-like TDMA protocol. Like Bluetooth, the network (piconet) consists of a master and a maximum of 7 slaves. The slaves communicate with the master in a round-robin fashion within a slot time of $625\mu s$. Unlike Bluetooth, our protocol fixes its frequency instead of hopping (a limitation of the USRP discussed in Section 5.6), varies slightly in synchronization (bypasses *pairing*), and the slot guard time is varied for evaluation.

Each slave in the network synchronizes with the start of a round by listening for the master's beacon, and calculates the start of transmission (Section 5.1) as the logical synchronization time T. The beacon frame also carries the total number of registered slaves (N) and the guard time (T_g). The slave can then compute the total round time, which must account for the master: $T_r = N + 1 * (T_s + T_g)$, where T_s is the slot time ($625\mu s$). The start of round k is computed as: $T_k = T + T_r * k$. We remind the reader that this is a logical time kept at each node, taken from the beacon frame which is a global reference point. Global hardware clock synchronization is explored in Section 6.2. Finally, each slave's slot offset is computed from its node ID (n), $\delta_n = n * (T_s + T_g)$, which is then used to compute the local start time of slave n's slot in round k: $T_{n(k)} = R_k + \delta_n$.

6.1.1 TDMA Results

We use two metrics in our evaluation: ability to maintain tight synchronization and overall throughput. The synchronization error at the master is 15ns, computed by measuring the actual spacing of 1000 beacons using a monitoring node (discussed in Section 5.1.2). This illustrates the tight timing of the master's beacon transmissions. To measure the synchronization error at the slaves, we record the calculated timestamps of 1000 beacons at 4 slaves. Each timestamp should be exactly T_r apart from the next. The absolute error in spacing represents shifts in the slave's calculation of the start of the round. We find the maximum error of the 1000 beacons at all 4 slaves to be 312 nanoseconds, with an average of 140ns. This answers the question of our platform's ability to obtain tight synchronization at both transmitters (master) and receivers (slaves).

We compare a split-functionality implementation to a host implementation, which differ in their guard times. A guard time of $1\mu s$ is used for the split-functionality implementation, which is nearly 3 times the maximum error. We use our round trip host and radio hardware delay measurements from Section 2.1, which accounts for both transmissions and reception timing variability, to estimate the host guard time needed. A guard time of 9ms would be needed to account for the maximum error, however, this delay occurs rarely and we therefore present results using a generous guard time of 3ms (approximately $3 *$ sdev) and a more realistic guard time of 6ms based on our recorded delay distribution.

We perform 100KB file transfers, varying the number of registered slaves and presenting averaged results across 100 transfers in Figure 10. The *split-functionality*

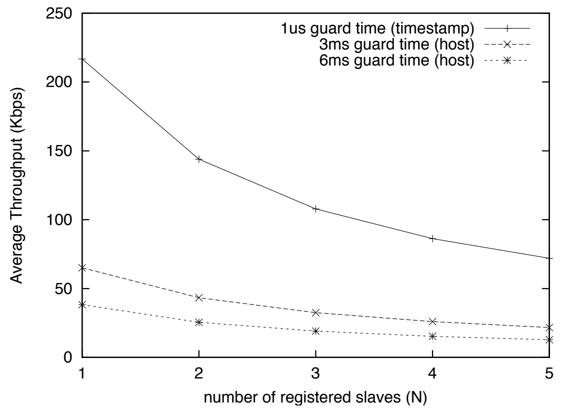


Figure 10: TDMA throughput comparison results.

implementation is able to achieve an average of 4 times the throughput of the host based implementation. While we had only been able to answer the question of obtaining synchronization, we find that throughout the full transfers no slave drifts into another slot period using only the initial beacon for synchronization, illustrating the ability to *maintain* tight synchronization. These results are promising for the development of TDMA protocols on the platform.

6.2 Additional TDMA Protocols

Another common TDMA implementation is the use of global clock synchronization. We extend the Bluetooth-like protocol to use global clock synchronization on the platform rather than the logical clock. The implementation design is as follows. The global clock in the network is the clock of the master, to which all slaves synchronize via beacon frames. In addition to the information sent in each beacon frame described in Section 6.1, the master includes the timestamp at which the beacon is locally scheduled for transmission.

For global synchronization, the slave takes its estimated local time of the master's beacon transmission and subtracts the incoming global clock timestamp included in the beacon to calculate δ, the local clock offset from the master. The error is within 312ns plus over-the-air propagation delay. The MAC framework can now synchronize to the global clock with a command packet (Section 4.2) which adds δ to the local clock. Another option is to use a timestamp transformation where the MAC adds δ to all timestamps. Using this methodology, we are able to achieve measurement results similar to those in Figure 10 using global synchronization.

6.3 802.11-like CSMA Protocol

We implemented two 802.11-like CSMA MAC protocols, one fully on the host CPU and one using our

	pairs	Avg (Kbps)	min	max
platform	1	408	387	415
host	1	215	190	240
platform	2	205	201	210
host	2	112	101	130

Table 2: 802.11-like CSMA protocol per-pair results.

split-functionality optimizations including on-board carrier sense (Section 5.2), dependent packet ACK generation (Section 5.5), and backoff (Section 5.3). The MAC implements 802.11's clear channel assessment (CCA), exponential backoff, and ACK'ing. Our protocol does not implement SIFS and DIFS periods; this work is in progress. For space reasons, we focus our description on how the 802.11-like protocol uses our architecture.

The host-based implementation places all functionality on the host CPU, including carrier sense, ACK generation, and the backoff. The optimized implementation uses the matched filter and SNR monitoring for ACK generation, and performs carrier sense and backoff on the radio hardware. We configure the USRPs for a target rate of 0.5Mbps, and run 100 1MB file transfers for each implementation using a center frequency of 2.485GHz in an attempt to avoid 802.11 interference. This allows us to present results that highlight the differences in the implementation without the effect of uncontrolled interference. We also vary the number of nodes in the network, where each pair of nodes performs a transfer.

The results for the two implementations are shown in Table 2. We see significant performance increases from the use of the *split-functionality* implementation. This nearly doubles the throughput on average, likely due to the time saved in decoding to generate the ACK, and the delays associated with carrier sense and backoff. We note that the matched filter detected every framing sequence, and the fast-ACK generation technique only failed 2 times over the total number of runs. To recover from these failures, we implemented a feedback mechanism on the host that checks the SNR monitoring technique's decision and retransmits. This is needed since we did not use a higher-layer recover mechanism like TCP.

7 Related Work

We review related work in the area of MAC development. Existing platforms mostly use the extremes of the design space where either the majority of functionality is fixed on the network card (*Traditional NICs*), or perform all processing at the host (*Software-defined Radios*).

7.1 Traditional NICs

Several efforts [13, 4, 16] have built new MAC protocols on top of existing commercial NICs (e.g., 802.11 cards). Unfortunately, commercial 802.11 cards implement the bulk of the MAC functionality in proprietary microcode on the card, limiting what functions can be changed by researchers. As a result, this approach is not very satisfactory: the range of MAC protocols that can be implemented is limited and performance (e.g. throughput, capacity) is often poor from the MAC needing to be implemented on the host. For example, past efforts have mostly implemented TDMA-based schemes.

7.2 Software-defined Radios

Software-defined radios (SDRs) provide a compelling architecture for flexible wireless protocol development since most aspects of both the MAC and physical layer are, by design, implemented in software and thus in principle, easy to modify. However, so far, SDR efforts have focused on implementing the physical layer [19] while MAC and higher layer protocol development has received little attention.

Recent work by Schmid et al [14] examines the impact of increased latency in software-defined radios using GNU Radio and the USRP. The authors address how the bus latency creates "blind spots" that increase collision rates when carrier sense is performed at the host, and how pre-computation of packets is not possible without fully demodulating (at the host), resulting in larger interframe spacing. Our design provides solutions for both of these issues in Sections 5.2 and 5.4, respectively. Bus delay measurements were also taken by Valentin et al [18].

On top of these hardware challenges, the original streaming-based design of GNU Radio and the fixed size data limitation on its blocks prevents packet processing. Dhar et al [3] take the approach of integrating the Click modular router [12] with GNU Radio. GNU Radio blocks are imported into Click to handle the physical layer, while Click is used to implement the MAC layer. Additionally, the authors interface with the USRP to provide a full SDR. Another approach extended the GNU Radio architecture with *m-blocks* [2], blocks that allow variable length data passing and include meta-data that can be used to represent packets. Our work is complementary to the above efforts: while they focus on a MAC development environment on the host, we focus on the partitioning of MAC layer processing between the host and radio hardware. Our architecture and results also do not depend on a particular environment on the host.

A number of groups have developed software radios with architectures that differ from the current GNU Radio and USRP design by including a CPU on the radio hardware (NC-CPU), either as a separate component or as a core on the FPGA. Examples include the Rice University Wireless Open-Access Research Platform (WARP) [20] and USRP2. These designs are more expensive, but they offer additional flexibility for partitioning the MAC. However, there is still a non-trivial delay (compared with traditional radios) due to physical layer processing and queueing. The NC-CPU is also likely to be slower than the host CPU, increasing the processing delay. Finally, in deployed products based on this architecture, the NC-CPU is likely to be off-limit to users, similar to the current situation with commercial wireless cards. As a result, we expect that our architecture will be useful this type of platform as well.

8 Conclusions

In this paper, we presented a set of techniques that support the implementation of diverse, high-performance MAC protocols on software radios. The work is motivated by the observation that a single one-size fits all MAC protocol cannot meet the demands of increasingly diverse deployments and application loads. Software radios offer flexibility, but their architecture, specifically the delay between the host and the radio frontend, has traditionally been a problem for MAC protocols. We introduce a split-functionally approach, which addresses this problem, and show that it enables the implementation of a set of core MAC functions. An implementation for the USRP and GNU Radio, along with the implementation of an 802.11-like and Bluetooth-like protocol, shows the approach is effective. To our best knowledge, these protocol implementations are the first high-speed, bi-directional MAC implementations for the GNU software radio platform. For future work, we plan to implement a more diverse set of MAC protocols to further evaluate our design and implement the architecture on different SDR platforms to evaluate its generality.

Acknowledgments

We thank the GNU Radio community for the help provided, especially the support from Eric Blossom and Matt Ettus, and their collaboration in the design of the control channel. A sincere thank you to Brian Padalino for the constant feedback and guidance throughout our work. This work was supported by grant CNS-0626827 from the National Science Foundation.

References

[1] A. Akella, G. Judd, S. Seshan, and P. Steenkiste. Self-management in chaotic wireless deployments. In *ACM MobiCom*, pages 185–199, 2005. ISBN 1-59593-020-5. doi: http://doi.acm.org/10.1145/1080829.1080849.

[2] BBN:ArchChanges. BBN Technologies Corperation, GNU Radio Architectural Changes (m-block). `http://acert.ir.bbn.com/downloads/adroit/gnuradio-architectural-enhancements-3.pdf`.

[3] R. Dhar, G. George, A. Malani, and P. Steenkiste. Supporting Integrated MAC and PHY Software Development for the USRP SDR. In *IEEE Workshop on Networking Technologies for Software Defined Radio (SDR) Networks*, Reston, 2006.

[4] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. C. Sicker, and D. Grunwald. MultiMAC - An Adaptive MAC Framework for Dynamic Radio Networking . In *IEEE DySPAN*, 2005.

[5] S. Gollakota and D. Katabi. Zigzag decoding: Combating hidden terminals in wireless networks. In *ACM SIGCOMM*, New York, NY, USA, 2008. ACM Press.

[6] GR. Gnu radio. `http://www.gnu.org/software/gnuradio/`.

[7] K. Jamieson and H. Balakrishnan. Ppr: partial packet recovery for wireless networks. *SIGCOMM Comput. Commun. Rev.*, 37 (4):409–420, 2007. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/1282427.1282426.

[8] S. Katti, D. Katabi, H. Balakrishnan, and M. Medard. Symbol-level network coding for wireless mesh networks. In *ACM SIGCOMM*, New York, NY, USA, 2008. ACM Press.

[9] kuagile. Kansas university agile radio. `https://agileradio.ittc.ku.edu/`.

[10] M.-H. Lu, P. Steenkiste, and T. Chen. Flexmac: a wireless protocol development and evaluation platform based on commodity hardware. In *WiNTECH '08: Proceedings of the third ACM international workshop on Wireless network testbeds, experimental evaluation and characterization*, pages 105–106, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-187-3. doi: http://doi.acm.org/10.1145/1410077.1410102.

[11] A. Mishra, V. Shrivastava, D. Agrawal, S. Banerjee, and S. Ganguly. Distributed channel management in uncoordinated wireless environments. In *ACM MobiCom*, pages 170–181, 2006. ISBN 1-59593-286-0. doi: http://doi.acm.org/10.1145/1161089.1161109.

[12] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The click modular router. volume 33, pages 217–231, New York, NY, USA, 1999. ACM. doi: http://doi.acm.org/10.1145/319344.319166.

[13] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. SoftMAC - Flexible Wireless Research Platform. In *Fourth Workshop on Hot Topics in Networks (HotNets)*, 2005.

[14] T. Schmid, O. Sekkat, and M. B. Srivastava. An Experimental Study of Network Performance Impact of Increased Latency in Software Defined Radios. In *WiNTECH'07*, 2007.

[15] A. Sharma and E. M. Belding. Freemac: framework for multi-channel mac development on 802.11 hardware. In *PRESTO*, pages 69–74, 2008.

[16] A. Sharma, M. Tiwari, and H. Zheng. MadMAC: Building a Reconfigurable Radio Testbed Using Commodity 802.11 Hardware. In *IEEE Workshop on Networking Technologies for Software Defined Radio Networks*, Reston, 2006.

[17] USRP. The universal software radio peripheral. `http://www.ettus.com/`.

[18] S. Valentin, H. von Malm, and H. Karl. Evaluating the gnu software radio platform for wireless testbeds. In *Technical Report TR-RT-06-273*, 2006.

[19] Vanu. Vanu software radio systems. `http://www.vanu.com`.

[20] WARP. Rice university wireless open-access research platform (warp). `http://warp.rice.edu`.

Antfarm: Efficient Content Distribution with Managed Swarms

Ryan S. Peterson Emin Gün Sirer
{ryanp,egs}@cs.cornell.edu
Department of Computer Science, Cornell University
United Networks, L.L.C.

This paper describes Antfarm, a content distribution system based on managed swarms. A managed swarm couples peer-to-peer data exchange with a coordinator that directs bandwidth allocation at each peer. Antfarm achieves high throughput by viewing content distribution as a global optimization problem, where the goal is to minimize download latencies for participants subject to bandwidth constraints and swarm dynamics. The system is based on a wire protocol that enables the Antfarm coordinator to gather information on swarm dynamics, detect misbehaving hosts, and direct the peers' allotment of upload bandwidth among multiple swarms. Antfarm's coordinator grants autonomy and local optimization opportunities to participating nodes while guiding the swarms toward an efficient allocation of resources. Extensive simulations and a PlanetLab deployment show that the system can significantly outperform centralized distribution services as well as swarming systems such as BitTorrent.

1 INTRODUCTION

Content distribution has emerged as a critical application as demand for high fidelity multimedia content has soared. Large multimedia files require effective content distribution services. Past solutions to the content distribution problem can be categorized into two approaches, namely client-server systems and peer-to-peer swarming systems, whose fundamental limitations render them inadequate for many deployment environments.

In the client-server approach to content distribution, the content owner operates a set of servers that provide the content to every client without tapping into any client-side resources. The presence of a central authority simplifies the design of client-server systems, exemplified by YouTube and Akamai: provisioning the network simply requires purchasing sufficient bandwidth for the desired quality of service and the targeted number of clients; accounting and admission control can be handled by the servers; clients can be prioritized and bandwidth can be dedicated to desired transfers at fine granularity. The chief drawback to the client-server approach is its cost and feasibility: the distributor must bear the entire bandwidth cost of distributing the content, and operating a high-bandwidth data center for a large client population can be prohibitively expensive [11].

Peer-to-peer swarms offer an emerging alternative, where clients interested in downloading a file provide content to other clients interested in the same file. Swarming protocols transfer part of the bandwidth cost from centralized servers to the participants and their ISPs by taking advantage of the additional upload capacity offered by downloading peers. This redistribution of costs reduces the bandwidth burden on the servers, helps improve download times for clients, and reduces ingress bandwidth demand for ISPs. Swarming protocols proposed to date, including BitTorrent [1], Avalanche [24], and Dandelion [52], have been designed to resist technical and legal attacks by avoiding management and centralization. This design choice has led to protocols that lack coordination among peers, rely solely on directly-obtained measurements to avoid trusting information relayed by peers, and depend on randomization to thwart adversaries. The highly decentralized nature of existing unmanaged swarming systems leads to a performance penalty for legitimate content distributors.

To understand why unmanaged swarming architectures fail to make efficient use of bandwidth in multi-swarm environments, imagine a content provider with two movies to distribute to two sets of users using a set of seeders[1] over which they have full control and at which both movies are replicated. Depending on the size of the swarms and the nature of the peers that make up each

[1]In this paper, seeders are trusted servers managed by the coordinator that distribute data blocks to peers. This is in contrast with BitTorrent terminology, where seeders are altruistic peers that have finished downloading a file and provide content without further downloads themselves.

swarm, the two swarms may have vastly different internal dynamics. Seeders and peers with blocks belonging to multiple swarms face a difficult choice: which swarm should they reward with their upload bandwidth? Simple heuristics, such as round-robin, are unlikely to work well because they do not take swarm dynamics into account. The default BitTorrent behavior, which awards download slots to the peers with a proven track of fast downloads, works well within a single torrent, but can lead to wholesale starvation in a multi-torrent setting.

The fundamental problem is one of global optimization: the seeders should award their bandwidth such that download times across all swarms are minimized. Current swarming protocols lack the mechanisms to compute and operate at this point. Consequently, administrators that run torrent sites manually prune old torrents and reallocate bandwidth to more popular downloads by hand. This approach is not guaranteed to achieve a good allocation of bandwidth, leads to the "heavy-tail" problem where old, unpopular torrents are difficult to find, and does not scale.

This paper describes an efficient content distribution system, called Antfarm, based on *managed swarms*. The goal of Antfarm is to distribute a large set of files to a potentially very large set of clients. Managed swarms introduce a hybrid approach to swarming systems in that they permit a *coordinator*, typically managed by the content distributor, to control the behavior of the swarms.

Antfarm is designed to maximize the system-wide benefit of the critical resource, seeder bandwidth. Each Antfarm peer provides resources to other participants, receives unforgeable *tokens* in return, and receives credit for its cooperation by presenting these tokens to the central coordinator. The Antfarm token protocol forces each participant to divulge its upload contributions to the swarm coordinator, which enables the coordinator to determine swarm dynamics and allocate bandwidth to competing swarms. This enables the coordinator to exert control while enabling peers to use microoptimizations, such as optimistic unchoking for peer discovery, tit-for-tat for peer selection, and rarest first, to improve the efficiency of swarming downloads. Overall, the Antfarm transport protocol makes the system resistant to attacks through unforgeable tokens, reveals a coarse-grain view of the network to the central coordinator, and enables the coordinator to adopt and enforce a chosen bandwidth allocation strategy.

The key contribution of this paper is the design of an efficient and scalable coordinator for multiple, concurrent swarms. Given the internal dynamics of a set of swarms, we show how to optimize bandwidth among the swarms such that average download latencies are minimized across all peers. If desired, the algorithm can guarantee a minimum service level to certain swarms, avoid starvation, and enforce prioritization among swarms. Minimizing the average download latency in turn enables a content distributor to achieve the best possible service from the available bandwidth.

This paper makes two additional contributions for achieving high throughput in a practical multiple-swarm download service. First, the paper presents a wire-level protocol for accurately measuring the internal dynamics of individual swarms by making peer contributions evident to the coordinator, enabling the coordinator to optimally allocate bandwidth among the competing swarms. Second, a full implementation of the protocol, accompanied by extensive simulations and a deployment on PlanetLab, quantifies the performance of Antfarm against a client-server system and BitTorrent. In our experiments, Antfarm achieves aggregate bandwidths up to a factor of five higher than BitTorrent, and the protocol scales well with increasing peers and swarms.

The rest of this paper is structured as follows. The next section describes the Antfarm system and the central optimization that underlies the approach. Section 3 outlines the protocol that Antfarm uses for data distribution. Section 4 shows that the system achieves high performance. Section 5 describes related work and highlights Antfarm's differences, and Section 6 summarizes our contributions.

2 APPROACH

Antfarm is based on a hybrid peer-to-peer architecture that utilizes resources provided by peers according to an optimal strategy for managing multiple swarms computed by a coordinator. Each coordinator can manage multiple swarms, a single peer may participate in swarms managed by multiple coordinators, and coordinators may be physically replicated to scale with the number of peers and swarms. For simplicity, we assume a single coordinator in the following discussion and address the issue of scale in Section 3.

The coordinator's central task is to achieve the shortest possible download times across multiple swarms. Finding the right allotment of bandwidth among swarms is best viewed as a constrained optimization problem. The primary constraint is the available bandwidth at the seeders. The primary input to this optimization problem is the inherent *response curve* of each swarm. The response curve represents the swarm bandwidth as a function of allocated seeder bandwidth. It depends on the number of peers in the swarm, number of seeders, spare bandwidth on upload and download links of swarm participants, and the distribution of unique blocks. Peers' local decisions also influence their swarms' response curves, as peers can advertise a lower upload bandwidth capacity than they are capable of providing. However, the Antfarm wire protocol, discussed in Section 3, encourages peers to cooperate within their swarms, granting the coordinator more available bandwidth to optimally allocate among all swarms in the system.

Response curves embody the critical properties of each swarm and have a characteristic shape—a fact we exploit in this work. Figure 1 illustrates the characteristic form of the response curve for a homogeneous swarm with static membership; for illustration purposes in this example, peer download capacities exceed upload capacities, and the set of peers does not change throughout the download. When the seeder bandwidth is small, the peers in the swarm have unused upload and download capacity. In this regime of operation (region A), the swarm's aggregate bandwidth increases rapidly with the seeder bandwidth, since peers can use their spare upload bandwidth to forward new blocks to other peers. Each individual block the seeders feed into the swarm will be shared among many peers, highly leveraging the bandwidth committed by the seeder. Once the peers in a homogeneous swarm have saturated their uplinks, the marginal benefit from additional seeder bandwidth drops significantly. In this regime (region B), any additional bandwidth that a peer receives only benefits that peer, since saturated upload links render it unable to forward the data to other peers. Finally, once downlinks of swarm participants are saturated (region C), the swarm has reached its maximum aggregate bandwidth. Further bandwidth provided by the seeders will not impact download latency. If download capacities are lower than upload capacities, region B will simply not exist, yielding a response curve with only two regions.

A coordinator relies on two key properties of response curves to maximize the achieved aggregate swarm bandwidth while respecting the seeder bandwidth constraint. First, response curves are monotonic: a swarm's aggregate bandwidth will never decrease as a result of increasing the seeder bandwidth to the swarm. Second, response curves are concave; that is, their derivatives monotonically decrease over possible seeder bandwidths. Concavity implies that a swarm's aggregate bandwidth exhibits diminishing returns as the seeders increase their bandwidth to the swarm. When the seeders increase their bandwidth beyond a swarm-specific threshold, the peers' uplinks and downlinks saturate, decreasing their ability to receive and forward data from the seeders and other peers.

Real-life swarms are more complex than the idealized swarms discussed above in that they may comprise heterogeneous hosts and exhibit peer churn. They nevertheless exhibit several critical properties that Antfarm exploits. In heterogeneous swarms, where peer uplinks and downlinks are nonuniform, the transitions between the disparate regions of the response curves are smoother. This is because different peers' upload and download capacities saturate at different points, smoothing the discontinuous transition seen in a homogeneous swarm. In addition to heterogeneity, real swarms exhibit peer churn, where peers can join at any time and leave due to failure, cancellation, or completion. Such membership changes shift the response curve because their influence affects the swarm's dynamics, but do not violate the monotonicity and concavity properties outlined above. Section 3 describes how Antfarm maintains an accurate view of the system and adjusts its behavior in the presence of dynamic membership.

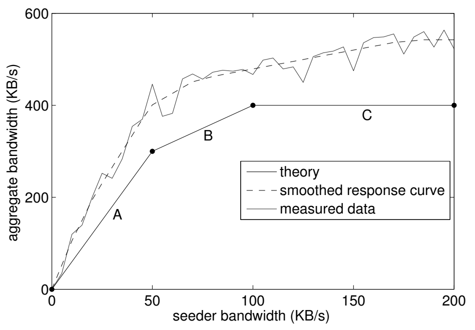


Figure 1: **Response curves of a theoretical homogeneous swarm and a measured heterogeneous swarm on PlanetLab.** Aggregate bandwidth increases rapidly as seeder bandwidth increases (A) until peer uplink capacity is exhausted (B) and reaches its maximum when downlinks are saturated (C).

The monotonicity and concavity of swarm response curves form the foundation of Antfarm's multiple-swarm optimization. Intuitively, when a seeder is supporting a swarm that has a large number of saturated peers, such as in regions B or C in Figure 1, it should reduce its bandwidth to that swarm and divert it to a swarm whose peers can readily share additional bandwidth. More generally, given a response curve for each swarm Antfarm is currently distributing, the coordinator "climbs" each of the curves, always preferring the steepest curve, until it has allocated all seeder bandwidth. The resulting *point of operation* on each curve represents the amount of bandwidth the seeders plan to feed to each swarm and the expected aggregate bandwidth within each swarm based on the seeder bandwidth. Given each swarm's measured response curve, this allocation of seeder bandwidth is optimal [40]: decreasing the seeder bandwidth to one swarm in favor of another will not improve the overall performance of the system. Antfarm's allocation of seeder

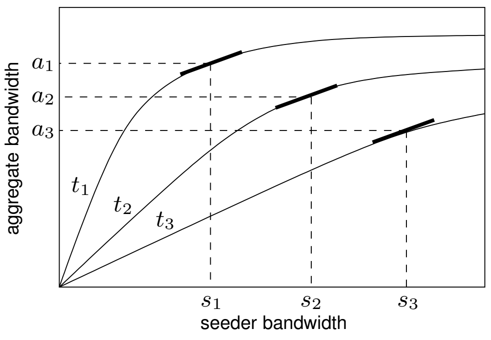


Figure 2: **Optimal bandwidth allocation for three concurrent swarms.** The Antfarm coordinator awards seeder bandwidth by hill-climbing the steepest response curves first until all available bandwidth has been allocated.

bandwidth ensures that the content distributor achieves the highest performance possible from its servers' bandwidth.

The optimization process described above may reach a point at which the seeders have excess bandwidth to award, yet the derivatives of multiple response curves are identical, indicating that multiple swarms offer the same global benefit (Figure 2). In such cases of equivalent global benefit, Antfarm uses a tie-breaker algorithm to maximize perceived improvement by peers. Suppose that two swarms t_1 and t_2 have response curves with equivalent slopes at seeder bandwidths s_1 and s_2, corresponding to swarm aggregate bandwidths of a_1 and a_2, with $a_1 > a_2$. While this indicates that awarding a block to either swarm would improve average download times across the entire network by an equal amount, the incremental benefit to members of t_1, which already enjoy a larger aggregate throughput, is small compared to the relative improvement that members of t_2 would perceive. Consequently, Antfarm breaks ties by awarding bandwidth to swarms with lower bandwidth when multiple response curves have the same slope. This mechanism ensures that the system maintains its primary goal of minimizing download time, while the participants receive maximal marginal benefit whenever there is freedom in making a bandwidth allocation that is in line with the primary goal.

3 IMPLEMENTATION

The Antfarm implementation is centered around a token-based wire protocol that implicitly reveals peer dynamics to the coordinator. This section provides an overview of the Antfarm implementation, outlines the wire protocol and the use of tokens, and describes how tokens are used in the larger context of bandwidth allocation. We illustrate the common case first and treat the corner cases stemming from token misuse, peer misbehavior, and overall scalability in Sections 3.4 and 3.5.

3.1 Overview

An Antfarm deployment consists of two types of servers provided by the content provider. *Coordinators* manage the system by issuing tokens, computing response curves, and determining bandwidth allocations. *Seeders* expend their bandwidth to distribute blocks of files to peers. For small deployments, a single server machine can act as both coordinator and seeder, while large deployments will comprise multiple physical hosts.

Antfarm seeders are members of all swarms and distribute data blocks without downloading any themselves. They may be under the direct administrative control of the coordinator, or they may be deployed by ISPs to reduce their ingress bandwidth demand; in either case, they may be geographically distributed to improve bandwidth to peers. Seeders do not demand tokens from peers in exchange for blocks because they do not place resource demands on the system.

Peers interact with coordinators, seeders, and each other to download files. Each peer in Antfarm is identified by a certificate acquired from the coordinator during an initial, one-time registration. Once a connection with a peer has been established and the peer has been authenticated with the coordinator, wire messages identify peers using a public IP address and port pair that is shorthand for the verified certificate. Antfarm assumes that peers are either rational, where the protocol will incentivize them to contribute resources to the global pool, or malicious, where they may behave in a Byzantine manner; the protocol is resilient to such malicious hosts (see Section 3.5).

The Antfarm wire protocol is designed around peer-to-peer data exchange in return for *tokens*. A token is a cheap, unforgeable capability that the bearer may exchange for a data block in a given swarm. Logically, a token is composed of a unique, randomly generated ID string, an expiration time after which the token is invalid, a reference to the intended spender of the token, and a reference to the file for which the token should be spent. The coordinator records these four fields when it mints a new token for a particular peer. A token can only be spent by the peer to which it was issued in exchange for blocks of the designated file; tokens are not interchangeable between swarms.

Downloadable files in Antfarm are described by a ".ant" *swarm description*, analogous to a ".torrent" file, which contains the name of the file, the address and port of the coordinator managing the swarm, data block size, and a hash of each data block.

3.2 A Peer's Perspective

An Antfarm peer joins a swarm by opening a connection to the swarm's coordinator and authenticating itself using its peer certificate. Once a connection has been established, all correspondence with the coordinator and peers occurs with the exchange of protocol messages summarized in Table 1. When a new peer joins a swarm, the coordinator sends the peer a subset of the peers in the swarm and an initial allowance of tokens unless the new peer has a history of malicious behavior. The peer can similarly join additional swarms, acquiring peer lists and initial tokens for each.

The basic data transmission protocol in Antfarm has three phases consisting of peer and block selection, data-for-token exchange, and bandwidth allocation.

Peers may determine their own criteria for selecting peers and blocks. This enables Antfarm peers to perform optimizations based on local information, reducing the burden on the centralized coordinator. The default behavior in Antfarm for peer and block selection is identical to BitTorrent. Peers retain a prioritized list of other peers with which to exchange data blocks (to *unchoke*). The priority order is determined by the running average bandwidth achieved through that peer's history of interactions. Blocks are chosen using a rarest-first algorithm; peers maintain a bitmap of blocks held by each connected peer constructed from block acquisition notifications sent by peers after each block transfer. Since swarming systems that rely solely on local information and randomized interactions may operate at reduced efficiency due to lack of information [30], the Antfarm coordinator uses its global knowledge to influence peer selection. The coordinator monitors each peer's upload history and identifies underutilized peers. It sends lists of such peers as candidates for data exchange through an asynchronous notification. This is an advisory notification that causes the recipient to increase the priority of the named, underutilized peers. This is a no-cost optimization for Antfarm; a peer is under no obligation to follow the recommendations and the protocol's correctness does not depend on the peer-selection algorithm. This process of aiding peer selection could be improved by the use of network proximity measures [19, 33, 41], though our current implementation does not yet include this optimization.

Once a peer (receiver) has chosen another peer (sender) and determined a suitable block for download, it sends a data-exchange request. If the sender has unchoked the receiver, it sends the requested block to the re-

Connections	
handshake	Sent by peers to establish connections; includes the identifier of a file the sender wants to download and the public port of the sender.
handshake_response	Sent in response to a handshake.
join_swarm	Sent to the coordinator to become a swarm member.
leave_swarm	Sent to the coordinator to be removed from a swarm.
time_request	Sent by a peer to the coordinator to get the system time.
time_response	Sent in response to a time_request; contains the time according to the coordinator.
Node state	
choke	Informs the recipient that the sender is not accepting block requests from the recipient.
unchoke	Informs the recipient that the sender is now accepting block requests from the recipient.
interested	Informs the recipient that it has at least one block that the sender needs.
not_interested	Informs the recipient that the recipient does not have any blocks that the sender needs.
have_block	A notification sent to directly-connected peers when a peer receives a new block.
bitfield	Contains a bitfield of all the blocks the sender possesses. Normally sent after establishing a new connection.
Block transfers	
request	A request for a specific block.
block	A block of file data, sent in response to a request.
Swarm info	
peer_request	Sent by a peer to the coordinator to request a set of peers in the swarm.
peer_response	A set of peers' addresses and ports.
good_peers	Sent periodically by the coordinator to each peer to notify them of peers to unchoke.
bad_peers	A notification containing a set of peers the coordinator has identified as malicious.
allocation	Sent by the coordinator to inform peers of the desired allocation of their upload bandwidth.
Token management	
new_tokens	Sent by the coordinator to deliver a set of fresh tokens to a peer.
token_receipt	Receipt for a block transfer; sent from one peer to another in response to a block message.
token_ledger	Contains a set of spent tokens sent to the coordinator in exchange for fresh tokens.
token_replace	Contains a set of fresh tokens sent to the coordinator in exchange for new tokens with later expiration times.

Table 1: **Antfarm wire protocol.** A comprehensive list of peer-peer and coordinator-peer messages. The protocol comprises messages to establish connections, notify peers of progress and status, exchange blocks, and handle tokens.

ceiver. Upon completion of the transfer, a non-malicious receiver checks the hash of the block against the hash specified in the swarm description and sends an unexpired token to the sender of the data block. Each peer maintains a *purse* of unused tokens issued by the coordinator for use by that peer, and a *ledger* of tokens received from other peers in exchange for data blocks. Tokens flow from the purse of the receiver to the ledger of the sender.

Peers communicate periodically with the coordinator to refresh their purses and ledgers. Each unexpired token in the ledger entitles the peer to a fresh token for its purse. This communication takes place every minute in the current implementation. If a newly received token in the ledger is going to expire before the next scheduled refresh, or if the purse contains nearly expired unspent tokens, the peer can preemptively redeem selected tokens for new tokens with later expiration times.

Peers following the above protocol will face a stream of competing requests for data blocks. Peers use a leaky-bucket algorithm to restrict upload bandwidth according to the coordinator-prescribed allocation. Altruistic peers that finish downloading a file may remain in the swarm and continue to upload content, functioning similarly to seeders.

3.3 The Coordinator's Perspective

The coordinator collects statistics on peer network behavior, computes response curves and bandwidth allocations for each peer and seeder, and steers the swarm toward an efficient operating point. It affects these through manipulation of the token supply and direct interaction with cooperative peers. Finally, it keeps track of malicious and uncooperative participants, excising them from the network when their misbehavior affects performance.

The primary task of the coordinator is to monitor network characteristics and swarm dynamics by keeping track of tokens for each data block transaction between peers. Each token the coordinator receives informs the coordinator of the swarm in which a transaction occurred, the specific peers involved in the transaction, and a window of time in which the data block was transferred based on the token's minting and expiration times. This information is sufficient to maintain two key parameters for each peer p: the set of swarms T_p that p is a member of and a rolling average of its upload bandwidth u_p. In addition, the coordinator keeps track of the set of all seeders S and two pieces of state for each swarm: a set P_t of peers in swarm t and a response scatterplot for each swarm, represented as a collection of data points with associated time-decaying weights. Data points decay according to $1/t$ and are removed after 30 minutes.

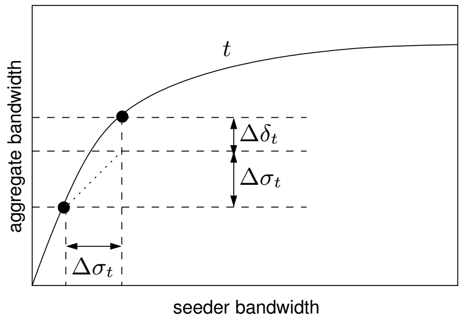


Figure 3: **Bandwidth allocation.** The black dots denote the allocation of bandwidth for swarm t before and after one iteration of allocation. For each $\Delta\sigma$ tasked to a seeder by the hill-climbing algorithm, a randomly selected peer with spare upload capacity is tasked with allocating a corresponding $\Delta\delta$. The dotted line has a slope of 1, accounting for the seeders' contribution to the swarm's aggregate bandwidth.

The coordinator chooses swarms to grant bandwidth based on collected swarm statistics. The response scatterplots are not immediately suitable for use in computing bandwidth allocation, as they contain artifacts due to measurement errors and changes over time, creating false local minima and maxima. The coordinator generates a response curve from a response scatterplot by fitting a piecewise linear function that respects the monotonicity and concavity constraints, contains a segment for each measurement point, and minimizes error using least-squares.

The coordinator computes the amount of bandwidth each seeder and peer should dedicate to each swarm based on the computed response curves, represented as two matrices σ and δ. For each swarm t, $\sigma_{s,t}$ captures the amount of bandwidth seeder s will dedicate to t, and $\delta_{p,t}$ captures the amount of bandwidth peer p is expected to dedicate to t. This determines the critical allocation of seeder upload bandwidth $\sigma_t = \sum_{s \in S} \sigma_{s,t}$ to swarm t in order to achieve a swarm aggregate bandwidth $(\sigma_t + \delta_t)$, where $\delta_t = \sum_{p \in P_t} \delta_{p,t}$ is the bandwidth component resulting from peer-to-peer uploads. The coordinator computes this allocation periodically, every 5 minutes in our current implementation, and also when the area under the curve has changed by more than 10%. In computing σ and δ the coordinator operates under two hard constraints. First, $\delta_p = \sum_{i \in T_p} \delta_{p,t}$ can never exceed p's upload capacity u_p. Second, the node must have the file to seed; a peer will never be tasked to upload blocks of a file it is not interested in downloading. The coordinator determines σ and δ iteratively. Initially, $\sigma_{s,t} = \delta_{p,t} = 0$ for all peers p, seeders s, and swarms t. The coordinator determines the allocation of bandwidth through a greedy hill-climbing algorithm using the computed response curves and its knowledge of the seeders' upload capacities, illustrated in Figure 3. It allocates bandwidth in discrete units to the swarms whose response curves have the highest gradient, breaking ties in favor of the swarm with the lower value of $(\sigma + \delta)$, as described in Section 2. For each increase in seeder bandwidth $\Delta\sigma_t$ to swarm t, the algorithm chooses a peer at random from P_t with spare upload bandwidth and tasks it with uploading an additional $\Delta\delta_t$ to t, as prescribed by t's response curve. The coordinator continues the process until all seeder bandwidth has been allocated. The final peer allocation δ satisfies the two critical constraints described above and ensures that peer transfers within each swarm achieve the previously measured aggregate bandwidth based on the seeders' allocation σ.

Computation of bandwidth allocation is not a highly time-critical task. Delays in network measurements and peer interactions imply inherent delays between computing an allocation and seeing a change in the network. Since the latency of computing the bandwidth allocation is dwarfed by the latency of data exchange, the computation can be performed in the background. The optimization algorithm is linear in the number of peers and grows according to $O(n \lg n)$ with the number of swarms, enabling the system to scale. The primary metric that determines the quality of the solution is the freshness of data on swarm dynamics.

Antfarm's token protocol incentivizes peers to report statistics to the coordinator in a timely manner. A token's expiration time (5 minutes in the current implementation) and spender-specificity force peers to return tokens to the coordinator in order to receive bandwidth in the future. The circulation of tokens reveals enough information to the coordinator to perform the allocation described above.

Token-based economies can suffer from inflation, deflation, and bankruptcy if left unmonitored. Based on analyses of scrip systems [32], the Antfarm coordinator maintains a constant number of tokens per swarm per peer (30 in the current implementation). New peers receive an initial allowance of 30 tokens. As unspent tokens expire, the coordinator redistributes an equal number of new tokens to random peers to prevent a token deficit when peers depart with positive token balances. Token unforgeability prohibits deflation, and token redistribution enables bankrupt peers to acquire new blocks and reintegrate themselves into the swarm.

The coordinator rewards peers that contribute to the system both directly, by offering seeder bandwidth to peers that have donated bandwidth to peers, and indirectly, by suggesting which peers are underutilized. The latter partly influences unchoking decisions as described previously. The coordinator determines this list for each peer by selecting a small subset of the top uploaders to that swarm, chosen randomly from a probability distribution determined by upload bandwidth.

Peer churn and changes in network conditions cause response curves to become stale over time. In addition, transient measurement errors can skew response curves, causing the system to operate suboptimally. Antfarm maintains response curves by actively exploring the swarm's response at different seeder bandwidths. In each epoch, the coordinator randomly perturbs the current bandwidth allocation by a small amount for each swarm, on the order of 5 KB/s (kilobytes per second). Such variances provide additional datapoints for the response scatterplot, enabling the system to overcome false local minima due to transient effects.

The coordinator does not enforce peers' compliance with the coordinator's directives in allocating their upload bandwidth. A peer is free to shift bandwidth away from one swarm in favor of another at its discretion. In such a scenario, the coordinator will simply observe a shift in the swarms' dynamics, which will be reflected in the response curves. In the next epoch, the coordinator will perform a new bandwidth allocation that takes the peer's behavior into account.

3.4 Scalability

The Antfarm coordinator is optimized to ensure that the logical centralization does not pose a CPU or bandwidth scalability bottleneck.

Shuttling tokens to and from the coordinator for each data block transaction is the main source of coordinator bandwidth expenditure. To reduce the burden, Antfarm does not rely on public-key cryptography to issue or exchange tokens. The Antfarm protocol minimizes the size of tokens on the wire, transmitting only relevant fields when tokens change hands. Only a token's ID, file reference, and expiration time are sent on the wire when the coordinator sends fresh tokens, and only the ID and expiration time are sent on the wire when a peer sends another peer a token. Spent tokens sent back to the coordinator are represented with only the token's ID and the identifier of the peer that spent the token. Using 4-byte token IDs, each token exchange requires less than 24 bytes of total bandwidth and less than 16 bytes of bandwidth at the coordinator for each data block of around 32-128 KB.

Antfarm uses highly compact versions of token identifiers to reduce bandwidth. A 4-byte ID is sufficient to disincentivize forgery because the coordinator will detect a malicious peer's attempt to forge a token upon its first failure to produce a legitimate token. In the event that a peer correctly guesses an active token's ID, it is unlikely to correctly identify the token's intended spender. In the worst case, should a peer successfully forge a token, it will only gain one data block for its efforts, whereas failures will lead to remedial action against the peer, described in Section 3.5. Thus, with 4-byte token IDs, several million peers, and several hundred million tokens, the likelihood of a successful, undetected token forgery is around 10^{-8} when tokens are uniformly distributed. With a skewed token distribution where some peers have 100 times more tokens than the average peer, the likelihood might rise as high as 10^{-6}. Downloading ten blocks with forged tokens is as likely as discovering a collision for a cryptographically secure hash function.

The Antfarm coordinator expends its bandwidth to send tokens to peers, receive spent tokens back from peers, and periodically send swarm allocations and lists of top contributors to peers and seeders. To alleviate the bandwidth demands placed on the coordinator, the Antfarm protocol enables the coordinator to be distributed hierarchically. A lead coordinator machine handles computing response curves and determining swarm bandwidth allocations. The remaining coordinators, called *token coordinators*, issue tokens, collect tokens back from peers, and periodically send each peer's upload and download rates to the lead coordinator each time the lead coordinator computes bandwidth allocations. The lead coordinator redirects each peer to a token coordinator based on a hash of the peer's IP address. When a token coordinator receives a spent token from an assigned peer, it applies the same hash function to the IP address of the token's original owner, a field in the token itself, so it can verify the token with the token coordinator that issued it. Thus, each token exchanged between peers involves at most two token coordinators.

Token coordination is an embarrassingly parallel task. The high ratio between token size and data block length ensures that the coordinator bandwidth is leveraged several thousand-fold. Section 4 shows that distributing the coordinator incurs negligible overhead and that the parallel nature of token management enables the system to grow linearly with the number of coordinator machines.

The coordinator performs two periodic CPU-bound tasks: it computes response curves from scatterplots and allocates seeders' and peers' bandwidth. These tasks are computed centrally in order to derive bandwidth allocations based on the most recent measurements. Our current implementation on a 2.2 GHz CPU with 3 GB of memory takes 6 seconds to perform these computations for 1,000,000 peers and 10,000 swarms whose popularities follow a realistic Zipf distribution. The lead coordinator can easily be replicated to mask network and host failures.

3.5 Security

A formal treatment of the security properties of the underlying Antfarm wire protocol is beyond the scope of this paper. Past work on similar, though heavier-weight, protocols [52] has established the feasibility of a secure wire protocol. Consequently, the focus of this section is to enunciate our assumptions, describe the overall goals of the protocol, provide design alternatives, and outline how to mitigate attacks targeting the bandwidth allocation algorithm.

Antfarm makes standard cryptographic assumptions on the difficulty of reversing one-way hashes and assumes that peers cannot snoop on or impersonate other peers at the IP level. Violation of the first assumption would render the Antfarm wire protocol, as well as most cryptographic algorithms, insecure; consequently, much effort has gone into the design of secure hash functions. Violation of the second assumption is unlikely without ISP collusion, and damage is limited to IP addresses that an attacker can successfully snoop and masquerade.

Antfarm requires peers to contribute bandwidth to their swarms, engage in legitimate token-for-block transactions with other peers, and report accurate statistics to the coordinator. The token protocol, coupled with verification at the coordinator, ensures detection of dishonest peers with relatively low overhead.

In order to measure accurate response curves, the coordinator verifies that all token transactions occur within the intended swarm, by the intended peer, and within the intended period of time. The coordinator detects token forgery upon receiving an invalid token from a peer by simply comparing the token ID against its own registry of active tokens. Similarly, the coordinator compares its own record of the intended sender with the spender as reported by the peer returning the token to prevent peers from spending maliciously obtained tokens. Peers are required to report the actual spender in order to receive a fresh replacement token. The coordinator detects all counterfeit tokens, but when it detects an invalid token, it is unable to differentiate the peer sending the token from its ledger from the peer that originally spent the token as the culprit. Therefore, it notifies both peers of the forgery so the honest peer can blacklist the culprit.

To hold peers more accountable for their actions when the coordinator is unable to precisely identify malicious peers, Antfarm peers employ a *strikes* system to record and act on undesirable behavior. Peers maintain a tally of strikes against other peers and disconnect from peers that have exceeded a threshold. By default, misbehaviors that can stem from network congestion, such as a late response to a block request or payment with a recently expired token, result in one strike against the offending peer. Circulating a counterfeit token results in automatic termination of the connection. In general, when the coordinator is unable to determine the identity of a malicious peer, it appeals to the strikes system rather than erroneously penalizing an honest peer. While it is possible to build a centralized reputation system for peers, the current Antfarm implementation avoids this to reduce burden on the coordinator.

Using cryptographically signed tokens can provide stronger guarantees than Antfarm currently does at the cost of additional overhead and complexity. In such a scheme, the coordinator can sign all minted tokens before issuing them to peers, enabling peers to verify that they are exchanging legitimate tokens with each other during each transaction. In addition, if the spender of a token were required to sign the token before sending it, peers could prove the identities of token double-spenders. Token signatures would prevent malicious peers from snooping packets and tampering with tokens without the recipient's knowledge. Antfarm does not implement a cryptographic scheme because the added overhead is not accompanied by a clear increase in performance.

It is possible for Antfarm peers to collude in order to coerce the coordinator into providing their swarm with more bandwidth. In particular, peers could band together and send each other large numbers of tokens without sending each other blocks in exchange. The resulting inflated estimate of that swarm's aggregate bandwidth can lead the system to deviate from a desired allocation. Several techniques mitigate such attacks. First, the coordinator never issues more tokens than strictly necessary to download the file, thereby bounding the impact of fake transactions by the number of Sybils. Second, forcing participants to register with a form of hard identity, such as credit card numbers, can mitigate Sybils [12]. Finally, the coordinator can mandate that peers trade with a diverse set of peers, reducing the effect of collusion among a small fraction of the swarm. Although the token protocol does not eliminate the possibility of malicious behavior, its simplicity and ability to detect malicious activity limit the harm peers can inflict.

3.6 Summary

The Antfarm protocol strikes a balance between micromanaging peers and granting them freedom over block transfers. Tokens that must be returned to the coordinator enable the coordinator to collect accurate statistics on swarm dynamics and peer behavior. Systems such as BitTorrent, which grant peers full autonomy, do so at the expense of control and efficiency. At the other extreme, a centralized solution that precomputes the entire download schedule for all participants would limit peers' ability to quickly determine which peers have blocks they require and retrieve them without intervention. Antfarm provides a hybrid approach that leaves peers free to determine their own local behavior while extracting sufficient information from the network to compute the globally optimal allocation of available bandwidth among swarms.

4 EVALUATION

We have implemented the full protocol described in this paper, as well as a simulator of the Antfarm and BitTorrent protocols. The Antfarm deployment runs on Windows, Linux, and Mac OS X. Both the implementation and the simulator contain optimizations present in version 5.0.9 of BitTorrent, including optimistic unchoke, regular unchoke, and local-rarest-block-first. For the experiments in this section, Antfarm's system parameters (block size=64KB, optimistic unchoke interval=30s, regular unchoke interval=10s) are identical to those found in this version of BitTorrent. We pick upload and download bandwidths representative of cable-connected end nodes.

This section evaluates the performance of the Antfarm protocol in comparison to BitTorrent and traditional client-server approaches. Through simulations, we illustrate scenarios under which BitTorrent misuses its seeder capacity and show how Antfarm can achieve qualitatively higher performance by allocating seeder bandwidth to swarms that provide the highest return. A PlanetLab deployment confirms Antfarm's allocation strategy under realistic network conditions. Lastly, this section shows that Antfarm's coordinator can scale to support large deployments using modest resources.

4.1 Simulations

The differences between Antfarm and BitTorrent in a multi-swarm setting stem from the way the two protocols allocate their bandwidth to competing swarms. Whereas BitTorrent seeders allocate their bandwidth greedily to peers that absorb the most bandwidth, Antfarm allocates the precious seeder bandwidth preferentially to swarms whose response curves demonstrate the most benefit. As a result, there is a qualitative and significant difference between the two protocols; under some scenarios, BitTorrent can starve swarms and perform much worse than Antfarm, while in others with ample bandwidth, seeder allocation may have little impact on client download times. Figure 4 shows Antfarm's performance in comparison to BitTorrent and a traditional client-server system similar to YouTube for two swarm distribution scenarios. In the *bimodal* scenario, there is a single swarm

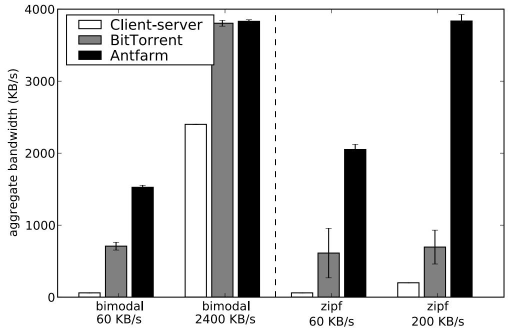


Figure 4: **Aggregate bandwidth for a client-server system, BitTorrent, and Antfarm.** When seeder bandwidth is plentiful, even a client-server model can deliver high throughput. When seeder bandwidth is limited, Antfarm outperforms BitTorrent by allocating bandwidth to swarms that receive the most benefit. Error bars indicate 95% confidence intervals.

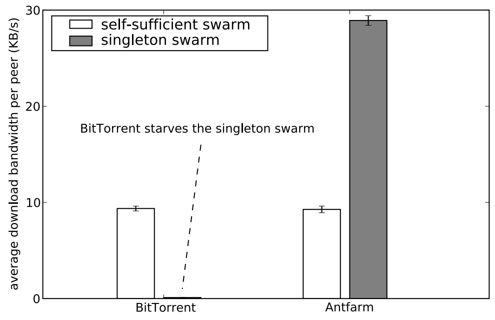


Figure 5: **Bandwidth of a singleton swarm and a large, self-sufficient swarm.** Even though a self-sufficient swarm can saturate its peers' bandwidth without seeder bandwidth, BitTorrent awards bandwidth to peers in the swarm. In contrast, Antfarm awards seeder bandwidth to the singleton swarm because it receives high marginal benefit.

of 30 peers and 30 swarms of one peer each. The *Zipf* scenario comprises swarms of sizes 50, 25, 16, 12, 10, 8, and 5, and 400 singleton participants. Each set of three bars shows the average aggregate bandwidth for a corresponding scenario and seeder bandwidth.

Overall, Antfarm achieves the highest aggregate download bandwidth. In scenarios where there is ample seeder bandwidth, the differences between the three systems are negligible and even a client-server approach is competitive with BitTorrent and Antfarm. As available seeder bandwidth per peer drops, however, swarming drastically outperforms the client-server approach, highlighting the efficiency of peer-to-peer over a client-server system using comparable resources. For the scaled-down but realistic Zipf scenario, Antfarm achieves a factor of 5 higher aggregate download bandwidth than BitTorrent. BitTorrent misallocates bandwidth by preferentially unchoking hosts based on their recent behavior, regardless of their potential to share blocks. In contrast, Antfarm steers the seeder's capacity to swarms where blocks can be further shared among peers.

Antfarm's dynamic bandwidth allocation adapts well to changes in swarm dynamics. A well-known phenomenon is that when swarms become large, they are often able to saturate their peers' uplinks, and sometimes even their downlinks, without the aid of seeder bandwidth. Such *self-sufficient* swarms yield flat response curves. Antfarm's allocation strategy naturally avoids dedicating bandwidth to self-sufficient swarms when there are other swarms that can benefit more. In contrast, BitTorrent does not take swarm dynamics into account, and can end up dedicating seeder bandwidth at the exclusion of available peer bandwidth, leading to a shortage of seeder bandwidth for other, needier swarms.

Figure 5 shows an exaggerated scenario that illustrates this effect. The figure shows the average download bandwidths of peers in BitTorrent and Antfarm of the two swarms. In this scenario, the seeder has a capacity of 100 KB/s, and each peer downloads a 10 MB file with 30 KB/s download capacity and 10 KB/s upload capacity. The self-sufficient swarm saturates peers' uplinks without seeder bandwidth and has a fresh peer arrive every second, resulting in a swarm of approximately 1000 peers. The Antfarm coordinator determines that the self-sufficient swarm does not benefit from seeder bandwidth, and awards bandwidth to the singleton swarm instead. Under Antfarm, the singleton peer is able to complete its download in an average of 6 minutes. BitTorrent fails to provide the singleton swarm any bandwidth over the course of the 20 minute simulation.

The problems with BitTorrent's allocation strategy are compounded in larger, more realistic scenarios. While large swarms are often self-sufficient, smaller non-singleton swarms can receive large multiplicative benefits from the seeder because their peers have available upload capacity to forward blocks. In contrast to the previous experiment, which examined the impact on a swarm at the tail end of the popularity distribution, Figure 6 illustrates the impact of seeder bandwidth allocation on a file of medium popularity. The figure shows the total amount of seeder bandwidth that Antfarm and BitTorrent allocate to a set of self-sufficient swarms, a new swarm of 5 peers, and 32 singleton swarms. It also shows the resulting average download bandwidths of peers in each swarm. The peers have 30 KB/s download capacities and 20 KB/s upload capacities, and the self-sufficient swarms have peer interarrival times of 3, 6, 12, 24, 50, and 100 seconds. In the left-hand graph, BitTorrent ded-

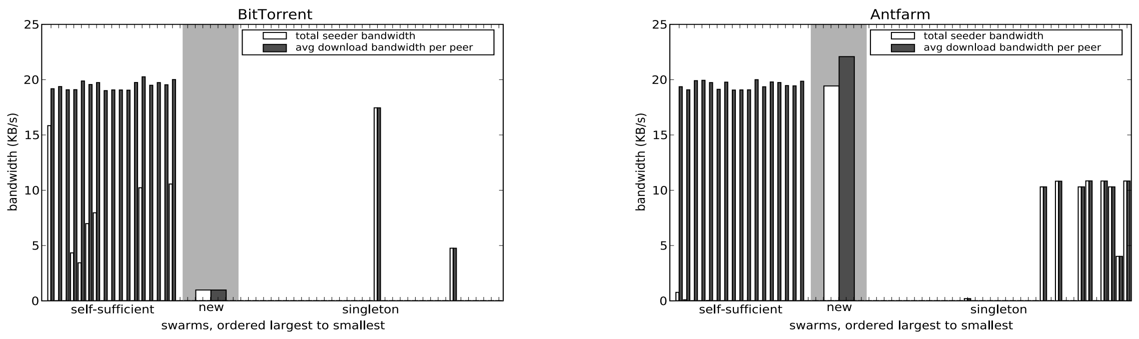


Figure 6: **BitTorrent versus Antfarm serving the middle of the popularity distribution.** The shaded region indicates a new swarm of 5 peers. Swarms to its left are self-sufficient; swarms to its right are singletons. BitTorrent (left) starves the new swarm, favoring to dedicate bandwidth to the many peers in self-sufficient swarms. Antfarm (right) allocates enough seeder bandwidth to the new swarm to saturate its peers' upload bandwidths, and allocates the rest to singleton swarms because they receive high marginal benefit.

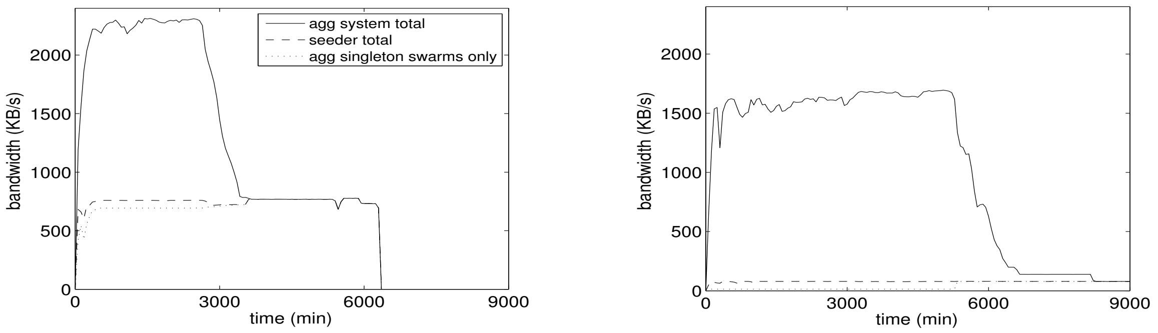


Figure 7: **Time versus bandwidth for Antfarm.** The figures show seeder and aggregate bandwidths of the bimodal experiment with seeder bandwidths of 800 KB/s (left) and 80 KB/s (right). Antfarm follows drastically different bandwidth allocation strategies (dashed and dotted lines) to achieve high throughput (solid lines).

icates almost all of its bandwidth to the self-sufficient swarms, whose peers are already saturated, and some randomly to singleton swarms, which are unable to forward blocks. The right-hand graph shows that Antfarm awards enough bandwidth to the new swarm to saturate its peers' uplinks and dedicates the rest of its bandwidth evenly among several singleton swarms because they receive high marginal benefit. BitTorrent's optimistic unchoking protocol causes it to dedicate its bandwidth to only a few singleton swarms over the 20 minute simulation. Overall, Antfarm achieves an order of magnitude increase in average download speed for the affected swarms without a corresponding penalty for the popular swarms.

Figure 7 shows Antfarm's bandwidth allocation over time to provide insight into Antfarm's strategy. The left-hand graph shows that when seeder bandwidth is plentiful, Antfarm spends the vast majority of its bandwidth on small swarms since they receive the most marginal benefit. When seeder bandwidth is constrained, as shown in the right-hand graph, Antfarm achieves high aggregate bandwidth by preferentially seeding large swarms that can leverage their upload capacity to multiply the benefits from the seeder. As peers of the large swarm complete their downloads at 5000 seconds, the seeder shifts its bandwidth to the singleton swarms. The staircase behavior is due to different swarms completing at different times.

Overall, Antfarm qualitatively outperforms BitTorrent in a multi-torrent setting by allocating bandwidth based on dynamically measured response curves and preferentially serving those swarms that benefit most from seeder bandwith.

4.2 PlanetLab Deployment

We tested Antfarm's performance through a PlanetLab [5] deployment. To demonstrate Antfarm's response curves in practice, Figure 8 shows a measured response curve of a swarm comprised of 25 PlanetLab nodes, each

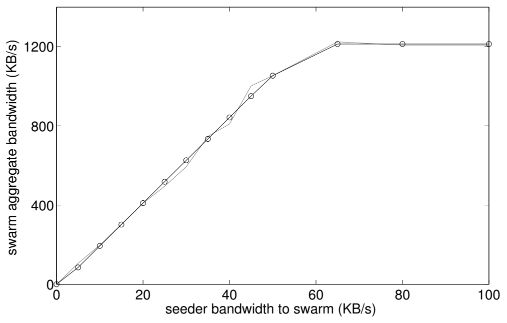


Figure 8: **A response curve for a swarm consisting of 25 PlanetLab nodes, each with an upload capacity of 50 KB/s.** Each data point is based on the average swarm aggregate bandwidth over 10 minutes. Real-world response curves confirm simulations.

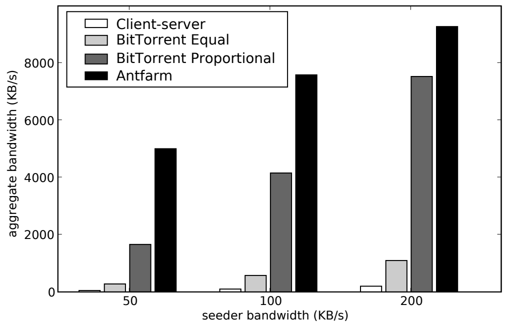


Figure 9: **PlanetLab experiments showing aggregate bandwidth in Antfarm versus BitTorrent and client-server.** 300 PlanetLab nodes are distributed among swarms ranging in size from 1 to 100. Antfarm achieves high average performance by making efficient use of limited bandwidth.

with an upload capacity of 50 KB/s. The graph plots both the response scatterplot and the response curve as computed by the coordinator from the token exchange. The results confirm the simulations.

Figure 9 compares the aggregate bandwidth achieved by Antfarm, BitTorrent, and traditional client-server downloads across 300 PlanetLab nodes, each with an upload capacity of 50 KB/s. Swarms have size 100, 50, 25, 12, 6, 3, and 1. In practice, the stock BitTorrent implementation uploads only a few hand-picked files concurrently; to evaluate BitTorrent in the presence of many swarms, we measured two values by running multiple seeder instances, each with its own upload capacity. *BitTorrent Equal* indicates the aggregate system bandwidth when the BitTorrent seeder splits its upload bandwidth equally among all swarms, including singleton swarms. *BitTorrent Proportional* shows performance when the seeder allocates to each swarm an upload bandwidth proportional to the size of the swarm.

Antfarm outperforms BitTorrent by allocating its bandwidth to the swarms that receive the most benefit. Antfarm's advantages over BitTorrent become more pronounced in systems with many swarms accompanied by relatively small seeder uplink capacities, a realistic scenario for a distribution center with a large number of files and a bandwidth bottleneck. In these experiments, Antfarm outperforms traditional client-server by a factor of between 50 and 100, BitTorrent Equal by a factor of 8 to 18, and BitTorrent Proportional by a factor of 1.2 to 3.

4.3 Scalability

In this section, we examine how the Antfarm coordinator scales. We examine the steady-state bandwidth cost of running a coordinator in a setting where peers download a file made up of 64 KB blocks with upload and download capacities of 64 KB/s.

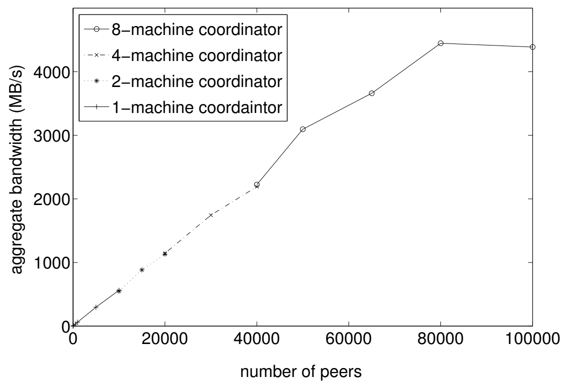


Figure 10: **Aggregate bandwidth of swarms managed by varying sizes of coordinator clusters.** Each coordinator machine runs on a PlanetLab node with an artificial bandwidth cap of 100 KB/s to limit scalability. The task of the token coordinator is embarrassingly parallel; the system capacity scales linearly with the size of the coordinator cluster.

Figure 10 shows the bandwidth consumption at the coordinator as a function of the number of peers based on experiments run on PlanetLab. In the experiment, the lead coordinator and each token coordinator ran on its own PlanetLab node, and peers were simulated across other PlanetLab nodes, engaging in the Antfarm protocol without sending actual data. The results show that even for large numbers of peers, the bandwidth consumption at the coordinator is modest. A coordinator running on a single PlanetLab host suffices for deployments of 80,000 peers or more. To demonstrate the scalability of the hierarchically distributed coordinator, we test a coordinator distributed across multiple PlanetLab hosts in a system with an aggregate bandwidth approaching

5 GB/s. To maximize generated load, peers omit the data exchange but engage in the token protocol with the coordinator. Further, we artificially limit the bandwidth available to each physical coordinator node to 100 KB/s to gain insight into the performance of multiple coordinator nodes running with severe bandwidth constraints. The bottom curve shows the capacity of a single, artificially-bottlenecked coordinator node, which is able to handle the tokens and peer lists of approximately 9000 peers before its performance reaches a plateau. Adding a second such coordinator node doubles the capacity of the system. Because the token coordinators engage in a massively parallel task with little communication overhead, increasing the number of coordinators linearly increases the maximum supported number of peers.

5 RELATED WORK

There has been much past work on content distribution, which can be grouped roughly into work on content distribution networks, token-based systems, and multicast and streaming systems.

CDNs: Content distribution networks are scalable systems used to alleviate server load, reduce download times, and avoid network hotspots. Akamai [31], for example, is a widely deployed infrastructure-based CDN that many content providers rely on to distribute their content. Similarly, cooperative web caching [7,25,27,57, 58] removes load from origin servers. ECHOS [34] proposes distributing servers using a peer-to-peer network of set-top boxes distributed at the Internet's periphery, managed by a single entity that can optimize system performance, but does not address bandwidth management at the servers. Although distributed CDNs scale, the bandwidth cost of operating them resides entirely with the content provider and distributor.

Peer-to-peer CDNs effectively shift bandwidth costs from the content provider to clients. BitTorrent [8] is one of the most popular client-based peer-to-peer CDNs, and studies consistently show that BitTorrent traffic constitutes a significant fraction of Internet traffic [43,53]. Piatek et al. [46] augment the BitTorrent protocol to enable peers to share reputation information through one level of intermediary nodes; it does not address the issue of multiple swarms. CoBlitz [44] is an HTTP-based content distribution network that splits a file into chunks, which are cached at distributed nodes. Choffnes et al. [15] reduce cross-ISP traffic in peer-to-peer systems by harvesting data from existing CDNs for locality information. Shark [3] and ChunkCast [9] reduce client-perceived download latency via a structured overlay, and Coral [23] and Bamboo [50] assist clients in finding nearby copies of data. Antfarm similarly shifts cost to clients; however, it retains control of network behavior by carefully allocating bandwidth to each swarm.

Further, many systems such as the Data Oriented Transfer (DOT) architecture [42, 54] use peer-to-peer swarming to speed up downloads.

Token-based Incentives: Early model and analysis by Qiu and Srikant [49] of BitTorrent's incentive mechanism showed that the system converges to a Nash equilibrium where all peers upload at their capacity. However, more recent work, BitTyrant [45], BitThief [35], and Sirivianos et al. [51], has demonstrated that average download times currently depend on significant altruism from high capacity peers that, when withheld, reduces performance for all users. Further, BitTorrent's tit-for-tat mechanism only operates within an individual swarm; it does not provide information on how to allocate resources, such as seeder bandwidth, among swarms.

Dandelion [52] and BAR gossip [36] avoid the problem of relying on altruism to distribute data. They use a cryptographic fair exchange mechanism that requires a client to upload content to other clients in exchange for virtual credit, which can be redeemed for future service. Microcurrencies [10, 37, 47, 59] similarly rely on cryptographically protected tokens for fair resource exchange, and optionally provide additional features such as spender anonymity. Antfarm's token system is domain-specific and significantly lighter-weight than these approaches.

Decentralized resource allocation in peer-to-peer systems requires incentives for participants to contribute resources. Ngan et al. [39] suggest cooperative audits to ensure that participants contribute storage commensurate with their usage. Samsara [16] considers storage allocation in a peer-to-peer storage system and introduces cryptographically signed storage claims to ensure that any user of remote storage devotes a like amount of storage locally. Both techniques center around audits of resources that are spatial in nature.

Karma [56] and SHARP [22] resource allocation can apply to renewable resources such as bandwidth. Karma employs a global credit bank, with which clients maintain accounts. The value of a client's account increases when it contributes and decreases when it consumes. A client can only consume resources if its account contains sufficient credit. SHARP operates at the granularity of autonomous systems or sites. To join the system a SHARP site must negotiate resource contracts with one or more existing group members. These contracts, in effect, specify the system's expectations of the site and the site's promise of available resources to the system. Accountable claims make it possible to monitor each participant's compliance with its contracts, simplifying audits and making collusion more difficult in SHARP relative to other decentralized peer-to-peer systems.

Streaming and Multicast: Multicast and streaming are alternative designs for distributing content. For instance, the seminal work by Deering proposed IP multicast to efficiently deliver content to multiple destinations [20]. Deployment difficulties with global IP multicast [18] led to application-level multicast systems such as End System Multicast [14], Your Own Internet Distribution (YOID) [21], and others [60].

Several techniques have been proposed to distribute data efficiently using application-level multicast. Overcast [26] distributes content by constructing a bandwidth-optimized overlay tree among dedicated infrastructure nodes. SplitStream [13] distributes content via a peer-to-peer overlay that disseminates content along branches of trees constructed on top of a peer-to-peer substrate. Bullet [29] and Bullet′ [28] also use a randomized overlay mesh to distribute data. Chainsaw [48] is a peer-to-peer multicast based on an unstructured overlay mesh in which peers explicitly request packets from neighbors. This mechanism ensures that peers are able to receive all packets and avoid receiving duplicate packets. ChunkySpread [55] is a hybrid that uses both structured and unstructured overlays to distribute content. Antfarm differs from streaming multicast systems in that it aims to maximize aggregate system bandwidth for multiple concurrent batch downloads.

Another set of work proposes augmenting BitTorrent-like protocols to accommodate streaming video in a peer-to-peer setting. BASS [17] exemplifies this approach by adding peer-to-peer interactions to a client-server model where peers stream video from the server while trading blocks with other peers to alleviate load on the server in the future. Antfarm also incorporates a peer-to-peer protocol to alleviate load, but manages the interactions via the coordinator to achieve high throughput for multiple swarms. Siddhartha et al. [4] propose a BitTorrent-like protocol with small neighborhoods of topographically close peers for exchanging blocks, using heuristics to handle swarms of heterogeneous link capacities.

Finally, many streaming and multicast architectures use coding to increase content delivery reliability [2, 6, 24, 38]. Integrating coding techniques into Antfarm could further improve performance.

6 CONCLUSIONS

In this paper we introduced Antfarm, a peer-to-peer content distribution system for the batch dissemination of large files. Antfarm explores a novel space in the design of swarming protocols; whereas past systems avoid all vestiges of centralization for both technical and legal reasons and suffered from lack of coordination across swarms, Antfarm examines how modest planning by a centralized coordinator can help a set of competing swarms achieve high performance.

The key to Antfarm's performance is its restatement of the download management task as an optimization problem. The hill-climbing algorithm we propose effectively leverages available bandwidth, accommodates desired minimum bandwidth limits, avoids starvation, and enforces desired swarm priorities. The wire-level protocol enables performance information to be extracted from the network, enabling a practical deployment that reacts to changing network and swarm conditions. Even though the approach embodies a logically centralized coordinator, the computational requirements of the coordinator are modest, the bandwidth requirement is feasibly small, and the coordinator carries out an embarrassingly parallel task that is easy to replicate across datacenters. PlanetLab deployments and simulations indicate that the system is practical, scalable, and capable of achieving significantly higher bandwidth than previous approaches.

Acknowledgments: We would like to thank Hakim Weatherspoon for his help refining the protocol, Aaron Lenfestey for implementing the least-squares algorithm that generates response curves, Ymir Vigfusson for his help formalizing swarm metrics, and Steve Gribble for his input in early discussions of this work. This work was supported in part by NSF-CAREER 0546568.

References

[1] Bittorrent. http://bittorrent.com.

[2] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network Information Flow. *IEEE Transactions on Information Theory*, 46(4), July 2000.

[3] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling File Servers Via Cooperative Caching. *Symposium on Networked System Design and Implementation*, Boston, MA, May 2005.

[4] S. Annapureddy, S. Guha, C. Gkantsidis, D. Gunawardena, and P. Rodriguez. Is High-quality Vod Feasible Using P2p Swarming? *International World Wide Web Conference*, Banff, Canada, May 2007.

[5] A. C. Bavier, M. Bowman, B. N. Chun, D. E. Culler, S. Karlin, S. Muir, L. L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating Systems Support For Planetary-scale Network Services. *Symposium on Networked System Design and Implementation*, San Francisco, CA, March 2004.

[6] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A Digital Fountain Approach To Reliable Distribution Of Bulk Data. *SIGCOMM Conference*, Vancouver, Canada, August 1998.

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A Hierarchical Internet Object Cache. *USENIX Annual Technical Conference*, San Diego, CA, January 1996.

[8] B. Cohen. Incentives Build Robustness In Bittorrent. *Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, CA, May 2003.

[9] B.-G. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. An Anycast Service For Large Content Distribution. *International Workshop on Peer-to-Peer Systems*, Santa Barbara, CA, February 2006.

[10] J. Camp, M. Sirbu, and J. D. Tygar. Token And Notational Money In Electronic Commerce. *USENIX Workshop on Electronic Commerce*, New York, NY, July 1995.

[11] M. Calore. Zudeo Announces Deal With Bbc. *Wired Blog Network*, December 19 2006.

[12] M. Castro, P. Druschel, A. J. Ganesh, A. I. T. Rowstron, and D. S. Wallach. Secure Routing For Structured Peer-to-peer Overlay Networks. *Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[13] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. I. T. Rowstron, and A. Singh. Splitstream: High-bandwidth Multicast In Cooperative Environments. *Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[14] Y.-h. Chu, S. G. Rao, S. Seshan, and H. Zhang. A Case For End System Multicast. *IEEE Journal on Selected Areas in Communications*, 20(8), October 2002.

[15] D. R. Choffnes and F. E. Bustamante. Taming The Torrent: A Practical Approach To Reducing Cross-isp Traffic In Peer-to-peer Systems. *SIGCOMM Conference*, Seattle, WA, August 2008.

[16] L. P. Cox and B. D. Noble. Samsara: Honor Among Thieves In Peer-to-peer Storage. *Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[17] C. Dana, D. Li, D. Harrison, and C.-N. Chuah. Bass: Bittorrent Assisted Streaming System For Video-on-demand. *IEEE Workshop on Multimedia Signal Processing*, 2005.

[18] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment Issues For The ip Multicast Service And Architecture. *IEEE Network*, 14(1), January 2000.

[19] F. Dabek, R. Cox, M. F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. *SIGCOMM Conference*, Portland, OR, August 2004.

[20] S. E. Deering. Multicast Routing In Internetworks And Extended Lans. *SIGCOMM Conference*, Stanford, CA, August 1988.

[21] P. Francis, Y. Pryadkin, P. Radoslavov, R. Govindan, and B. Lindell. Yoid: Your Own Internet Distribution. *http://www.isi.edu/div7/yoid*, March 2001.

[22] Y. Fu, J. S. Chase, B. N. Chun, S. Schwab, and A. Vahdat. Sharp: An Architecture For Secure Resource Peering. *Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[23] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing Content Publication With Coral. *Symposium on Networked System Design and Implementation*, San Francisco, CA, March 2004.

[24] C. Gkantsidis and P. Rodriguez. Network Coding For Large Scale Content Distribution. *IEEE International Conference on Computer Communications*, Miami, FL, March 2005.

[25] S. Gadde, J. S. Chase, and M. Rabinovich. Web Caching And Content Distribution: A View From The Interior. *Computer Communications*, 24(2), May 2001.

[26] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. O'Toole, Jr. Overcast: Reliable Multicasting With An Overlay Network. *Symposium on Operating System Design and Implementation*, San Diego, CA, October 2000.

[27] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi. Web Caching With Consistent Hashing. *International World Wide Web Conference*, 1999.

[28] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining High-bandwidth Under Dynamic Network Conditions. *USENIX Annual Technical Conference*, Anaheim, CA, April 2005.

[29] D. Kostic, A. Rodriguez, J. R. Albrecht, and A. Vahdat. Bullet: High Bandwidth Data Dissemination Using An Overlay Mesh. *Symposium on Operating Systems Principles*, Bolton Landing, NY, October 2003.

[30] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet Service Providers Fear Peer-assisted Content Distribution? *Internet Measurement Conference*, Berkeley, CA, October 2005.

[31] D. R. Karger, E. Lehman, F. T. Leighton, R. Panigrahy, M. S. Levine, and D. Lewin. Consistent Hashing And Random Trees: Distributed Caching Protocols For Relieving Hot Spots On The World Wide Web. *ACM Symposium on Theory of Computing*, El Paso, TX, May 1997.

[32] I. A. Kash, E. J. Friedman, and J. Y. Halpern. Optimizing Scrip Systems: Efficiency, Crashes, Hoarders, And Altruists. *ACM Conference on Electronic Commerce*, San Diego, CA, June 2007.

[33] J. Ledlie, P. Gardner, and M. Seltzer. Network Coordinates In The Wild. *Symposium on Networked System Design and Implementation*, Cambridge, MA, April 2007.

[34] N. Laoutaris, P. Rodriguez, and L. Massoulie. Echos: Edge Capacity Hosting Overlays Of Nano Data Centers. *ACM SIGCOMM: Computer Communication Review*, 38, January 2008.

[35] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding In Bittorrent Is Cheap. *Workshop on Hot Topics in Networks*, Irvine, CA, November 2006.

[36] H. C. Li, A. Clement, E. L. Wong, J. Napper, I. Roy, L. Alvisi, and M. Dahlin. Bar Gossip. *Symposium on Operating System Design and Implementation*, Seattle, WA, November 2006.

[37] M. Manasse. The Millicent Protocol For Electronic Commerce. *USENIX Workshop on Electronic Commerce*, New York, NY, August 1995.

[38] P. Maymounkov and D. Mazières. Rateless Codes And Big Downloads. *International Workshop on Peer-to-Peer Systems*, Springer Lecture Notes in Computer Science 2735, Berkeley, CA, February 2003.

[39] T.-W. Ngan, D. S. Wallach, and a. P. Druschel. Enforcing Fair Sharing Of Peer-to-peer Resources. *International Workshop on Peer-to-Peer Systems*, Springer Lecture Notes in Computer Science 2735, Berkeley, CA, February 2003.

[40] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An Analysis Of Approximations For Maximizing Submodular Set Functions. *Mathematical Programming*, 14(1), December 1978.

[41] T. S. E. Ng and H. Zhang. Towards Global Network Positioning. *ACM SIGCOMM Internet Measurement Workshop*, San Francisco, CA, November 2001.

[42] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting Similarity For Multi-source Downloads Using File Handprints. *Symposium on Networked System Design and Implementation*, Cambridge, MA, April 2007.

[43] J. Pouwelse, P. Garbacki, D. Epema, and H. Sips. The Bittorrent P2p File-sharing System: Measurements And Analysis. *International Workshop on Peer-to-Peer Systems*, Springer Lecture Notes in Computer Science 3640, Ithaca, NY, February 2005.

[44] K. Park and V. S. Pai. Scale And Performance In Coblitz Large-file Distribution Service. *Symposium on Networked System Design and Implementation*, San Jose, CA, May 2006.

[45] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do Incentives Build Robustness In Bittorrent? *Symposium on Networked System Design and Implementation*, Cambridge, MA, April 2007.

[46] M. Piatek, T. Isdal, A. Krishnamurthy, and T. Anderson. One Hop Reputations For Peer To Peer File Sharing Workloads. *Symposium on Networked System Design and Implementation*, 2008.

[47] T. Poutanen, H. Hinton, and M. Stumm. Netcents: A Lightweight Protocol For Secure Micropayments. *USENIX Workshop on Electronic Commerce*, Boston, MA, August 1998.

[48] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. E. Mohr. Chainsaw: Eliminating Trees From Overlay Multicast. *International Workshop on Peer-to-Peer Systems*, Springer Lecture Notes in Computer Science 3640, Ithaca, NY, February 2005.

[49] D. Qiu and R. Srikant. Modeling And Performance Analysis Of Bittorrent-like Peer-to-peer Networks. *SIGCOMM Conference*, Portland, OR, August 2004.

[50] S. C. Rhea, D. Geels, T. Roscoe, and J. Kubiatowicz. Handling Churn In A Dht (awarded Best Paper!). *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[51] M. Sirivianos, J. H. Park, R. Chen, and X. Yang. Free-riding In Bittorrent Networks With The Large View Exploit. *International Workshop on Peer-to-Peer Systems*, 2007.

[52] M. Sirivianos, X. Yang, and S. Jarecki. Dandelion: Cooperative Content Distribution With Robust Incentives. *USENIX Annual Technical Conference*, 2007.

[53] S. Saroiu, P. K. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An Analysis Of Internet Content Delivery Systems. *Symposium on Operating System Design and Implementation*, Boston, MA, December 2002.

[54] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An Architecture For Internet Data Transfer. *Symposium on Networked System Design and Implementation*, San Jose, CA, May 2006.

[55] V. Venkataraman, P. Francis, and J. Calandrino. Chunkyspread: Multi-tree Unstructured Peer-to-peer. *International Workshop on Peer-to-Peer Systems*, Santa Barbara, CA, February 2006.

[56] V. Vishnumurthy, S. Chandrakumar, and E. G. Sirer. Karma: A Secure Economic Framework For P2p Resource Sharing. *Workshop on the Economics of Peer-to-Peer Systems*, Berkeley, CA, May 2003.

[57] A. Wolman, G. M. Voelker, N. Sharma, N. Cardwell, A. R. Karlin, and H. M. Levy. On The Scale And Performance Of Cooperative Web Proxy Caching. *Symposium on Operating Systems Principles*, Kiawah Island, SC, December 1999.

[58] L. Wang, K. Park, R. Pang, V. S. Pai, and L. L. Peterson. Reliability And Security In The Codeen Content Distribution Network. *USENIX Annual Technical Conference*, Boston, MA, June 2004.

[59] P. Wayner. *Digital Cash: Commerce On The Net*. Morgan Kaufmann, April 1996.

[60] Y. Zhu, W. Shu, and M.-Y. Wu. Approaches To Establishing Multicast Overlays. *IEEE International Conference on Services Computing*, 2, July 2005.

HashCache: Cache Storage for the Next Billion

Anirudh Badam[*], KyoungSoo Park[*,+], Vivek S. Pai[*] and Larry L. Peterson[*]
[*]*Department of Computer Science*
Princeton University
[+]*Department of Computer Science*
University of Pittsburgh

Abstract

We present HashCache, a configurable cache storage engine designed to meet the needs of cache storage in the developing world. With the advent of cheap commodity laptops geared for mass deployments, developing regions are poised to become major users of the Internet, and given the high cost of bandwidth in these parts of the world, they stand to gain significantly from network caching. However, current Web proxies are incapable of providing large storage capacities while using small resource footprints, a requirement for the integrated multi-purpose servers needed to effectively support developing-world deployments. HashCache presents a radical departure from the conventional wisdom in network cache design, and uses 6 to 20 times less memory than current techniques while still providing comparable or better performance. As such, HashCache can be deployed in configurations not attainable with current approaches, such as having multiple terabytes of external storage cache attached to low-powered machines. HashCache has been successfully deployed in two locations in Africa, and further deployments are in progress.

1 Introduction

Network caching has been used in a variety of contexts to reduce network latency and bandwidth consumption, ranging from FTP caching [31], Web caching [15, 4], redundant traffic elimination [20, 28, 29], and content distribution [1, 10, 26, 41]. All of these cases use local storage, typically disk-based, to reduce redundant data fetches over the network. Large enterprises and ISPs particularly benefit from network caches, since they can amortize their cost and management over larger user populations. Cache storage system design has been shaped by this class of users, leading to design decisions that favor first-world usage scenarios. For example, RAM consumption is proportional to disk size due to in-memory indexing of on-disk data, which was developed when disk storage was relatively more expensive than it is now. However, because disk size has been growing faster than RAM sizes, it is now much cheaper to buy terabytes of disk than a machine capable of indexing that much storage, since most low-end servers have lower memory limits.

This disk/RAM linkage makes existing cache storage systems problematic for developing world use, where it may be very desirable to have terabytes of cheap storage (available for less than US $100/TB) attached to cheap, low-power machines. However, if indexing a terabyte of storage requires 10 GB of RAM (typical for current proxy caches), then these deployments will require server-class machines, with their associated costs and infrastructure. Worse, this memory is dedicated for use by a single service, making it difficult to deploy consolidated multi-purpose servers. When low-cost laptops from the One Laptop Per Child project [22] or the Classmate from Intel [13] cost only US $200 each, spending thousands of dollars per server may exceed the cost of laptops for an entire school.

This situation is especially unfortunate, since bandwidth in developing regions is often more expensive, both in relative and absolute currency, than it is in the US and Europe. Africa, for example, has poor terrestrial connectivity, and often uses satellite connectivity, backhauled through Europe. One of our partners in Nigeria, for example, shares a 2 Mbps link, which costs $5000 per month. Even the recently-planned "Google Satellite," the O3b, is expected to drop the cost to only $500/Mbps per month by 2010 [21]. With efficient cache storage, one can reduce the network connectivity expenses.

The goal of this project is to develop network cache stores designed for developing-world usage. In this paper, we present HashCache, a configurable storage system that implements flexible indexing policies, all of which are dramatically more efficient than traditional cache designs. The most radical policy uses no main memory for indexing, and obtains performance comparable to traditional software solutions such as the Squid Web proxy cache. The highest performance policy performs equally with commercial cache appliances while using main-memory indexes that are only one-tenth their size. Between these policies are a range of distinct policies that trade memory consumption for performance suitable for a range of workloads in developing regions.

1.1 Rationale For a New Cache Store

HashCache is designed to serve the needs of developing-world environments, starting with classrooms but working toward backbone networks. In addition to good performance with low resource consumption, HashCache provides a number of additional benefits suitable for developing-world usage: (a) many HashCache policies can be tailored to use main memory in proportion to system activity, instead of cache size; (b) unlike commercial caching appliances, HashCache does not need to be the sole application running on the machine; (c) by simply choosing the appropriate indexing scheme, the same cache software can be configured as a low-resource end-user cache appropriate for small classrooms, as well as a high-performance backbone cache for higher levels of the network; (d) in its lowest-memory configurations, HashCache can run on laptop-class hardware attached to external multi-terabyte storage (via USB, for example), a scenario not even possible with existing designs; and (e) HashCache provides a flexible caching layer, allowing it to be used not only for Web proxies, but also for other cache-oriented storage systems.

A previous analysis of Web traffic in developing regions shows great potential for improving Web performance [8]. According to the study, kiosks in Ghana and Cambodia, with 10 to 15 users per day, have downloaded over 100 GB of data within a few months, involving 12 to 14 million URLs. The authors argue for the need for applications that can perform HTTP caching, chunk caching for large downloads and other forms of caching techniques to improve the Web performance. With the introduction of personal laptops into these areas, it is reasonable to expect even higher network traffic volumes.

Since HashCache can be shared by many applications and is not HTTP-specific, it avoids the problem of diminishing returns seen with large HTTP-only caches. HashCache can be used by both a Web proxy and a WAN accelerator, which stores pieces of network traffic to provide protocol-independent network compression. This combination allows the Web cache to store static Web content, and then use the WAN accelerator to reduce redundancy in dynamically-generated content, such as news sites, Wikipedia, or even locally-generated content, all of which may be marked uncacheable, but which tend to only change slowly over time. While modern Web pages may be large, they tend to be composed of many small objects, such as dozens of small embedded images. These objects, along with tiny fragments of cached network traffic from a WAN accelerator, put pressure on traditional caching approaches using in-memory indexing.

A Web proxy running on a terabyte-sized HashCache can provide a large HTTP store, allowing us to not only cache a wide range of traffic, but also speculatively preload content during off-peak hours. Furthermore, this kind of system can be driven from a typical OLPC-class laptop, with only 256MB of total RAM. One such laptop can act as a cache server for the rest of the laptops in the deployment, eliminating the need for separate server-class hardware. In comparison, using current Web proxies, these laptops could only index 30GB of disk space.

The rest of this paper is structured as follows. Section 2 explains the current state of the art in network storage design. Section 3 explains the problem, explores a range of HashCache policies, and analyzes them. Section 4 describes our implementation of policies and the HashCache Web proxy. Section 5 presents the performance evaluation of the HashCache Web Proxy and compares it with Squid and a modern high-performance system with optimized indexing mechanisms. Section 6 describes the related work, Section 7 describes our current deployments, and Section 8 concludes with our future work.

2 Current State-of-the-Art

While typical Web proxies implement a number of features, such as HTTP protocol handling, connection management, DNS and in-memory object caching, their performance is generally dominated by their filesystem organization. As such, we focus on the filesystem component because it determines the overall performance of a proxy in terms of the peak request rate and object cacheability. With regard to filesystems, the two main optimizations employed by proxy servers are hashing and indexing objects by their URLs, and using raw disk to bypass filesystem inefficiencies. We discuss both of these aspects below.

The Harvest cache [4] introduced the design of storing objects by a hash of their URLs, and keeping an in-memory index of objects stored on disk. Typically, two levels of subdirectories were created, with the fan-out of each level configurable. The high-order bits of the hash were used to select the appropriate directories, and the file was ultimately named by the hash value. This approach not only provided a simple file organization, but it also allowed most queries for the presence of objects to be served from memory, instead of requiring disk access. The older CERN [15] proxy, by contrast, stored objects by creating directories that matched the components of the URL. By hashing the URL, Harvest was able to con-

System	Naming	Storage Management	Memory Management
CERN	URL	Regular Filesystem	Filesystem Data Structures
Harvest	Hash	Regular Filesystem	LRU, Filesystem Data Structures
Squid	Hash	Regular Filesystem	LRU & others
Commercial	Hash	Log	LRU

Table 1: System Entities for Web Caches

trol both the depth and fan-out of the directories used to store objects. The CERN proxy, Harvest, and its descendant, Squid, all used the filesystems provided by the operating system, simplifying the proxy and eliminating the need for controlling the on-disk layout.

The next step in the evolution of proxy design was using raw disk and custom filesystems to eliminate multiple levels of directory traversals and disk head seeks associated with them. The in-memory index now stored the location on disk where the object was stored, eliminating the need for multiple seeks to find the start of the object. [1]

The first block of the on-disk file typically includes extra metadata that is too big to be held in memory, such as the complete URL, full response headers, and location of subsequent parts of the object, if any, and is followed by the content fetched from the origin server. In order to fully utilize the disk writing throughput, those blocks are often maintained consecutively, using a technique similar to log-structured filesystem(LFS) [30]. Unlike LFS, which is expected to retain files until deleted by the user, cache filesystems can often perform disk cache replacement in LIFO order, even if other approaches are used for main memory cache replacement. Table 1 summarizes the object lookup and storage management of various proxy implementations that have been used to build Web caches.

The upper bound on the number of cacheable objects per proxy is a function of available disk cache and physical memory size. Attempting to use more memory than the machine's physical memory can be catastrophic for caches, since unpredictable page faults in the application can degrade performance to the point of unusability. When these applications run as a service at network access points, which is typically the case, all users then suffer extra latency when page faults occur.

The components of the in-memory index vary from system to system, but a representative configuration for a high-performance proxy is given in Table 2. Each entry has some object-specific information, such as its hash value and object size. It also has some disk-related information, such as the location on disk, which disk, and which generation of log, to avoid problems with log wrapping. The entries typically are stored in a chain per hash bin, and a doubly-linked LRU list across all index entries. Finally, to shorten hash bin traversals (and the associated TLB pressure), the number of hash bins is typically set to roughly the number of entries.

[1]This information was previously available on the iMimic Networking Web site and the Volera Cache Web site, but both have disappeared. No citable references appear to exist

Entity	Memory per Object (Bytes)
Hash	4 - 20
LFS Offset	4
Size in Blocks	2
Log Generation	1
Disk Number	1
Bin Pointers	4
Chaining Pointers	8
LRU List Pointers	8
Total	32 - 48

Table 2: High Performance Cache - Memory Usage

Using these fields and their sizes, the total consumption per index entry can be as low as 32 bytes per object, but given that the average Web object is roughly 8KB (where a page may have tens of objects), even 32 bytes per object represents an in-memory index storage that is 1/256 the size of the on-disk storage. With a more realistic index structure, which can include a larger hash value, expiration time, and other fields, the index entry can be well over 80 bytes (as in the case of Squid), causing the in-memory index to exceed 1% of the on-disk storage size. With a single 1TB drive, the in-memory index alone would be over 10GB. Increasing performance by using multiple disks would then require tens of gigabytes of RAM.

Reducing the RAM needed for indexing is desirable for several scenarios. Since the growth in disk capacities has been exceeding the growth of RAM capacity for some time, this trend will lead to systems where the disk cannot be fully indexed due to a lack of RAM. Dedicated RAM also effectively limits the degree of multiprogramming of the system, so as processors get faster relative to network speeds, one may wish to consolidate multiple functions on a single server. WAN accelerators, for example, cache network data [5, 29, 34], so having very large storage can reduce bandwidth consumption more than HTTP proxies alone. Similarly, even in HTTP proxies, RAM may be more useful as a hot object cache than as an index, as is the case in reverse proxies (server accelerators) and content distribution networks. One goal in designing HashCache is to determine how much index memory is really necessary.

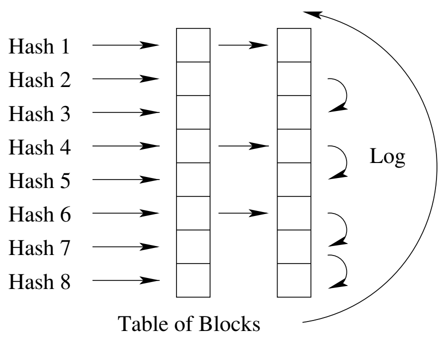


Figure 1: HashCache-Basic: objects with hash value i go to the ith bin for the first block of a file. Later blocks are in the circular log.

3 Design

In this section, we present the design of HashCache and show how performance can be scaled with available memory. We begin by showing how to eliminate the in-memory index while still obtaining reasonable performance, and then we show how selective use of minimal indexing can improve performance. A summary of policies is shown in Table 3.

3.1 Removing the In-Memory Index

We start by removing the in-memory index entirely, and incrementally introducing minimal metadata to systematically improve performance. To remove the in-memory index, we have to address the two functions the in-memory index serves: indicating the existence of an object and specifying its location on disk. Using filesystem directories to store objects by hash has its own performance problems, so we seek an alternative solution – treating the disk as a simple hashtable.

HashCache-Basic, the simplest design option in the HashCache family, treats part of the disk as a fixed-size, non-chained hash table, with one object stored in each bin. This portion is called the Disk Table. It hashes the object name (a URL in the case of a Web cache) and then calculates the hash value modulo the number of bins to determine the location of the corresponding file on disk. To avoid false positives from hash collisions, each stored object contains metadata, including the original object name, which is compared with the requested object name to confirm an actual match. New objects for a bin are simply written over any previous object.

Since objects may be larger than the fixed-size bins in the Disk Table, we introduce a circular log that contains the remaining portion of large objects. The object metadata stored in each Disk Table bin also includes the location in the log, the object size, and the log generation number, and is illustrated in Figure 1.

The performance impact of these decisions is as follows: in comparison to high-performance caches, HashCache-Basic will have an increase in hash collisions (reducing cache hit rates), and will require a disk access on every request, even cache misses. Storing objects will require one seek per object (due to the hash randomizing the location), and possibly an additional write to the circular log.

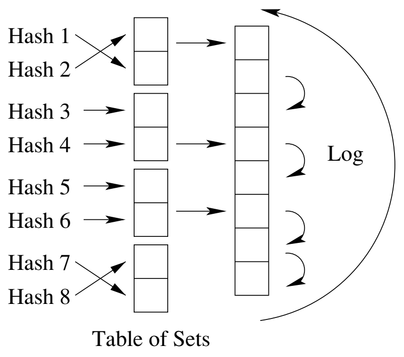


Figure 2: HashCache-Set: Objects with hash value i search through the $\frac{i}{N}^{th}$ set for the first block of a file. Later blocks are in the circular log. Some arrows are shown crossed to illustrate that objects that map on to a set can be placed anywhere in the set.

3.2 Collision Control Mechanism

While in-memory indexes can use hash chaining to eliminate the problem of hash values mapped to the same bin, doing so for an on-disk index would require many random disk seeks to walk a hash bin, so we devise a simpler and more efficient approach while retaining most of the benefits.

In HashCache-Set, we expand the Disk Table to become an N-way set-associative hash table, where each bin can store N elements. Each element still contains metadata with the full object name, size, and location in the circular log of any remaining part of the object. Since these locations are contiguous on disk, and since short reads have much lower latency than seeks, reading all of the members of the set takes only marginally more time than reading just one element. This approach is shown in Figure 2, and reduces the impact of popular objects mapping to the same hash bin, while only slightly increasing the time to access an object.

While HashCache-Set eliminates problems stemming from collisions in the hash bins, it still has several problems: it requires disk access for cache misses, and lacks an efficient mechanism for cache replacement within the set. Implementing something like LRU within the set using the on-disk mechanism would require a potential disk write on every cache hit, reducing performance.

3.3 Avoiding Seeks for Cache Misses

Requiring a disk seek to determine a cache miss is a major issue for workloads with low cache hit rates, since an

Policy	Bits Per Object	RAM GB per Disk TB	Read Seeks	Write Seeks	Miss Seeks	Comments
Squid	576-832	9 - 13	~ 6	~ 6	0	Harvest descendant
Commercial	256-544	4 - 8.5	< 1	~ 0	0	custom filesystem
HC-Basic	0	0	1	1	1	high collision rate
HC-Set	0	0	1	1	1	adds N-way sets to reduce collisions
HC-SetMem	11	0.17	1	1	0	small in-mem hash eliminates miss seeks
HC-SetMemLRU	< 11	< 0.17	1	1	< 1	only some sets kept in memory
HC-Log	47	0.73	1	~ 0	0	writes to log, log position added to entry
HC-LogLRU	15-47	0.23 - 0.67	$1 + \epsilon$	~ 0	0	log position for only some entries in set
HC-LogLRU + Prefetch	23-55	0.36 - 0.86	< 1	~ 0	0	reads related objects together
HC-Log + Prefetch	55	0.86	< 1	~ 0	0	reads related objects together

Table 3: Summary of HashCache policies, with Squid and commercial entries included for comparison. Main memory consumption values assume an average object size of 8KB. Squid memory data appears in http://www.comfsm.fm/computing/squid/FAQ-8.html

index-less cache would spend most of its disk time confirming cache misses. This behavior would add extra latency for the end-user, and provide no benefit. To address the problem of requiring seeks for cache misses, we introduce the first HashCache policy with any in-memory index, but employ several optimizations to keep the index much smaller than traditional approaches.

As a starting point, we consider storing in main memory an H-bit hash values for each cached object. These hash values can be stored in a two-dimensional array which corresponds to the Disk Table, with one row for each bin, and N columns corresponding to the N-way associativity. An LRU cache replacement policy would need forward and reverse pointers per object to maintain the LRU list, bringing the per-object RAM cost to (H + 64) bits assuming 32-bit pointers. However, we can reduce this storage as follows.

First, we note that all the entries in an N-entry set share the same modulo hash value (%S) where S is the number of sets in the Disk Table. We can drop the lowest log(S) bits from each hash value with no loss, reducing the hash storage to only H - log(S) bits per object.

Secondly, we note that cache replacement policies only need to be implemented within the N-entry set, so LRU can be implemented by simply ranking the entries from 0 to N-1, thereby using only log(N) bits per entry.

We can further choose to keep in-memory indexes for only some sets, not all sets, so we can restrict the number of in-memory entries based on request rate, rather than cache size. This approach keeps sets in an LRU fashion, and fetches the in-memory index for a set from disk on demand. By keeping only partial sets, we need to also keep a bin number with each set, LRU pointers per set, and a hash table to find a given set in memory.

Deciding when to use a complete two-dimensional array versus partial sets with bin numbers and LRU pointers depends on the size of the hash value and the set associativity. Assuming 8-way associativity and the 8 most significant hash bits per object, the break-even point is around 50% – once more than half the sets will be stored in memory, it is cheaper to remove the LRU pointers and bin number, and just keep all of the sets. A discussion of how to select values for these parameters is provided in Section 4.

If the full array is kept in memory, we call it HashCache-SetMem, and if only a subset are kept in memory, we call it HashCache-SetMemLRU. With a low hash collision rate, HashCache-SetMem can determine most cache misses without accessing disk, whereas HashCache-SetMemLRU, with its tunable memory consumption, will need disk accesses for some fraction of the misses. However, once a set is in memory, performing intra-set cache replacement decisions requires no disk access for policy maintenance. Writing objects to disk will still require disk access.

3.4 Optimizing Cache Writes

With the previous optimizations, cache hits require one seek for small files, and cache misses require no seeks (excluding false positives from hash collisions) if the associated set's metadata is in memory. Cache writes still require seeks, since object locations are dictated by their hash values, leaving HashCache at a performance disadvantage to high-performance caches that can write all content to a circular log. This performance problem is not an issue for caches with low request rates, but will become a problem for higher request rate workloads.

To address this problem, we introduce a new policy, HashCache-Log, that eliminates the Disk Table and treats the disk as a log, similar to the high-performance caches. For some or all objects, we store an additional offset (32 or 64 bits) specifying the location on disk. We retain the N-way set associativity and per-set LRU replacement because they eliminate disk seeks for cache misses with compact implementation. While this approach significantly increases memory consumption, it can also yield a large performance advantage, so this tradeoff is useful in many situations. However, even when adding the log location, the in-memory index is still much smaller than traditional caches. For example, for 8-way set associativity, per-set LRU requires 3 bits per entry, and 8 bits per entry can minimize hash collisions within the set. Adding a 32-bit log position increases the per-entry size from 11 bits to 43 bits, but virtually eliminates the impact of write traffic, since all writes can now be accumulated and written in one disk seek. Additionally, we need a few bits (assume 4) to record the log generation number, driving the total to 47 bits. Even at 47 bits per entry, HashCache-Log still uses indexes that are a factor of 6-12 times smaller than current high-performance proxies.

We can reduce this overhead even further if we exploit Web object popularity, where half of the objects are rarely, if ever, re-referenced [8]. In this case, we can drop half of the log positions from the in-memory index, and just store them on disk, reducing the average per-entry size to only 31 bits, for a small loss in performance. HashCache-LogLRU allows the number of log position entries per set to be configured, typically using $\frac{N}{2}$ log positions per N-object set. The remaining log offsets in the set are stored on the disk as a small contiguous file. Keeping this file and the in-memory index in sync requires a few writes reducing the performance by a small amount. The in-memory index size, in this case, is 9-20 times smaller than traditional high-performance systems.

3.5 Prefetching Cache Reads

With all of the previous optimizations, caching storage can require as little as 1 seek per object read for small objects, with no penalty for cache misses, and virtually no cost for cache writes that are batched together and written to the end of the circular log. However, even this performance can be further improved, by noting that prefetching multiple objects per read can amortize the read cost per object.

Correlated access can arise in situations like Web pages, where multiple small objects may be embedded in the HTML of a page, resulting in many objects being accessed together during a small time period. Grouping these objects together on disk would reduce disk seeks for reading and writing. The remaining blocks for these pages can all be coalesced together in the log and written together so that reading them can be faster, ideally with one seek.

The only change necessary to support this policy is to keep a content length (in blocks) for all of the related content written at the same time, so that it can be read together in one seek. When multiple related objects are read together, the system will perform reads at less than one seek per read on average. This approach can be applied to many of the previously described HashCache policies, and only requires that the application using HashCache provide some information about which objects are related. Assuming prefetch lengths of no more than 256 blocks, this policy only requires 8 bits per index entry being read. In the case of HashCache-LogLRU, only the entries with in-memory log position information need the additional length information. Otherwise, this length can also be stored on disk. As a result, adding this prefetching to HashCache-LogLRU only increases the in-memory index size to 35 bits per object, assuming half the entries of each set contain a log position and prefetch length.

For the rest of this paper, we assume all the policies to have this optimization except HashCache-LogN which is the HashCache-Log policy without any prefetching.

3.6 Expected Throughput

To understand the throughput implications of the various HashCache schemes, we analyze their expected performance under various conditions using the parameters shown in Table 4.

The maximum request rate(rr) is a function of the disk seek rate, the hit rate, the miss rate, and the write rate. The write rate is required because not all objects that are fetched due to cache misses are cacheable. Table 4 presents throughputs for each system as a function of these parameters. The cache hit rate(chr) is simply a number between 0 and 1, as is the cacheability rate (cbr). Since the miss rate is (1 - chr), the write rate can be represented as (1 - chr) $\cdot$ cbr. The peak disk seek rate(t) is a measured quantity that is hardware-dependent, and the average number of related objects(rel) is always a positive number. Due to space constraints, we omit the derivations for these calculations. These throughputs are

Policy	Throughput
HC-Basic	$rr = \frac{t}{1+\frac{1}{rel}+(1-chr)\cdot cbr}$
HC-Set	$rr = \frac{t}{1+\frac{1}{rel}+(1-chr)\cdot cbr}$
HC-SetMem	$rr = \frac{t}{chr\cdot\left(1+\frac{1}{rel}\right)+(1-chr)\cdot cbr}$
HC-LogN	$rr = \frac{t}{2\cdot chr+(1-chr)\cdot cbr}$
HC-LogLRU	$rr = \frac{t\cdot rel}{2\cdot chr+(1-chr)\cdot cbr}$
HC-Log	$rr = \frac{t\cdot rel}{2\cdot chr+(1-chr)\cdot cbr}$
Commercial	$rr = \frac{t\cdot rel}{2\cdot chr+(1-chr)\cdot cbr}$

Table 4: Throughputs for techniques, rr = peak request rate, chr = cache hit rate, cbr = cacheability rate, rel = average number of related objects, t = peak disk seek rate – all calculations include read prefetching, so the results for Log and Grouped are the same. To exclude the effects of read prefetching, simply set rel to one.

conservative estimates because we do not take into account the in-memory hot object cache, where some portion of the main memory is used as a cache for frequently used objects, which can further improve throughput.

4 HashCache Implementation

We implement a common HashCache filesystem I/O layer so that we can easily use the same interface with different applications. We expose this interface via POSIX-like calls, such as open(), read(), write(), close(), seek(), etc., to operate on files being cached. Rather than operate directly on raw disk, HashCache uses a large file in the standard Linux ext2/ext3 filesystem, which does not require root privilege. Creating this zero-filled large file on a fresh ext2/ext3 filesystem typically creates a mostly contiguous on-disk layout. It creates large files on each physical disk and multiplexes them for performance. The HashCache filesystem is used by the HashCache Web proxy cache as well as other applications we are developing.

4.1 External Indexing Interface

HashCache provides a simple indexing interface to support other applications. Given a key as input, the interface returns a data structure containing the file descriptors for the Disk Table file and the contiguous log file (if required), the location of the requested content, and metadata such as the length of the contiguous blocks belonging to the item, etc. We implement the interface for each indexing policy we have described in the previous section. Using the data returned from the interface one can utilize the POSIX calls to handle data transfers to and from the disk. Calls to the interface can block if disk access is needed, but multiple calls can be in flight at the same time. The interface consists of roughly 600 lines of code, compared to 21000 lines for the HashCache Web Proxy.

4.2 HashCache Proxy

The HashCache Web Proxy is implemented as an event-driven main process with cooperating helper processes/threads handling all blocking operations, such as DNS lookups and disk I/Os, similar to the design of Flash [25]. When the main event loop receives a URL request from a client, it searches the in-memory hot-object cache to see if the requested content is already in memory. In case of a cache miss, it looks up the URL using one of the HashCache indexing policies. Disk I/O helper processes use the HashCache filesystem I/O interface to read the object blocks into memory or to write the fetched object to disk. To minimize inter-process communication (IPC) between the main process and the helpers, only beacons are exchanged on IPC channels and the actual data transfer is done via shared memory.

4.3 Flexible Memory Management

HTTP workloads will often have a small set of objects that are very popular, which can be cached in main memory to serve multiple requests, thus saving disk I/O. Generally, the larger the in-memory cache, the better the proxy's performance. HashCache proxies can be configured to use all the free memory on a system without unduly harming other applications. To achieve this goal, we implement the hot object cache via anonymous `mmap()` calls so that the operating system can evict pages as memory pressure dictates. Before the HashCache proxy uses the hot object cache, it checks the memory residency of the page via the `mincore()` system call, and simply treats any missing page as a miss in the hot object cache. The hot object cache is managed as an LRU list and unwanted objects or pages no longer in main memory can be unmapped. This approach allows the HashCache proxy to use the entire main memory when no other applications need it, and to seamlessly reduce its memory consumption when there is memory pressure in the system.

In order to maximize the disk writing throughput, the HashCache proxy buffers recently-downloaded objects so that many objects can be written in one batch (often to a circular log). These dirty objects can be served from memory while waiting to be written to disk. This dirty object cache reduces redundant downloads during flash crowds because many popular HTTP objects are usually requested by multiple clients.

HashCache also provides for grouping related objects to disk so that they can be read together later, providing the benefits of prefetching. The HashCache proxy uses this feature to amortize disk seeks over multiple objects, thereby obtaining higher read performance. One commercial system parses HTML to explicitly find embedded objects [7], but we use a simpler approach – simply grouping downloads by the same client that occur within a small time window and that have the same HTTP Referrer field. We have found that this approach works well in practice, with much less implementation complexity.

4.4 Parameter Selection

For the implementation, we choose some design parameters such as the block size, the set size, and the hash size. Choosing the block size is a tradeoff between space usage and the number of seeks necessary to read small objects. In Table 5, we show an analysis of object sizes from a live, widely-used Web cache called CoDeeN [41]. We see that nearly 75% of objects are less than 8KB, while 87.2% are less than 16KB. Choosing an 8KB block would yield better disk usage, but would require multiple seeks for 25% of all objects. Choosing the larger block size wastes some space, but may increase performance.

Since the choice of block size influences the set size,

Size (KB)	% of objects < size
8	74.8
16	87.2
32	93.8
64	97.1
128	98.8
256	99.5

Table 5: CDF of Web object sizes

we make the decisions based on the performance of current disks. Table 6 shows the average number of seeks per second of three recent SATA disks (18, 60 and 150 GB each). We notice the sharp degradation beyond 64KB, so we use that as the set size. Since 64KB can hold 4 blocks of 16KB each or 8 blocks of 8KB each, we opt for an 8KB block size to achieve 8-way set associativity. With 8 objects per set, we choose to keep 8 bits of hash value per object for the in-memory indexes, to reduce the chance of collisions. This kind of an analysis can be automatically performed during initial system configuration, and are the only parameters needed once the specific HashCache policy is chosen.

5 Performance Evaluation

In this section, we present experimental results that compare the performance of different indexing mechanisms presented in Section 3. Furthermore, we present a comparison between the HashCache Web Proxy Cache, Squid, and a high-performance commercial proxy called Tiger, using various configurations. Tiger implements the best practices outlined in Section 2 and is currently used in commercial service [6]. We also present the impact of the optimizations that we included in the HashCache Web Proxy Cache. For fair comparison, we use the same basic code base for all the HashCache variants, with differences only in the indexing mechanisms.

5.1 Workload

To evaluate these systems, we use the Web Polygraph [37] benchmarking tool, the *de facto* industry standard for testing the performance of HTTP intermediaries such as content filters and caching proxies. We use the Polymix [38] environment models, which models many key Web traffic characteristics, including: multiple content types, diurnal load spikes, URLs with transient popularity, a global URL set, flash crowd behavior, an unlimited number of objects, DNS names in URLs, object life-cycles (expiration and last-modification times), persistent connections, network packet loss, reply size variations, object popularity (recurrence), request rates and inter-arrival times, embedded objects and browser behavior, and cache validation (If-Modified-Since requests and reloads).

Read Size (KB)	Seeks/sec	Latency/seek (ms)
1	78	12.5
4	76	12.9
8	76	13.1
16	74	13.3
32	72	13.7
64	70	14.1
128	53	19.2

Table 6: Disk performance statistics

We use the latest standard workload, Polymix-4 [38], which was used at the Fourth Cache-off event [39] to benchmark many proxies. The Polygraph test harness uses several machines for emulating HTTP clients and others to act as Web servers. This workload offers a cache hit ratio (CHR) of 60% and a byte hit ratio (BHR) of 40% meaning that at most 60% of the objects are cache hits while 40% of bytes are cache hits. The average download latency is 2.5 seconds (including RTT). A large number of objects are smaller than 8.5 KB. HTML pages contain 10 to 20 embedded (related) objects, with an average size of 5 to 10 KB. A small number (0.1 %) of large downloads (300 KB or more) have higher cache hit rates. These numbers are very similar to the characteristics of traffic in developing regions [8].

We test three environments, reflecting the kinds of caches we expect to deploy. These are the low-end systems that reflect the proxy powered by a laptop or similar system, large-disk systems where a larger school can purchase external storage to pre-load content, and high-performance systems for ISPs and network backbones.

5.2 Low-End System Experiments

Our first test server for the proxy is designed to mimic a low-memory laptop, such as the OLPC XO Laptop, or a shared low-powered machine like an OLPC XS server. Its configuration includes a 1.4 GHz CPU with 512 KB of L2 cache, 256 MB RAM, two 60GB 7200 RPM SATA drives, and the Fedora 8 Linux OS. This machine is far from the standard commercial Web cache appliance, and is likely to be a candidate machine for the developing world [23].

Our tests for this machine configuration run at 40-275 requests per second, per disk, using either one or two disks. Figure 3 shows the results for single disk performance of the Web proxy using HashCache-Basic (HC-B), HashCache-Set (HC-S), HashCache-SetMem (HC-SM), HashCache-Log without object prefetching (HC-LN), HashCache-Log with object prefetching (HC-L), Tiger and Squid. The HashCache tests use 60 GB caches. However, Tiger and Squid were unable to index this amount of storage and still run acceptably, so were limited to using 18 GB caches. This smaller cache is still sufficient to hold the working set of the test, so Tiger and

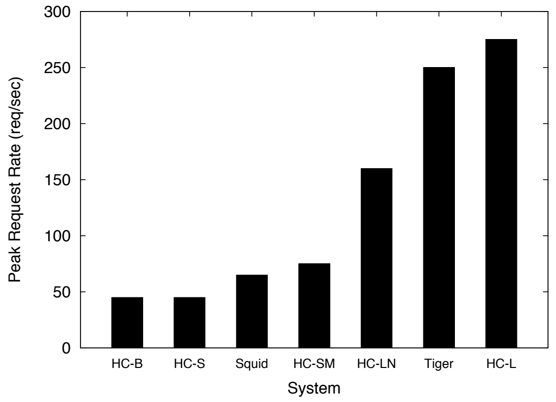


Figure 3: Peak Request Rates for Different policies for low end SATA disk.

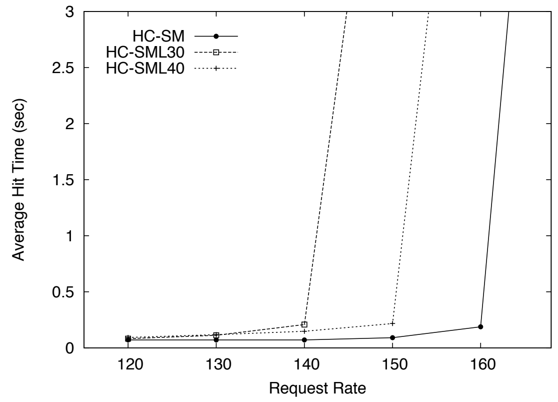


Figure 4: Peak Request Rates for Different SetMemLRU policies on low end SATA disks.

policy	SATA 7200	SCSI 10000	SCSI 15000
HC-Basic	40	50	85
HC-Set	40	50	85
HC-SetMem	66	85	140
HC-LogN	132	170	280
HC-LogLRU	264	340	560
HC-Log	264	340	560
Commercial	264	340	560

Table 7: Expected throughputs (reqs/sec) for policies for different disk speeds– all calculations include read prefetching

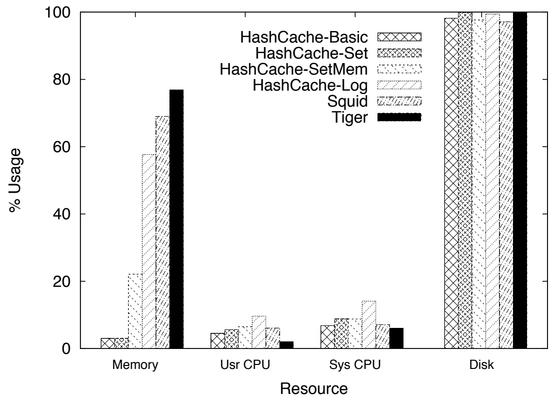


Figure 5: Resource Usage for Different Systems

Squid do not suffer in performance as a result. Table 7 gives the analytical lowerbounds for performance of each of these policies for this workload and the disk performance. The tests for HashCache-Basic and HashCache-Set achieve only 45 reqs/sec. The tests for HashCache-SetMem achieve 75 reqs/sec. Squid scales better than HashCache-Basic and HashCache-Set and achieves 60 reqs/sec. HashCache-Log (with prefetch), in comparison, achieves 275 reqs/sec. The Tiger proxy, with its optimized indexing mechanism, achieves 250 reqs/sec. This is less than HashCache-Log because Tiger's larger index size reduces the amount of hot object cache available, reducing its prefetching effectiveness.

Figure 4 shows the results from tests conducted on HashCache-SetMem and two configurations of HashCache-SetMemLRU using 2 disks. The performance of the HashCache-SetMem system scales to 160 reqs/sec, which is slightly more than double its performance with a single disk. The reason for this difference is that the second disk does not have the overhead of handling all access logging for the entire system. The two other graphs in the figure, labeled HC-SML30 and HC-SML40, are the 2 versions of HashCache-SetMemLRU where only 30% and 40% of all the set headers are cached in main memory. As mentioned earlier, the hash table and the LRU list overhead of HashCache-SetMemLRU is such that when 50% of set headers are cached, it takes about the same amount of memory when using HashCache-SetMem. These experiments serve to show that HashCache-SetMemLRU can provide further savings when working set sizes are small and one does not need all the set headers in main memory at all times to perform reasonably well.

These experiments also demonstrate HashCache's small systems footprint. Those measurements are shown in Figure 5 for the single-disk experiment. In all cases, the disk is the ultimate performance bottleneck, with nearly 100% utilization. The user and system CPU remain relatively low, with the higher system CPU levels tied to configurations with higher request rates. The most surprising metric, however, is Squid's high memory usage rate. Given that its storage size was only one-third that used by HashCache, it still exceeds HashCache's memory usage in HashCache's highest-performance configuration. In comparison, the lowest-performance HashCache configurations, which have performance comparable to Squid, barely register in terms of memory usage.

	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-Log	2200	116.98	77	1147	2508	56.91	41.06
Tiger	2300	121.40	98	1150	2512	56.49	41.40
Squid	400	21.38	63	1109	2509	57.25	41.22

Table 8: Performance on a high end system

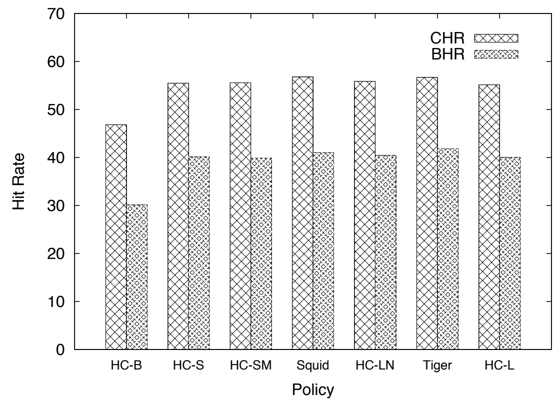


Figure 6: Low End Systems Hit Ratios

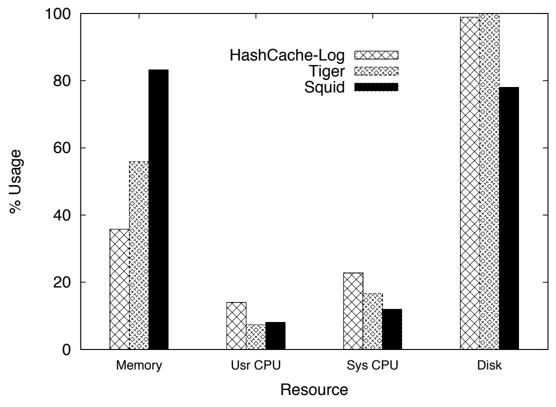


Figure 7: High End System Performance Statistics

Figure 6 shows the cache hit ratio (by object) and the byte hit ratios (bandwidth savings) for the HashCache policies at their peak request rate. Almost all configurations achieve the maximum offered hit ratios, with the exception of HashCache-Basic, which suffers from hash collision effects.

While the different policies offer different tradeoffs, one might observe that the performance jump between HashCache-SetMem and HashCache-Log is substantial. To bridge this gap one can use multiple small disks instead of one large disk to increase performance while still using the same amount of main memory. These experiments further demonstrate that for low-end machines, HashCache can not only utilize more disk storage than commercial cache designs, but can also achieve comparable performance while using less memory. The larger storage size should translate into greater network savings, and the low resource footprint ensures that the proxy machine need not be dedicated to just a single task. The HashCache-SetMem configuration can be used when one wants to index larger disks on a low-end machine with a relatively low traffic demand. The lowest-footprint configurations, which use no main-memory indexing, HashCache-Basic and HashCache-Set, would even be appropriate for caching in wireless routers or other embedded devices.

5.3 High-End System Experiments

For our high-end system experiments, we choose hardware that would be more appropriate in a datacenter. The processor is a dual-core 2GHz Xeon, with 2MB of L2 cache. The server has 3.5GB of main memory, and five 10K RPM Ultra2 SCSI disks, of 18GB each. These disks perform 90 to 95 random seeks/sec. Using our analytical models, we expect a performance of at least 320 reqs/sec/disk with HashCache-Log. On this machine we run HashCache-Log, Tiger and Squid. From the HashCache configurations, we chose only HashCache-Log because the ample main memory of this machine would dictate that it can be used for better performance rather than maximum cache size.

Figure 7 shows the resource utilization of the three systems at their peak request rates. HashCache-Log consumes just enough memory for hot object caching, write buffers and also the index, still leaving about 65% of the memory unused. At the maximum request rate, the workload becomes completely disk bound. Since the working set size is substantially larger than the main memory size, expanding the hot object cache size produces diminishing returns. Squid fails to reach 100% disk throughput simultaneously on all disks. Dynamic load imbalance among its disks causes one disk to be the system bottleneck, even though the other four disks are underutilized. The load imbalance prevents it from achieving higher request rates or higher average disk utilization.

The performance results from this test are shown in Table 8, and they confirm the expectations from the analytical models. HashCache-Log and Tiger perform comparably well at 2200-2300 reqs/sec, while Squid reaches only 400 reqs/sec. Even at these rates, HashCache-Log is purely disk-bound, while the CPU and memory consumption has ample room for growth. The per-disk performance of HashCache-Log of 440 reqs/sec/disk is in

1TB Configuration	Request Rate per sec	Throughput Mb/s	Hit Time msec	All Time msec	Miss Time msec	CHR %	BHR %
HashCache-SetMem	75	3.96	27	1142	2508	57.12	40.11
HashCache-Log	300	16.02	48	1139	2507	57.88	40.21
HashCache-LogLRU	300	16.07	68	1158	2510	57.15	40.08
2TB Configuration	**Request Rate per sec**	**Throughput Mb/s**	**Hit Time msec**	**All Time msec**	**Miss Time msec**	**CHR %**	**BHR %**
HashCache-SetMem	150	7.98	32	1149	2511	57.89	40.89
HashCache-Log	600	32.46	56	1163	2504	57.01	40.07
HashCache-LogLRU	600	31.78	82	1171	2507	57.67	40.82

Table 9: Performance on large disks

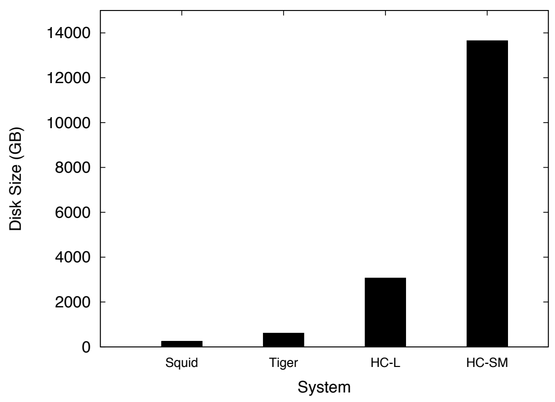


Figure 8: Sizes of disks that can be indexed by 2GB memory

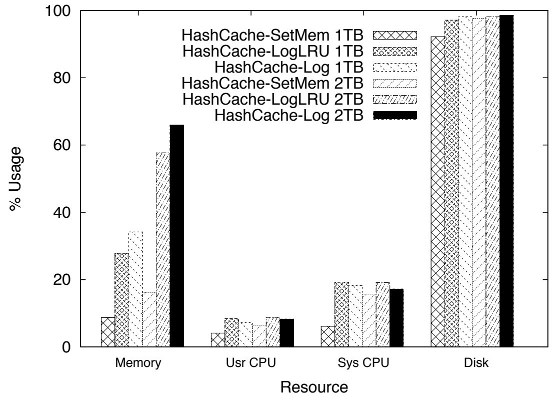


Figure 9: Large Disk System Performance Statistics

line with the best commercial showings – the highest-performing system at the Fourth Cacheoff achieved less than an average of 340 reqs/sec/disk on 10K RPM SCSI disks. The absolute best throughput that we find from the Fourth Cacheoff results is 625 reqs/sec/disk on two 15K RPM SCSI disks, and on the same speed disks HashCache-Log and Tiger both achieve 700 reqs/sec/disk, confirming the comparable performance.

These tests demonstrate that the same HashCache code base can provide good performance on low-memory machines while matching or exceeding the performance of high-end systems designed for cache appliances. Furthermore, this performance comes with a significant savings in memory, allowing room for larger storage or higher performance.

5.4 Large Disk Experiments

Our final set of experiments involves using HashCache configurations with large external storage systems. For this test, we use two 1 TB external hard drives attached to the server via USB. These drives perform 67-70 random seeks per second. Using our analytical models, we would expect a performance of 250 reqs/sec with HashCache-Log. In other respects, the server is configured comparably to our low-end machine experiment, but the memory is increased from 256MB to 2GB to accommodate some of the configurations that have larger index requirements, representative of low-end servers being deployed [24].

We compare the performance of HashCache-SetMem, HashCache-Log and HashCache-LogLRU with one or two external drives. Since the offered cache hit rate for the workload is 60%, we cache 6 out of the 8 log offsets in main memory for HashCache-LogLRU. For these experiments, the Disk Table is stored on a disk separate from the ones keeping the circular log. Also, since filling the 1TB hard drives at 300 reqs/second would take excessively long, we randomly place 50GB of data across each drive to simulate seek-limited behavior.

Unfortunately, even with 2GB of main memory, Tiger and Squid are unable to index these drives, so we were unable to test them in any meaningful way. Figure 8 shows the size of the largest disk that each of the systems can index with 2 GB of memory. In the figure, HC-SM and HC-L are HashCache-SetMem and HashCache-Log, respectively. The other HashCache configurations, Basic and Set have no practical limit on the amount of externally-attached storage.

The Polygraph results for these configurations are shown in Table 9, and the resource usage details are in Figure 9. With 2TB of external storage, both HashCache-Log and HashCache-LogLRU are able to perform 600 reqs/sec. In this configuration, HashCache-Log uses slightly more than 60% of the system's memory, while HashCache-LogLRU uses slightly less. The hit time for HashCache-LogLRU is a little higher than HashCache-Log because in some cases it requires 2 seeks (one for the position, and one for the content) in order to perform a read. The slightly higher cache hit rates exhibited on this test versus the high-end systems test are due the Polygraph environment – without filling the cache, it has a smaller set of objects to reference, yielding a higher offered hit ratio.

The 1TB test achieves half the performance of the 2TB test, but does so with correspondingly less memory utilization. The HashCache-SetMem configuration actually uses less than 10% of the 2GB overall in this scenario, suggesting that it could have run with our original server configuration of only 256MB.

While the performance results are reassuring, these experiments prove that HashCache can index disks that are much larger than conventional policies could handle. At the same time, HashCache performance meets or exceeds what other caches would produce on much smaller disks. This scenario is particularly important for the developing world, because one can use these inexpensive high-capacity drives to host large amounts of content, such as a Wikipedia mirror, WAN accelerator chunks, HTTP cache, and any other content that can be preloaded or shipped on DVDs later.

6 Related Work

Web caching in its various forms has been studied extensively in the research and commercial communities. As mentioned earlier, the Harvest cache [4] and CERN caches [17] were the early approaches. The Harvest design persisted, especially with its transformation into the widely-used Squid Web proxy [35]. Much research has been performed on Squid, typically aimed at reorganizing the filesystem layout to improve performance [16, 18], better caching algorithms [14], or better use of peer caches [11]. Given the goals of HashCache, efficiently operating with very little memory and large storage, we have avoided more complexity in cache replacement policies, since they typically use more memory to make the decisions. In the case of working sets that dramatically exceed physical memory, cache policies are also likely to have little real impact. Disk cache replacement policies also become less effective when storage sizes grow very large. We have also avoided Bloom-filter approaches [2] that would require periodic rebuilds, since scanning terabyte-sized disks can sap disk performance for long periods. Likewise, approaches that require examining multiple disjoint locations [19, 32] are also not appropriate for this environment, since any small gain in reducing conflict misses would be offset by large losses in checking multiple locations on each cache miss.

Some information has been published about commercial caches and workloads in the past, including the design considerations for high-speed environments [3], proxy cache performance in mixed environments [9], and workload studies of enterprise user populations [12]. While these approaches have clearly been successful in the developed world, many of the design techniques have not typically transitioned to the more price-sensitive portions of the design space. We believe that HashCache demonstrates that addressing problems specific to the developing world can also open interesting research opportunities that may apply to systems that are not as price-sensitive or resource-constrained.

In terms of performance optimizations, two previous systems have used some form of prefetching, including one commercial system [7], and one research project [33]. Based on published metrics, HashCache performs comparably to the commercial system, despite using a much similar approach to grouping objects, and despite using a standard filesystem for storage instead of raw disk access. Little scalability information is presented on the research system, since it was tested only using Apache mod_proxy at 8 requests per second. Otherwise, very little information is publically available regarding how high-performance caches typically operate from the extremely competitive commercial period for proxy caches, centered around the year 2000. In that year, the Third Cache-Off [40] had a record number of vendors participate, representing a variety of different caching approaches. In terms of performance, HashCache-Log compares favorably to all of them, even when normalized for hardware.

Web caches also get used in two other contexts: server accelerators and content distribution networks (CDNs) [1, 10, 26, 41]. Server accelerators, also known as reverse proxies, typically reside in front of a Web server and offload cacheable content, allowing the Web server to focus on dynamically-generated content. CDNs geographically distribute the caches reducing latency to the client and bandwidth consumption at the server. In these cases, the proxy typically has a very high hit rate, and is often configured to serve as much content from memory as possible. We believe that HashCache is also well-suited for this approach, because in the SetMemLRU configuration, only the index entries for popular content need to be kept in memory. By freeing the main memory from storing the entire index, the extra memory can be used to expand the size of the hot object cache.

Finally, in terms of context in developing world projects, HashCache is simply one piece of the infrastructure that can help these environments. Advances in wireless network technologies, such as WiMax [42] or rural WiFi [27, 36] will help make networking available to larger numbers of people, and as demand grows, we believe that the opportunities for caching increase. Given the low resource usage of HashCache and its suitability for operation on shared hardware, we believe it is well-suited to take advantage of networking advancements in these communities.

7 Deployments

HashCache is currently deployed at two different locations in Africa, at the Obafemi Awolowo University (OAU) in Nigeria and at the Kokrobitey Institute (KI) in Ghana. At OAU, it runs on their university server which has a 100 GB hard drive, 2 GB memory and a dual core Xeon processor. For Internet connection, they pay $5,000 per month for a 2 Mbps satellite link to an ISP in Europe and the link has a high variance ICMP ping time from Princeton ranging 500 to 1200 ms. We installed HashCache-Log on the machine but were asked to limit resource usage for HashCache to 50 GB disk space and no more than 300 MB of physical memory. The server is running other services such as a E-mail service and a firewall for the department and it is also used for general computation for the students. Due to privacy issues we were not able to analyze the logs from this deployment but the administrator has described the system as useful and also noticed the significant memory and CPU usage reduction when compared to Squid.

At KI, HashCache runs on a wireless router for a small department on a 2 Mbps LAN. The Internet connection is through a 256 Kbps sub-marine link to Europe and the link has a ping latency ranging from 200 to 500 ms. The router has a 30 GB disk and 128 MB of main memory and we were asked to use 20 GB of disk space and as little memory as possible. This prompted us to use the HashCache-Set policy as there are only 25 to 40 people using the router every day. Logging is disabled on this machine as well since we were asked not to consume network bandwidth on transferring the logs.

In both these deployments we have used HashCache policies to improve the Web performance while consuming minimum amount of resource. Other solutions like Squid would not have been able to meet these resource constraints while providing any reasonable service. People at both places told us that the idea of a faster Internet to popular Web sites seemed like a distant dream until we discussed the complete capabilities of HashCache. We are currently working with OLPC to deploy HashCache at more locations with the OLPC XS servers.

8 Conclusion and Future Work

In this paper we have presented HashCache, a high-performance configurable cache storage for the developing regions. HashCache provides a range of configurations that scale from using no memory for indexing to ones that require only one-tenth as much as current high-performance approaches. It provides this flexibility without sacrificing performance – its lowest-resource configuration has performance comparable to free software systems, while its high-end performance is comparable to the best commercial systems. These configurations allow memory consumption and performance to be tailored to application needs, and break the link between storage size and in-memory index size that has been commonly used in caching systems for the past decade. The benefits of HashCache's low resource consumption allow it to share hardware with other applications, share the filesystem, and to scale to storage sizes well beyond what present approaches provide.

On top of the HashCache storage layer, we have built a Web caching proxy, the HashCache Proxy, which can run using any of the HashCache configurations. Using industry-standard benchmarks and a range of hardware configurations, we have shown that HashCache performs competitively with existing systems across a range of workloads. This approach provides an economy of scale in HashCache deployments, allowing it to be powered from laptops, low-resource desktops, and even high-resource servers. In all cases, HashCache either performs competitively or outperforms other systems suited to that class of hardware.

With its operation flexibility and a range of available performance options, HashCache is well suited to providing the infrastructure for caching applications in developing regions. Not only does it provide competitive performance with the stringent resource constraint , but also enables new opportunities that were not possible with existing approaches. We believe that HashCache can become the basis for a number of network caching services, and are actively working toward this goal.

9 Acknowledgements

We would like to thank Jim Gettys and John Watlington for their discussions about OLPC's caching needs, and Marc Fiuczynski for arranging and coordinating our deployments in Africa. We also thank our shepherd, Michael Mitzenmacher as well as anonymous NSDI reviewers. This research was partially supported by NSF Awards CNS-0615237, CNS-0519829, and CNS-0520053. Anirudh Badam was partially supported by a Technology for Developing Regions Fellowship from Princeton University.

References

[1] AKAMAI TECHNOLOGIES INC. http://www.akamai.com/.

[2] BLOOM, B. H. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM 13* (1970), 422–426.

[3] Brewer, E., Gauthier, P., and McEvoy, D. Long-term viability of large-scale caches. In *Proceedings of the 3rd International WWW Caching Workshop* (1998).

[4] Chankhunthod, A., Danzig, P. B., Neerdaels, C., Schwartz, M. F., and Worrell, K. J. A hierarchical internet object cache. In *Proceedings of the USENIX Annual Technical Conference* (1996).

[5] Citrix Systems. http://www.citrix.com/.

[6] CoBlitz, Inc. http://www.coblitz.com/.

[7] Cox, A. L., Hu, Y. C., Pai, V. S., Pai, V. S., and Zwaenepoel, W. Storage and retrieval system for WEB cache. U.S. Patent 7231494, 2000.

[8] Du, B., Demmer, M., and Brewer, E. Analysis of WWW traffic in Cambodia and Ghana. In *Proceedings of the 15th International conference on World Wide Web (WWW)* (2006).

[9] Feldmann, A., Caceres, R., Douglis, F., Glass, G., and Rabinovich, M. Performance of web proxy caching in heterogeneous bandwidth environments. In *Proceedings of the 18th IEEE INFOCOM* (1999).

[10] Freedman, M. J., Freudenthal, E., and Mazieres, D. Democratizing content publication with coral. In *Proceedings of the USENIX Symposium on Networked Sytems Design and Implementation (NSDI)* (2004).

[11] Gadde, S., Chase, J., and Rabinovich, M. A taste of crispy Squid. In *Workshop on Internet Server Performance* (1998).

[12] Gribble, S., and Brewer, E. A. System design issues for internet middleware services: Deductions from a large client trace. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS)* (1997).

[13] Intel. Classmate PC, http://www.classmatepc.com/.

[14] Jin, S., and Bestavros, A. Popularity-aware greedydual-size web proxy caching algorithms. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS)* (2000).

[15] Luotonen, A., Henryk, F., Lee, T. B. http://info.cern.ch/hypertext/WWW/Daemon/Status.html.

[16] Maltzahn, C., Richardson, K., and Grunwald, D. Reducing the disk I/O of Web proxy server caches. In *Proceedings of the USENIX Annual Technical Conference* (1999).

[17] Maltzahn, C., Richardson, K. J., and Grunwald, D. Performance issues of enterprise level web proxies. In *Proceedings of the ACM SIGMETRICS* (1997).

[18] Markatos, E. P., Pnevmatikatos, D. N., Flouris, M. D., and Katevenis, M. G. Web-conscious storage management for web proxies. *IEEE/ACM Transactions on Networking 10*, 6 (2002), 735–748.

[19] Mitzenmacher, M. The power of two choices in randomized load balancing. *IEEE Transsactions on Parallel and Distributed Systems 12*, 10 (2001), 1094–1104.

[20] Mogul, J. C., Chan, Y. M., and Kelly, T. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2004).

[21] O3b Networks. http://www.o3bnetworks.com/.

[22] OLPC. http://www.laptop.org/.

[23] OLPC. http://wiki.laptop.org/go/Hardware_specification.

[24] OLPC. http://wiki.laptop.org/go/XS_Recommended_Hardware.

[25] Pai, V. S., Druschel, P., and Zwaenepoel, W. Flash: An efficient and portable Web server. In *Proceedings of the USENIX Annual Technical Conference* (1999).

[26] Park, K., and Pai, V. S. Scale and performance in the CoBlitz large-file distribution service. In *Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2006).

[27] Patra, R., Nedevschi, S., Surana, S., Sheth, A., Subramanian, L., and Brewer, E. WiLDNet: Design and implementation of high performance wifi based long distance networks. In *Proceedings of the 4th Symposium on Networked Systems Design and Implementation (NSDI)* (2007).

[28] Rhea, S., Liang, K., and Brewer, E. Value-based web caching. In *In Proceeding of the 13th International Conference on World Wide Web (WWW)* (2003).

[29] Riverbed Technology, Inc. http://www.riverbed.com/.

[30] Rosenblum, M., and Ousterhout, J. K. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems 10*, 1 (1992), 26–52.

[31] Russell, M., and Hopkins, T. CFTP: a caching FTP server. *Computer Networks and ISDN Systems 30*, 22–23 (1998), 2211–2222.

[32] Seznec, A. A case for two-way skewed-associative caches. In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)* (New York, NY, USA, 1993), ACM, pp. 169–178.

[33] Shriver, E. A. M., Gabber, E., Huang, L., and Stein, C. A. Storage management for web proxies. In *Proceedings of the USENIX Annual Technical Conference* (2001).

[34] Silver Peak Systems, Inc. http://www.silver-peak.com/.

[35] Squid. http://www.squid-cache.org/.

[36] Subramanian, L., Surana, S., Patra, R., Nedevschi, S., Ho, M., Brewer, E., and Sheth, A. Rethinking wireless in the developing world. In *Proceedings of Hot Topics in Networks (HotNets-V)* (2006).

[37] The Measurement Factory. http://www.web-polygraph.org/.

[38] The Measurement Factory. http://www.web-polygraph.org/docs/workloads/polymix-4/.

[39] The Measurement Factory. http://www.measurement-factory.com/results/public/cacheoff/N04/report.by-alph.html.

[40] The Measurement Factory. http://www.measurement-factory.com/results/public/cacheoff/N03/report.by-alph.html.

[41] Wang, L., Park, K., Pang, R., Pai, V., and Peterson, L. Reliability and security in the CoDeeN content distribution network. In *Proceedings of the USENIX Annual Technical Conference* (2004).

[42] WiMax Forum. http://www.wimaxforum.org/home/.

iPlane Nano: Path Prediction for Peer-to-Peer Applications

Harsha V. Madhyastha[*] *Ethan Katz-Bassett*[†] *Thomas Anderson*[†]
Arvind Krishnamurthy[†] *Arun Venkataramani*[‡]

Abstract

Many peer-to-peer distributed applications can benefit from accurate predictions of Internet path performance. Existing approaches either 1) achieve high accuracy for sophisticated path properties, but adopt an unscalable centralized approach, or 2) are lightweight and decentralized, but work only for latency prediction.

In this paper, we present the design and implementation of *iPlane Nano*, a library for delivering Internet path information to peer-to-peer applications. *iPlane Nano* is itself a peer-to-peer application, and scales to a large number of end hosts with little centralized infrastructure and with a low cost of participation. The key enabling idea underlying *iPlane Nano* is a compact model of Internet routing. Our model can accurately predict end-to-end PoP-level paths, latencies, and loss rates between arbitrary hosts on the Internet, with 70% of AS paths predicted exactly in our evaluation set. Yet our model can be stored in less than 7MB and updated with approximately 1MB/day. Our evaluation of *iPlane Nano* shows that it can provide significant performance improvements for large-scale applications. For example, *iPlane Nano* yields near-optimal download performance for both small and large files in a P2P content delivery system.

1 Introduction

Peer-to-peer (P2P) systems offer a number of potential advantages to the network systems designer, such as scalability, resilience, and perhaps most importantly, cost-effectiveness: P2P systems require little or no fixed infrastructure, and yet can scale to millions of end hosts. These advantages have provoked considerable interest in the P2P design paradigm among researchers [10, 14, 44]. There have also been several widespread deployments, including BitTorrent file sharing [11], Skype's use of detour routing for voice over IP [52], and multiplayer game servers that reduce bandwidth costs by using well-provisioned players to distribute objects to other peers [4].

In this paper, we argue that a key missing piece of infrastructure for P2P applications is scalable and inexpensive access to accurate information about Internet paths. P2P applications by their nature select among a large number of alternative paths; more accurate information can help streamline that search process. For example, a P2P content distribution network [45, 38, 25] might benefit from directing requests to a replica with a low latency, low loss path. Similarly, an IP layer detour routing service would benefit from structural information about the Internet, to quickly find a path around a network failure [59, 23].

While server-based solutions for providing timely information about the Internet have been proposed and built in the past [30, 1], they are less appropriate in the P2P case. The iPlane [30] query engine, for example, runs as a service, but since its algorithms require multi-gigabyte memory resident data structures to generate predictions, it would be difficult and costly to scale, especially for a popular P2P application with millions of end hosts. iPlane's memory footprint means it cannot even run on PlanetLab [41]. Further, iPlane's data cannot be easily distributed given its size and running the service on a few nodes in turn significantly limits the rate at which queries can be served.While network coordinate systems [13] do scale, they only predict latency, and not the full range of topology-aware performance metrics needed by P2P applications.

To address this gap, we have designed and built a system called *iPlane Nano*, or *iNano*. *iNano* uses the same input data and provides the same query interface as iPlane, but is designed as a lightweight library that can run on client machines, and even on small devices such as Internet-capable smart phones. To make this work, we have developed a compact model of Internet topology, routing policy, and link performance metrics that can be represented in less than 7MB, and updated with approximately 1MB/day. Yet this model is rich enough to be able to accurately predict end to end routes, latencies, and loss rates between arbitrary end hosts on the Internet. In our evaluation, we find that *iNano* predicts 70% of AS paths exactly, estimates latencies with less than 20ms of error for over 60% of paths, and estimates loss rates with less than 10% error for over 80% of paths.

Because *iNano*'s data set is the same for all end hosts, both the model and its incremental daily updates can be efficiently distributed using standard file sharing techniques, such as via BitTorrent swarms. Our evaluation shows that although our predictions are based on only a tiny fraction of the total information available about the Internet, *iNano* can significantly improve application performance. For example, *iNano* yields near-optimal me-

[*]University of California, San Diego
[†]University of Washington
[‡]University of Massachusetts Amherst

dian download performance for both small and large files in a P2P content delivery system.

In summary, our primary contribution is to develop an accurate yet lightweight approach for Internet performance prediction. To this end, we develop:

- A pocket-sized, annotated link-level map of the Internet, that can be represented in 7MB and updated daily with 1MB of data.
- Techniques to infer and concisely represent information stored in the forwarding tables of Internet routers, but in orders of magnitude lesser space.
- Implementation of *iNano*, a system that enables Internet-scale P2P applications to discover properties of Internet paths.
- Case studies using CDNs, VoIP, and detour routing to demonstrate the utility of *iNano*.

2 Motivation and Design Goals

2.1 Goals

iNano targets network applications that choose among multiple candidate paths to improve data transfer performance. The design goals of *iNano* and their motivations are as follows.

Rich path metrics: *iNano* should enable distributed applications to orchestrate their actions based on sophisticated path information. Application-perceived path performance may depend on one or more path metrics such as latency, loss rate, or bottleneck capacity. For example, TCP performance depends upon the latency as well as loss rate along the path, so a CDN re-director or BitTorrent tracker may wish to use both metrics in its decisions. A VoIP server such as in Skype may wish to pick a relay node according to the mean-opinion-score (MOS) metric [5] that depends upon loss rate and latency. Live video streaming systems [3, 2, 10] that set up an overlay network among participating end-hosts may wish to incorporate path metrics such as latency, loss rate, and bottleneck capacity in the construction of the overlay. A combination of these metrics determines the quality of the video a client receives as well as its initial buffering delay.

Scalable lookup: *iNano* should scale to every end-host in the Internet. The trend towards massively distributed applications such as CDNs, BitTorrent, and Skype suggests that the potential demand for path performance prediction requests may be comparable to DNS or web search. Given the frequent occurrence of detour routes [48, 29], it is conceivable that every transfer is preceded by a query about alternative paths to the destination. Furthermore, the lookups must be local to be effective; otherwise, the delay incurred may outweigh the resultant improvement in data transfer performance.

Low infrastructure cost: *iNano* should incur a low infrastructure cost to set up and maintain. A server-based infrastructure will need to be continually provisioned as demand increases and will incur significant cost to deploy and maintain. Instead, *iNano* should leverage the property of P2P applications—users not only create demand but also contribute resources to the system—by using computing cycles and bandwidth on participating end-hosts rather than on dedicated servers.

Structural information: *iNano* should enable network applications to base their decisions on the structure of the path. For example, recent proposals have advocated locality-aware peer selection in peer-to-peer systems by either choosing paths that minimize the AS path length [9] or by jointly optimizing network cost and application performance [57]. Knowing the route can also enable applications to perform detour [48, 7] or multipath routing [58, 24] for reliability or performance objectives. Structural information can also be used to route around network failures [59, 23].

Arbitrary end-hosts: *iNano* should enable an application to infer path information between an arbitrary pair of end-hosts, not just from itself to others. Many of the examples above involving redirection in peer-to-peer content distribution, VoIP relays, multicast overlay construction, and detour routing require this capability. Furthermore, *iNano* should provide forward as well as reverse path information between arbitrary end-hosts—a goal that is challenging even for paths originating locally because of the asymmetric nature of Internet routing.

2.2 Exploring design alternatives

Why can't existing techniques achieve the above goals? To appreciate the challenge, let us consider a few natural design alternatives as shown in Table 1.

A1 is the well-studied network coordinates approach to infer latencies between end-hosts without on-demand measurement. In this approach, each end-host is assigned a coordinate, typically in a metric space, and the latency between two end-hosts is estimated as the distance between their coordinates. Distributed systems such as Vivaldi [13] implement the coordinate approach in a scalable manner. However, the only information they provide to an application running on an end-host is the latency on paths from that end-host to the rest of the Internet. Although the coordinate system could potentially be modified to predict latencies between arbitrary end-hosts by periodically disseminating a coordinate for every Internet prefix, it is unclear how to extend this approach to other path metrics such as loss rate or bottleneck capacity. Also, since coordinate systems rely only on end-to-end measurements, they do not provide information on the route traversed by a path.

A2 is an approach where applications issue queries about path performance to a network information ser-

	Design alternative	Rich path metrics	Structural information	Arbitrary end-hosts	Scalable	Infrastructure cost
A1	Network coordinates	×	×	×	√	0
A2	Information plane (e.g., iPlane) servers	√	PoP path	√	×	High
A3	Information plane as "network newspaper"	√	PoP path	√	×	Low
A4	Uncoordinated end-host measurements	√	PoP path	×	×	Low
A5	*iNano*	√	PoP path	√	√	Low

Table 1: Qualitative comparison of design alternatives for Internet path performance prediction.

vice hosted on centralized or replicated query servers. This approach is suggested and made plausible by prior work, namely iPlane, that developed techniques to accurately predict the path and path metrics between an arbitrary pair of end-hosts. However, scaling replicated query servers to handle requests from all end-hosts—a workload comparable to DNS—is challenging and would incur a huge infrastructure cost to set up and maintain. The number of query servers provisioned will need to grow in proportion to the number of end-hosts issuing queries, making this approach impractical for typical P2P applications.

A3 replicates a query server on each end-host. This approach dubbed as "network newspaper" in [30] would disseminate an atlas of measured Internet paths to end-hosts to enable them to locally service their queries. The atlas can be refreshed daily by sending incremental updates; since most Internet paths do not change over a day [40], daily updates are expected to be small. Unfortunately, iPlane's atlas of paths is several gigabytes in size, making this approach unlikely to be adopted in practice. An alternative is to delegate this task to a local agent (like a local DNS nameserver) in each subnet, but the bootstrapping overhead would pose a barrier to widespread deployment and use. Another alternative is for each client to only download its "view" of the network, i.e., properties of paths originating at itself, but this approach does not allow an end-host to predict properties of paths between arbitrary end-hosts, e.g., as required to enable detour routing.

A4, where each end-host conducts its own measurements as needed, also suffers from the problem of not being able to predict properties of paths between arbitrary end-hosts. Furthermore, such uncoordinated measurements might impose an unreasonable measurement overhead, e.g., measurement of loss rates and bandwidth capacities require many large-sized packet probes to be sent into the network. A centralized coordinator and aggregator of measurements like iPlane amortizes this overhead, but makes dissemination a challenge as discussed in A2 and A3.

3 *iNano* Design

Our system, *iNano*, combines the best of the above alternatives. For scalability, *iNano* replicates query servers at each end-host. To predict rich path metrics, *iNano* uses a structural technique like iPlane that predicts the PoP-level [1] path between an arbitrary pair of end-hosts. However, the data required to make such predictions needs to be compact, like coordinates or like the AS-level Internet graph, unlike a huge atlas of measured paths.

The key insight in *iNano* is a novel model for predicting paths and their properties between arbitrary end-hosts using a compact Internet atlas. iPlane uses a path composition technique to perform path predictions. To predict the path from a source to a destination, the path composition technique composes two path segments that intersect with each other. The first segment is from a path out from the source to an arbitrary destination. The second segment is from a path measured from one of iPlane's vantage points to the destination's prefix. Depending on which intersecting pair of segments is chosen, the path obtained by composition is often similar to the actual route from source to destination.

Instead of using an atlas of measured paths like iPlane's, *iNano* uses an atlas of measured links. The space required by the former representation is proportional to the number of vantage points while the latter representation requires space linear in the number of nodes and edges in the underlying Internet graph. Consequently, *iNano*'s atlas fits in less than 7MB, almost three orders of magnitude smaller than iPlane's atlas, enabling it to be distributed to lightly powered end-hosts. The key challenge in making this approach work is to make accurate predictions about Internet path performance from an atlas of observed links.

iNano's approach of distributing a compact atlas and locally resolving queries at end-hosts avoids significant investment in server infrastructure. The approach also offloads the bandwidth cost of disseminating the atlas and its periodic updates; the atlas can be swarmed among end-hosts using, for example, BitTorrent. The genera-

[1] A Point-of-Presence (PoP) of an AS is the set of routers in that AS in the same location.

tion of the atlas itself is the only centralized component in *iNano*. A central coordinator distributes the task of issuing measurements to participating end-hosts and aggregates the measured paths into a set of measured links.

iNano's current measurement infrastructure is largely the same as that of iPlane [30] but processes the measurements in a completely different manner to make path performance predictions in keeping with the goals stated in Section 2. Although we use end-host measurements in building the atlas, we use as a starting point traceroutes from PlanetLab [41] to destinations in 140K prefixes, which include roughly 90% of prefixes at the Internet's edge. The interfaces discovered in the traceroutes are clustered together such that interfaces in the same Point of Presence (PoP) within an AS are in the same cluster; routers in the same PoP within an AS are similar from a routing perspective. To map the IP address of an interface to its corresponding AS, *iNano* uses the mapping from prefixes to their origin ASes as seen in BGP feeds [33] and also resolves aliases [53] to ensure different interfaces on the same router are mapped to the same AS. The clustering of interfaces in each AS into PoPs is performed using a combination of alias resolution, mapping DNS names to locations [55], and identifying colocated interfaces based on similarity in reverse path lengths.

iNano processes the gathered traceroutes in combination with the PoP clustering information to build an atlas of inter-cluster links. To annotate links in this atlas with performance metrics, *iNano* performs measurements to infer the latencies and loss rates of inter-cluster links. *iNano* uses the frontier search algorithm described in [30] to partition the set of links across the PlanetLab vantage points, with some redundancy to account for measurement noise. Each node then attempts to measure the latency and loss rates of links assigned to it. The technique for measuring loss rates is the same as that used by iPlane. Measuring latencies of links is hard due to the wide prevalence of asymmetric routing [40, 21]. *iNano* tackles this challenge using a two-pronged approach—first, by identifying symmetric paths, and second, by leveraging measurements of symmetric paths to measure latencies of other links that do not appear on symmetric routes. *iNano*'s link latency measurement techniques are described in [28]. To estimate the end-to-end latency and loss rate between a source and destination, *iNano* predicts the forward and reverse paths between these end-hosts and composes the properties of the inter-cluster links on the predicted paths.

4 Route Prediction

In this section, we develop an inference algorithm that predicts routes by composing observed links between routers. The set of observed links yields a graph capturing the Internet's physical topology. In order to predict an end-to-end route accurately, we need to compactly model the routing decisions made by routers along candidate paths in this graph.

This inference and modeling problem is not easy. Inferring routes would be easy using a naive model that explicitly stores the information contained in the forwarding tables of routers in the graph. However, that defeats our primary goal of predicting routes using a compact graph representation. Thus, the key challenge to developing a compact model is to understand and describe the procedure routers use to compute routes, i.e., to concisely describe how Internet routing works!

4.1 The Problem: Modeling Internet Routing

Compactly modeling Internet routing would be trivial if routers simply used shortest path routing. The weights used for shortest path computation could be inferred using existing approaches [31]. However, Internet route selection is driven by a number of factors such as routing policies driven by economic considerations, traffic engineering driven by load balancing goals, and performance considerations that can not be characterized as shortest path routing. Furthermore, end-to-end Internet routes are computed by a set of complex interacting protocols (such as BGP, OSPF, and RIP) rather than a single protocol.

Fortunately, we are aided by a large body of prior research on understanding and reverse-engineering the routing decision process, as well as the knowledge the research community has acquired on how Internet routing works in practice. These result in the following commonly accepted "textbook" principles about how Internet routing works.

1. *Policy preference*: ASes use *local preferences* to select routes. Typically, an AS prefers routes through its customers over those through its peers, and either of those over routes through its providers [2]. Further, ASes do not export all of their paths to their neighbors; for instance, ASes do not export paths through their peers to other peers/providers. Commonly used export policies and AS preferences are believed to result in *valley-free* Internet routes [19], in which any path that traverses a provider-to-customer edge or a peer-to-peer edge does not later traverse a customer-to-provider or peer-to-peer edge.

2. *Shortest AS path*: After applying local preferences, if a router has multiple candidate paths that it prefers equally, the default is to select the route containing the fewest ASes. Typically, several paths may have the same local preference and AS path length.

[2]Customer ASes pay their providers while peers connect to each other at no cost.

```
GRAPH($s, d$):
$N' \leftarrow \{d\}$
for each $v \in G$
  if $v$ is a neighbor of $d$, then $\mathbf{D}(v,d) = c(v,d)$;
  else $\mathbf{D}(v,d) = [\infty, \infty]$ ;
Do
  Pick $w \notin N'$ such that $\mathbf{D}(w,d)$ is a minimum
  $N' \leftarrow N' \cup \{w\}$;
  for each neighbor $v$ of $w$
    if $\mathbf{D}(v,d) > \mathbf{D}(w,d) \oplus c(v,w)$, then
      $\mathbf{D}(v,d) = \mathbf{D}(w,d) \oplus c(v,w)$;
      $P(v,d) = v.P(w,d)$;
until $N = N'$
```

Figure 1: The algorithm used by GRAPH to predict a valley-free route from s to d in a graph G. $\oplus$ is the operator that defines how edge weights compose in our application of Dijkstra's shortest path algorithm.

3. *Exit policies*: Among these, routes are chosen so as to meet intradomain objectives, e.g. by choosing the nearest exit point to the next AS (referred to as early-exit or hot potato routing) along the path. In some cases that often involve explicit compensation or negotiation among adjacent ASes to reduce their combined costs, ASes adopt a late-exit policy.

How well does the above procedure describe Internet routing? To evaluate this, we develop a simple algorithm based on dynamic programming that underlies various forms of shortest path computation. The algorithm incorporates the above criteria to compute an on-demand route, based on a graph representation of the Internet.

Our first attempt, GRAPH, reduces the representation size by over two orders of magnitude, but has poor prediction accuracy. This suggests that exceptions to the above criteria are common and must be carefully integrated into the model, as we describe in Sections 4.3.1–4.3.4.

4.2 GRAPH: A first cut

We present the algorithm in three steps. First, we describe a basic algorithm using dynamic programming (similar to Dijkstra's shortest path algorithm) that captures the preference for short AS paths, assuming early-exit between every pair of ASes. Second, we augment the algorithm to model late-exit when necessary. Third, we augment the algorithm to model common export policies and local preferences for routes.

4.2.1 Basic algorithm

Figure 1 shows the pseudocode for GRAPH, an algorithm that predicts the route between a source s and a destination d. It chooses the shortest AS path among all valley-free paths between s and d; further, it uses early-exit at every AS. The algorithm is similar to Dijkstra's shortest path algorithm. Unlike conventional Dijkstra however, the route computation 1) backtracks from the destination to all sources, and 2) uses a two-tuple cost metric.

The cost of a route from each node v to the destination d, represented as $\mathbf{D}(v,d)$, is a strictly ordered two-tuple [number of AS hops to the destination, cost to exit the current AS], with the first component considered as the more significant value. For two adjacent nodes v and w connected by a link of latency $l(v,w)$, the cost of the edge between them, represented as $c(v,w)$, is defined as $[0, l(v,w)]$ if v and w are in the same AS, and as $[1, 0]$ otherwise. The $\oplus$ operator in the algorithm resets the second component to 0 upon crossing an AS boundary as follows. If v and w belong to the same AS, $\mathbf{D}(w,d) \oplus c(v,w)$ is defined as $\mathbf{D}(w,d) + [0, l(v,w)]$, where '+' does the usual component-wise addition. If v and w belong to adjacent ASes, $\mathbf{D}(w,d) \oplus c(v,w)$ is defined as $[\mathbf{D}(w,d)[1] + 1, 0]$. It is straightforward to verify that this definition of cost preserves the invariant that if a node $u \in N'$, then $P(u,d)$ is a shortest path from u to d. As in Dijkstra's algorithm, this invariant ensures the correctness of the algorithm.

4.2.2 Incorporating late-exit

It is straightforward to extend the above algorithm to handle pairs of ASes that use late-exit instead of early-exit. We model late-exit as two adjacent ASes v, w (such as AS6380 and AS6389 – both of which are owned by Bell South) jointly computing the path through them in order to minimize the overall transit latency. To infer late exit, we use the technique proposed in [54]. We simply redefine the $\oplus$ operator in the following way. An inter-AS edge (v, w) corresponding to a late-exit route has $c(v,w) = [0, l(v,w)]$, meaning that it is treated as an intra-AS edge. We do however have to increment the AS hop count by two when we backtrack out of the AS containing v. This is accomplished by maintaining another component in the cost tuple that corresponds to the number of consecutive late-exit transitions. This component corresponds to the number of AS hops that are not yet accounted for in the AS path length component of the cost metric. Whenever an AS transition is traversed where late exit is not applied, this third component is added into the AS path length component and reset to zero.

4.2.3 Incorporating export policies

Next, we incorporate constraints corresponding to commonly used export policies. We infer AS relationships, such as which are peers and which have paid customer/provider transit , using a combination of CAIDA's inferences [16] and Gao's technique [19]. We model the default export policy in which an AS advertises any paths through customer ASes to all its neighbors, and it exports paths from peers and providers to only its customers. It is well-known that this export policy leads to valley-free

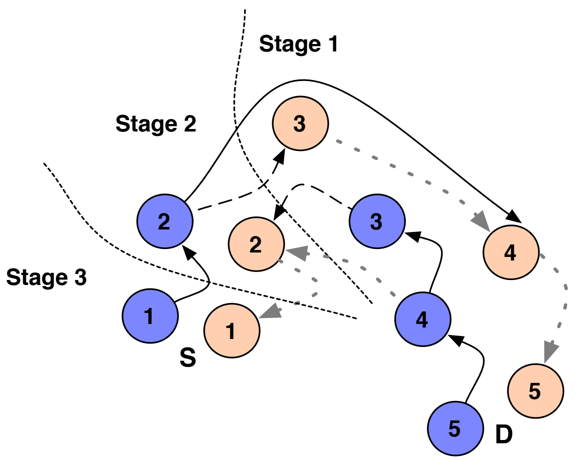


Figure 2: Route prediction from S to D so as to satisfy customer<peer<provider preferences. Dark nodes are down nodes, and light nodes are up nodes. Bold lines go from customers to their providers, dashed lines connect peers, and faded dotted lines go from providers to their customers. GRAPH traverses all the customer-to-provider edges in the first phase to finalize routes from 3 and 4 to D. Only peering links are traversed in the second phase making 2 choose a path through 3 over a shorter one via 4. Finally, provider-to-customer edges are traversed.

routes.

To compute valley-free routes, instead of having a single node for each cluster (PoP) i, we instead introduce two nodes in the graph: an *up* node up_i and a *down* node $down_i$, and GRAPH computes the path from up_s to $down_d$. The idea is that the construction of edges will force every path to transition from *up* nodes to *down* nodes at most once, thereby guaranteeing the path is valley-free. Let i and j be two clusters observed as adjacent.

1. If i and j belong to the same AS, there is an undirected edge between up_i and up_j and one between $down_i$ and $down_j$.

2. If i's AS is a provider of j's AS, there is a directed edge from up_j to up_i and another directed edge from $down_i$ to $down_j$. This edges capture that a customer will not provide transit between two providers.

3. If i and j belong to peer ASes, there is a directed edge from up_i to $down_j$ and from up_j to $down_i$. These edges capture that i's AS will use paths through j only for itself and its customers (and similarly for j's AS and paths through i).

Finally, for each IP address i, there is a directed edge from up_i to $down_i$. It is easy to verify that all routes in the graph are valley-free by construction (after transitioning from up to $down$, a transition from $down$ to up can no longer occur).

4.2.4 Incorporating local preferences

Next, we incorporate local preferences in selecting AS paths. We assume that an AS prefers paths through its customers over those through its peers, which are in turn preferable to paths through provider ASes. To incorporate these preferences, instead of calculating paths to the destination from all ASes and all routers in a batch, GRAPH computes routes in three phases.

Figure 2 illustrates the phased approach. GRAPH first limits the graph to contain only the set of *down* nodes, along with the edges connecting them, and computes the optimal paths from these nodes to the destination. This frontier reaches precisely the routers in those ASes that get paid for providing transit to the destination. Once all such nodes have been visited and their best paths discovered, the algorithm is allowed to reach any additional nodes that can be reached only using peering; by construction, only one peering is traversed. Finally, the algorithm is allowed to use any link (e.g., provider links) to reach all remaining addresses.

Results preview: As we show in detail in Section 6, GRAPH—despite taking into account many aspects of default routing behavior—correctly predicts only 30% of the AS paths for our measured dataset. In contrast, the path composition approach [30] (that dominates our achievable accuracy) achieves 70% accuracy using the entire set of observed routes.

On the other hand, the storage overhead of GRAPH is directly proportional to the number of observed Internet links. As we will see in the evaluation section, this is two orders of magnitude more compact than the path composition approach. Thus, the challenge is to improve GRAPH's accuracy while keeping it compact.

4.3 Addressing sources of prediction error

A careful examination of the above results reveals that GRAPH's inaccuracies arise partly from our failure to model certain other aspects of Internet routing behavior and partly from errors in inferred AS relationships. GRAPH's deficiencies are due to the following reasons.

1. *Asymmetry*: A significant fraction of Internet routes are asymmetric [40, 21]. While GRAPH reflects some asymmetry, e.g., due to early exit routing, it does not fully capture asymmetric policy behavior.

2. *Inaccurate export policy*: If GRAPH fails to identify a peer-to-peer relationship between two ASes, it is overly lenient in inferring export policy and predicts non-existent routes that would be filtered in practice.

3. *Incorrect local preferences*: An AS's customer may be a provider for specific paths. For example, two ASes may have different relationships in different

regions because one AS may have larger network presence than the other in one region and vice-versa in another region. Incorrect local preferences could result in an AS selecting a less preferable route, e.g., via a customer.

4. *Traffic engineering*: ASes may engineer routes in order to improve routing for their customer traffic compared to transit traffic.

We address each of these challenges by adding information in our data set back into the graph.

4.3.1 Addressing asymmetry

Due to the asymmetric nature of Internet routing, adding routes originating *from* the source to the atlas significantly improves the accuracy of predicted routes [29]. To reduce the likelihood of predicting non-existent routes, *iNano* splits the graph into two subgraphs: 1) TO_DST that consists of all directed links observed on the traceroutes from *iNano*'s vantage points to all prefixes, and 2) FROM_SRC that consists of all directed links on the traceroutes contributed to *iNano* by participating end-host sources.

For each cluster, we introduce a directed edge from its corresponding node in FROM_SRC to its corresponding node in TO_DST. *iNano* then predicts the route using the Dijkstra-style algorithm that backtracks from the $down$ node corresponding to the destination in TO_DST to the up node corresponding to the source in FROM_SRC. If it fails to find such a route, a likely scenario if the atlas lacks sufficient paths from the source prefix, then it attempts to find a path from the down node corresponding to the destination in TO_DST to the up node corresponding to the source in TO_DST.

4.3.2 Inferring export policies

GRAPH predicts non-existent routes that would be filtered given accurate AS relationships. Recall that we inferred the AS relationships automatically by analyzing observed behavior. Now, instead of explicitly distilling the AS relationships from the observed routes, we explore an alternate strategy that trades off a small amount of space for improved prediction accuracy. We seed *iNano* with known templates of export policy, e.g., if we observe a path that traverses the ASes Cogent, AT&T, and Sprint, we know that AT&T exports paths from Sprint to Cogent.

To implement this strategy, the valley-free check in GRAPH is replaced with the following *3-tuple check*. *iNano* explicitly stores the list of all 3-tuples corresponding to three consecutive ASes observed in traceroutes as well as BGP feeds (discounting prepending). Ideally, we would consider a predicted route valid only if all constituent segments of size three satisfy the 3-tuple check by appearing in the list, meaning that the path was exported

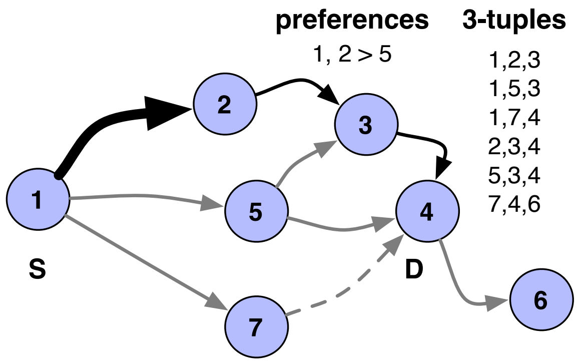


Figure 3: Predicting the path from S to D. Thicker lines show preferences, dashed lines show non-provider links, and dark lines show the prediction. *iNano* cannot choose $1-5-4$ because the 3-tuple does not appear and cannot choose $1-7-4$ because 7 is not a provider for 4. It predicts $1-2-3-4$ because of the preference for 2 over 5.

at every intermediate AS. In Figure 3, we see that, even though it is shorter, *iNano* cannot choose path $1-5-4$ because the 3-tuple $(1, 5, 4)$ does not appear in any BGP advertisement or traceroute. *iNano* easily incorporates the check in the backtracking step of the algorithm. However, since visibility into ASes at the edge is limited, we might fail to observe all of the export policies for the edge ASes. *iNano* thus performs this check only for 3-segments in which the degree of the middle AS in the Internet's AS-level graph is greater than a threshold (5 in the current implementation). Finally, we assume commutativity among triples, so that if we observe (AS1, AS2, AS3), we include (AS3, AS2, AS1) as well.

4.3.3 Improving local preferences

Recall that we infer AS relationships and incorporate the customer<peer<provider preference order in the route prediction algorithm. Unfortunately, AS relation inference by itself is difficult and error-prone. For example, AS relationship inference based on Gao's algorithm [19] predicts that half of the edges observed between the top hundred ASes ranked by degree correspond to sibling relationships, which seems rather implausible. The 3-tuple check by itself is not sufficient; although it ensures that predicted routes consist only of observed tuples, it does not take AS preferences into account when multiple options are available.

iNano uses a relationship-agnostic method to infer AS preferences based only on observed routes. We infer these preferences using the entire set of observed paths, but include only the results of the inferences within the compressed link-level representation of the atlas. The technique works as follows. For each observed AS route r, let $r_1, \ldots, r_m$ be the set of alternative routes available from the source, visible in the topology but not taken. For each route r_i, if r and r_i share the first k ASes but differ at the $(k+1)$'th AS, then the k'th AS is said to prefer the $(k+1)'th$ AS on r over the $(k+1)'th$ AS on r_i. Each alternative route in the set $r_1, \ldots, r_m$ similarly yields a preference.

iNano stores the preferences obtained above as 3-tuples (AS1, AS2 $>$ AS3), where AS1 prefers a route through AS2 over a route through AS3 when both routes are of the same length. In Figure 3, *iNano* selects the path $1-2-3-4$ over the path $1-5-3-4$ because of a preference (1, 2 $>$ 5). In some cases, we observe both 3-tuples (AS1, AS2 $>$ AS3) and (AS1, AS3 $>$ AS2). So, we include the preference (AS1, AS2 $>$ AS3) only if it was observed at least three times as often as the preference (AS1, AS3 $>$ AS2). If not, we ignore both preferences; we conjecture that such wavering preferences are likely due to load balancing by AS1. While some AS preferences might be restricted to paths from specific source prefixes or to specific destination prefixes, *iNano*'s model of Internet routing currently captures only preferences valid across sources and destinations. However, as we show in our evaluation, this suffices to significantly improve prediction accuracy.

4.3.4 Incorporating traffic engineering

In many cases, we observe an edge from AS1 to AS2 on some route in the atlas, but never see this edge on a route terminating at AS2, i.e., when the destination is in AS2. This occurs when an AS provides transit using one policy but routes to its own prefixes using a different policy, e.g., AS2 provides transit from AS1 to other ASes but does not send out BGP updates to AS1 for its own prefixes. The optimizations described above, the 3-tuple check and AS preferences, are insufficient to handle such cases.

To address the problem, *iNano* explicitly maintains information about provider ASes. For each AS, we determine its upstream neighbor ASes, i.e., the set of ASes observed immediately prior to this AS in the atlas. We also determine the set of providers for each AS, i.e., the set of ASes observed upstream of this AS when it is the origin. For the latter, we use both our traceroute data as well as BGP snapshots [33, 47]. For 1,352 ASes out of a total of 27,515 ASes in the atlas, we find the set of providers to be a proper subset of the set of upstream neighbors. In such cases, the previous algorithms could give the wrong path. We refine the approach further to determine the provider and upstream neighbor sets on a per-prefix basis. In Figure 3, *iNano* cannot select the path $1-7-4$, even though it is shorter, because 7 is not a provider for 4.

5 Implementation of *iNano*

Our implementation of *iNano* can roughly be divided into two logical components—server-side and client-side. The primary function of the server-side implementation is to gather measurements and to build the link-based atlas as described in the previous section. In addition, the *iNano* server bootstraps the distribution of the atlas to end-hosts.

The client-side implementation comprises a library providing information about Internet paths. The library performs four functions—fetching the atlas, augmenting the atlas with local measurements, servicing queries for path information from applications, and keeping the atlas up-to-date.

Fetching the Atlas: On startup, the *iNano* library fetches the atlas required for making predictions. The atlas fetched includes the following datasets: the set of inter-cluster links annotated with latencies and loss rates, data to map IP addresses to prefixes and ASes, AS degrees, AS 3-tuples, AS preferences, and the set of providers for each AS. Having all end-hosts fetch the atlas from *iNano*'s server would require an extremely large amount of bandwidth to be provisioned at the server. This would significantly drive up the cost required to run and maintain *iNano*.

Therefore, we instead rely on *swarming* the atlas across clients in order to distribute it. *iNano*'s central server serves as the seed for the dissemination of the atlas. In addition, every end-host running the *iNano* library makes available the portion of the atlas it has downloaded for other end-hosts to download. We have made our implementation sufficiently modular that any peer-to-peer filesharing protocol can be plugged in for distribution of the atlas. Our current implementation uses CoBlitz [39] and we are working on a version that uses BitTorrent [11].

Client-side Measurements: As previously explained in Section 4.3.1, *iNano* explicitly incorporates path asymmetry into its prediction model to improve the accuracy of path prediction. To enable this, *iNano*'s library includes a measurement toolkit used to gather measurements of the Internet from the perspective of end-hosts. The library uses this toolkit to issue traceroutes daily to destinations in a few hundred prefixes, chosen at random from all the routable prefixes in the Internet. The new links discovered as part of these traceroutes are added to the FROM_SRC plane of the atlas. The library also uploads the measured traceroutes to the central server. The server incorporates these measurements into the atlas distributed out to all end-hosts. Buggy or malicious clients could distort the atlas by contributing incorrect or fabricated measurements. While such discrepancies could be inferred by comparing with measurements from other clients, we leave such inference to future work.

Serving Queries: Once the atlas is fetched and augmented with client-side measurements, the library starts up a local query server. This query server implements the prediction algorithm developed in Section 4. The API exported by the library enables applications to query for information on paths between (src, dst) IP address pairs

Dataset	No. of entries		Compressed file size (in MB)	
	Atlas	Delta	Atlas	Delta
Inter-cluster links with latencies	309K	121K	1.99	0.49
Link loss rates	47K	65K	0.21	0.29
Prefix to cluster	140K	0	0.76	0
Prefix to AS	287K	0	1.67	0
AS degrees	28K	0	0.09	0
AS three-tuples	1.05M	230K	1.23	0.56
AS preferences	9K	0	0.03	0
Provider mappings	33K	0	0.63	0
Total			**6.61**	**1.34**

Table 2: Current size of *iNano*'s atlas, in terms of number of entries, compressed bytes on disk, and the delta between consecutive days.

in batches of arbitrary sizes. In future work, we plan to support remote queries so that only one local host need download the atlas.

Keeping Atlas Up-to-date: Paths and path properties on the Internet change over time. Hence, *iNano*'s atlas needs to be kept up-to-date to reflect current network conditions. Fortunately, the stationarity of Internet routing keeps the bandwidth cost of such updates low. A significant fraction of Internet routes are stationary [40] across days and path properties are stationary [60, 30] on the timescale of several hours. Therefore, as we show later in our evaluation, the difference between the atlases of consecutive days can typically be represented in approximately 1MB. As a result, once an end-host fetches the complete atlas, it can maintain an up-to-date atlas thereafter by downloading a daily 1MB update also as a swarmed file download.

6 Evaluation

In this section, we evaluate the accuracy of *iNano*'s predictions of paths and path properties, and study the contribution that each of *iNano*'s components makes towards its predictive ability. We also quantify the stationarity of *iNano*'s atlas across days, *iNano*'s storage requirements, and how the atlas size would grow with additional vantage points.

6.1 Size of the atlas

First, we discuss the typical size of *iNano*'s atlas and then evaluate how this size would scale with measurements from more vantage points.

6.1.1 What is the current size of the atlas?

We describe a typical day's atlas that we use for most of the evaluation in this section. We leverage PlanetLab nodes as vantage points for gathering the *iNano* atlas. The atlas we use in our evaluation comprises traceroutes from 197 PlanetLab nodes to one destination each in 140K prefixes. All of these traceroutes were gathered over the course of a day. After alias resolution and clustering, 85K distinct clusters are present in the atlas, with 309K links between them. The dataset obtained by combining these inter-cluster links annotated with latencies and loss rates, observed AS 3-tuples, inferred AS preferences, and the mapping of ASes to their providers is roughly 6.6MB in size. AS 3-tuples, the dataset with the most number of entries, are highly amenable to compression because only 2500 ASes, less than 10% of all the ASes in the atlas, occur as the middle component of any 3-tuple. Table 2 shows the size associated with each of these components of the atlas.

6.1.2 Does *iNano*'s atlas scale w.r.t vantage points?

iNano uses measurements from end-hosts to improve prediction accuracy for asymmetric routes. However, adding more measurements could significantly inflate the size of *iNano*'s atlas, questioning the basic tenet of our work—is the atlas still tractable if it includes measurements from millions of end-hosts?

To study this question, we use the DIMES measurement infrastructure [50]. The DIMES project runs an Internet measurement agent on a few thousand end-hosts distributed worldwide. We issued traceroutes from 845 DIMES agents to 100 randomly chosen destinations each over the course of a week.

The addition of measurements from more vantage points primarily impacts the number of inter-cluster links and the number of AS three-tuples in the atlas. As stated previously, measurements from PlanetLab find approximately 309K links and 1.05M AS three-tuples. Including the measurements from the 845 DIMES agents into the atlas added approximately 16K links and 14K AS three-tuples in total. Even though the addition of links from more vantage points is likely to be sublinear in practice, we extrapolate linearly to get a conservative estimate of the increase in the size of the atlas if we had measurements from all of the Internet's edge. Including traceroutes from end-hosts in all 100K prefixes at the Internet's edge would increase the number of links in the atlas from 309K to approximately 2.2M (16K new links added for every 845 hosts), an eight-fold increase, and the number of AS three-tuples from 1.05M to 2.7M (14K new three-tuples for every 845 hosts), a three-fold increase. Assuming this data is as compressible as the PlanetLab data, this would add 18MB to the atlas and 5MB to the daily update. It is future work to determine how much of this data is truly needed, discarding information that adds little in terms of added accuracy.

6.2 Stationarity of measurements

iNano refreshes its atlas once every day. To evaluate whether the interval of a day between updates suffices, we examine the stationarity of the

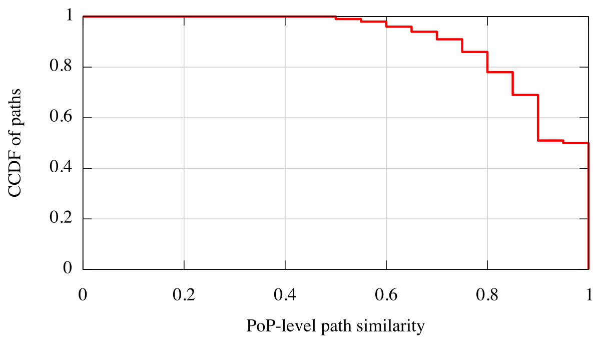


Figure 4: Similarity of PoP-level paths across consecutive days for routes measured from 195 PlanetLab nodes to destinations in 140K prefixes.

two kinds of measurements—traceroutes and loss rate measurements—used to construct *iNano*'s atlas. Our link latencies do not capture transmission and queueing delays, and hence, are extremely stable. We then present the size of the difference between successive atlases that arises as a result of the stationarity in measurements.

6.2.1 How stationary are routes?

We studied the stationarity of routing by comparing the traceroutes measured from each of 195 PlanetLab nodes to destinations in 140K prefixes on successive days. Since *iNano* only considers the Internet topology at the granularity of clusters corresponding to PoPs, we map traceroutes to cluster-level paths for comparison. We compared every path between a PlanetLab node and a destination on one day with the same path the next day using the path similarity metric [22, 29]. The similarity metric compares two paths as the ratio of the size of the intersection to the size of the union, of the sets of clusters in each of the paths; the ordering of clusters in the paths is not considered. The maximum value of this metric is 1 when both paths pass through exactly the same set of clusters, and the minimum value is 0 when the paths are completely disjoint. Figure 4 shows the distribution of PoP-level path similarity we obtained by comparing paths across consecutive days, grouping the similarity values into bins of 0.05. 91% of the paths on the first day have a similarity of at least 0.75 with the corresponding paths measured the next day, 68% have a similarity of at least 0.9, and 50% remain identical.

The main prior work on studying path stationarity has been by Paxson [40] and Zhang *et al.* [60]. Both observed more stationarity in routes than we do—Paxson found 68% of paths to be identical across days at the granularity of routers, and Zhang *et al.* found the same number to be more than 75%. We believe the difference in our findings is due to our significantly larger dataset. Paxson's measurement dataset included traceroutes between 27 vantage points and Zhang *et al.* used traceroutes between 220 vantage points. In contrast, our analysis of path stationarity uses traceroutes from 195 vantage points to 140K destinations each.

6.2.2 How stationary are loss rates?

To evaluate the stationarity of packet loss, we probed paths from 201 PlanetLab nodes to destinations in 5000 randomly chosen prefixes each. We sent out 100 ICMP probes of size 1KB on each path, with successive probes separated by 2 seconds, and determined the fraction of probes for which we received no response. We repeated these loss measurements 6 hours later. We found that 66% of paths on which we originally observed packet loss continued to be lossy 6 hours later. We also repeated these measurements 12 hours and 24 hours after the original measurements. The fraction of lossy paths that continued to remain so decreased from 66% to 53% when the interval between measurements was increased from 6 hours to 12 hours but remained steady at 53% when the interval was increased further to 24 hours.

6.2.3 How stationary is *iNano*'s atlas?

As a result of the significant stationarity seen in both paths and path properties over the interval of a day, the difference between *iNano*'s atlases on consecutive days is much smaller in size than the atlas itself. To update the atlas from the previous day, *iNano* ships the union of the old entries not present any more and new entries added to the inter-cluster links, link loss rates, and observed AS three-tuples datasets. The size of the link loss rates delta is larger than the loss rates dataset itself because we have to update a link's loss rate not just when it changes from being lossless to lossy (or vice-versa), as in our study on stationarity of loss above, but also when the link's loss rate changes. All the other datasets do not change on a day-to-day basis and hence, are updated in full only once a month. Table 2 shows that the typical difference is 1.34MB in size, less than one-fifth the typical size of a complete atlas. This implies that once an end-host downloads *iNano*'s atlas, it can keep its local information up-to-date by fetching a significantly smaller update daily thereafter.

6.3 Accuracy of Predictions

We next evaluate the accuracy of *iNano*'s predictions of both paths and path properties. From the 197 vantage points used in gathering the atlas described in Section 6.1.1, we choose a subset of 37 at random as our representative end-hosts. We pick 100 random traceroutes performed from each of them. After discarding paths that do not reach the destination or have AS-level loops, we are left with a validation set of 2816 paths. To predict the paths and path properties from one of the 37 sources, we include links from all traceroutes from the remaining 196 vantage points in the TO_DST plane and links from 100 other randomly chosen traceroutes from this source in the

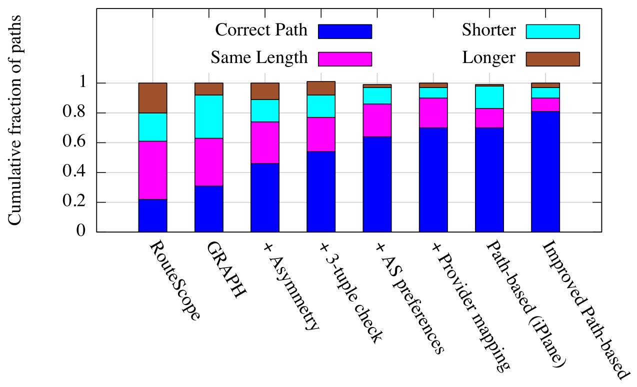


Figure 5: AS path prediction accuracy for measured traces as components are incorporated into *iNano*. RouteScope is the algorithm from [32], GRAPH is the algorithm described in Section 4.2, and path-based is the iPlane algorithm. Improved path-based incorporates *iNano*'s techniques into the iPlane algorithm.

FROM_SRC plane.

6.3.1 Can *iNano* predict AS paths accurately?

We evaluate the accuracy of *iNano*'s ability to predict the AS paths in our validation set. We evaluate the accuracy of *iNano*'s path prediction only at AS-level and not at PoP-level because our dataset clustering router interfaces into PoPs is complete. As a result, when our clustering indicates that two PoP-level paths are not identical, it is hard to say whether the difference is because of the incompleteness of our clustering data or they are indeed different. In contrast, our mapping from IPs to ASes is significantly more comprehensive.

Figure 5 shows the improvement in accuracy of AS path prediction as each component of *iNano* is incorporated into the GRAPH algorithm. The fraction of paths for which we predict the AS path exactly right increases from 31% with GRAPH to 70% with all components of *iNano* included. Each of the four techniques that *iNano* uses significantly improves *iNano*'s ability to predict paths. In fact, our final predictive model achieves the same AS path accuracy as iPlane's path composition technique, which uses a path-based dataset two orders of magnitude larger than *iNano*'s link-based atlas. Furthermore, *iNano* outdoes path composition in the ability to predict AS path length.

Figure 5 also compares *iNano*'s AS path prediction accuracy with that of RouteScope [32], the only prior work that predicts AS paths from a graph representation of Internet topology. First, RouteScope computes relationships between ASes using an observed set of AS paths as input. However, to predict the path between a *(src, dst)* pair, it needs only the AS-level graph of the Internet. RouteScope computes the set of shortest AS paths determined to be valley-free between the AS of *src* and the AS of *dst*. For the problem setting targeted by *iNano*, a single predicted path is required to estimate end-to-end performance. Therefore, to evaluate the utility of RouteScope in this setting, we choose one path at random from the set of paths returned by RouteScope for each *(src, dst)* pair. RouteScope's accuracy at predicting AS path length is only as good as that of GRAPH, and its accuracy at predicting the correct AS path is worse than GRAPH's. *iNano*'s significantly better accuracy stems from its modeling of Internet routing at PoP-level instead of AS-level and its modeling of routing with techniques beyond simple valley-free routing.

iNano's techniques are also applicable to a structural approach that works by composing path segments. We incorporate these techniques into iPlane's path composition algorithm to improve the accuracy of prediction using an atlas of paths. When two path segments are being spliced together, we check whether the sequence of ASes prior to, at, and after the point of intersection exists in our database of 3-tuples. We also ensure that AS preferences are enforced when multiple candidate intersections pass the 3-tuple check. Figure 5 shows that the modified path composition technique increases iPlane's ability to predict AS paths from 70% to 81%.

The ability to predict paths using either *iNano* or path composition is limited by two factors, the comprehensiveness of the atlas measured from our vantage points and the accuracy of our inferred routing policies. We quantified the contribution of the former to the inaccuracy in path predictions as follows. For each path in our validation set, we determined whether all the inter-cluster links on the path were present in the corresponding atlas used to predict the path. 7% of paths were such that at least one of the inter-cluster links along the path was not observed in the atlas used for prediction. Therefore, if we had better coverage of the Internet's topology with measurements from more vantage points, the accuracy of path prediction could increase to up to 77% using *iNano* and to up to 88% using path composition.

6.3.2 How accurately can *iNano* estimate path properties?

Next, we evaluate *iNano*'s ability to estimate latencies along paths to arbitrary end-hosts. For each of the paths used in our evaluation of path prediction accuracy, we compose *iNano*'s link latency estimates along the predicted forward and reverse paths to derive an estimate for the end-to-end latency. Figure 6 shows the error in *iNano*'s latency estimates. We derive latency estimates for the same paths using the path-composition technique of iPlane [30] and using Vivaldi [13], a popular network coordinate system. *iNano*'s median latency estimation error is 11ms, as compared to a median error of 20ms with Vivaldi. The path composition technique yields an even

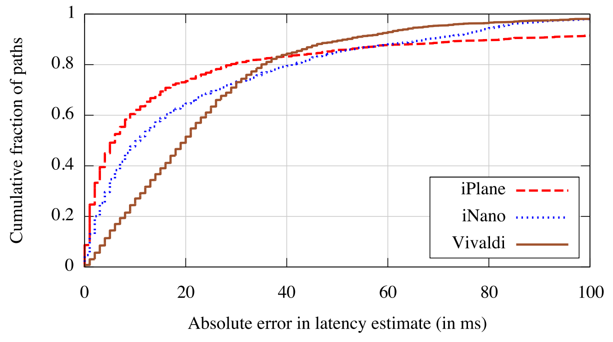


Figure 6: Accuracy of latency estimates along paths to arbitrary destinations.

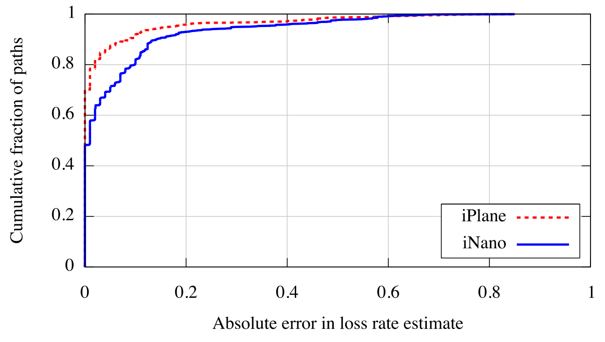


Figure 8: Accuracy of loss rate estimates along paths to arbitrary destinations.

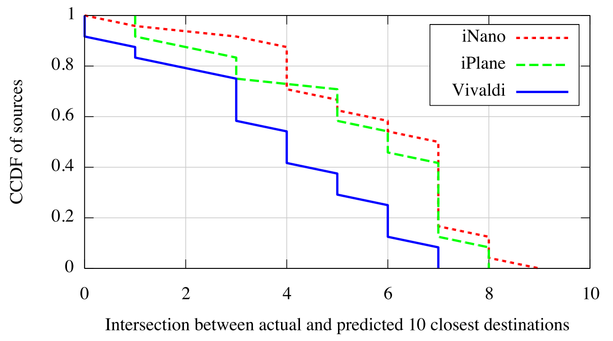


Figure 7: Accuracy of techniques in predicting 10 closest destinations (in terms of delay).

lower median error of 6ms, partly because of its better accuracy at predicting paths and partly because estimates of latencies along path segments tend to be more accurate than the sum of individual links.

However, the order of the three lines is reversed in the tail. *iNano* yields better latency estimates than the path composition technique in the tail because of differences in the methodology used to obtain link latencies for the former and path segment latencies for the latter. Our techniques for inferring link latencies identify and use measurements obtained by symmetric traversal of links [28], whereas our latency estimates of path segments do not. Like in iPlane [30], our latency estimates for path segments are obtained by just subtracting RTTs measured in traceroutes. The fact that Vivaldi produces better latency estimates in the tail than both *iNano* and path composition shows the significant room for improvement in our latency estimates for both links and path segments.

Applications such as peer selection and detour routing benefit from the ability to discern which destinations have low latency from a source. We therefore also assess latency estimation from the perspective of ranking different destinations in terms of latency from a common source. To quantify each technique's predictive ability on this criterion, we use the following metric. From each source, we determine the 10 closest nodes in terms of actual measured RTT among the 100 destinations per source in our validation set. We then do the same using estimated latencies and compute the intersection between the actual and predicted sets of 10 closest nodes. Figure 7 plots the cardinality of this intersection for each source in our validation set. *iNano*'s ability to rank paths is significantly better than that of Vivaldi, while being comparable to the path-based approach.

We next consider how well *iNano* can predict loss rates. We measured the loss rates along each of our validation paths and also measured the loss rate of each intercluster link in our atlas. We then use *iNano* to estimate the loss rate by composing the loss rates of the links along the predicted forward and reverse paths. Figure 8 plots the accuracy of *iNano*'s loss rate estimates. Since coordinate systems, such as Vivaldi, can only estimate latency, we restrict our comparison to iPlane's path composition technique in the case of loss rate. *iNano* approximates path-based estimates with a much smaller atlas.

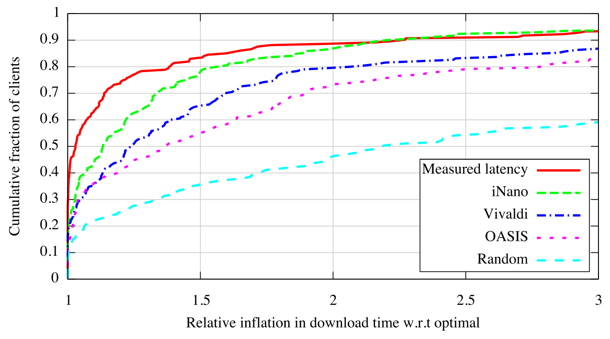
7 Applications

Our motivation in building *iNano* is to provide information on Internet paths to peer-to-peer applications. Therefore, we investigate the utility of the *iNano* library by using it in three sample peer-to-peer applications—peer-to-peer file transfer, voice-over-IP, and detour routing around failures.

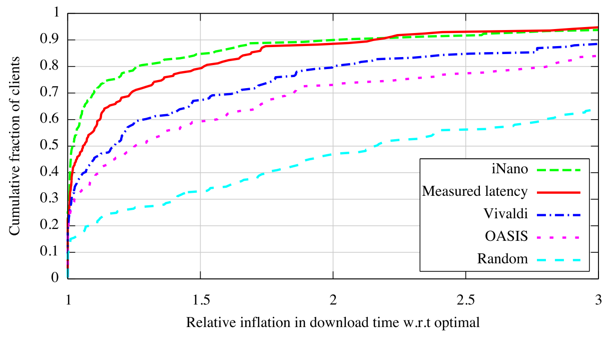
7.1 P2P file transfer

The next generation of content distribution networks (CDNs) are moving away from server-based deployments to client-based models. In contrast to services like Akamai [6], several alternatives [45, 38, 25] have recently emerged that perform content delivery by utilizing client end-hosts for storage and bandwidth. In such client-based CDNs, which are not centrally managed, a common problem is to determine the best replica for a given client. *iNano* enables clients to make this decision locally.

To evaluate the utility of *iNano* in client-based content-delivery systems, we emulated such a system as follows. We considered 199 PlanetLab nodes as clients. We re-



(a)



(b)

Figure 9: Evaluation of peer selection in a peer-to-peer file transfer system for file sizes of (a) 30KB and (b) 1.5MB. Each point is a median of 10 samples, with each sample obtained with a different randomly selected set of replicas.

solved an Akamai-zed DNS name from these nodes to discover 199 Akamai servers. For each client, we then determined the set of replicas that host the content of its interest by choosing 5 Akamai servers at random [3], independently for every client. We then determined the best replica for every client using four different sources of path information—1) measured latencies, 2) latency estimates from Vivaldi [13], 3) latency estimates from OASIS [18], a server-selection system used by many CDNs deployed on PlanetLab, and 4) latency and loss rate estimates from *iNano*. We also consider the strategy of choosing replicas at random. We evaluated each strategy by downloading from every client a file from each replica. We compare the download times for each strategy with the optimal, which is the minimum of the download times from the 5 replicas associated with the client.

Figure 9 shows the results of this experiment. First, we downloaded a 30KB file wherein we only used *iNano*'s estimates of path latency, because short TCP transfers are dominated by latency [8]. *iNano* closely tracks the performance obtained with measured latencies and is significantly better than the performance obtained with the use of Vivaldi or OASIS. We then repeated this experiment for a 1.5MB file. In this setting, we use *iNano*'s latency and loss rate estimates in combination to choose the replica that would maximize TCP throughput based on the PFTK model [37]. *iNano*'s predictions of loss rates enable it to choose replicas that deliver significantly better download performance than that obtained using measured latencies. Vivaldi and OASIS, restricted to modeling path latency, continue to yield poorer performance.

[3]We used such a setup instead of using PlanetLab nodes as replicas because the locations of PlanetLab nodes are hard-coded into OASIS.

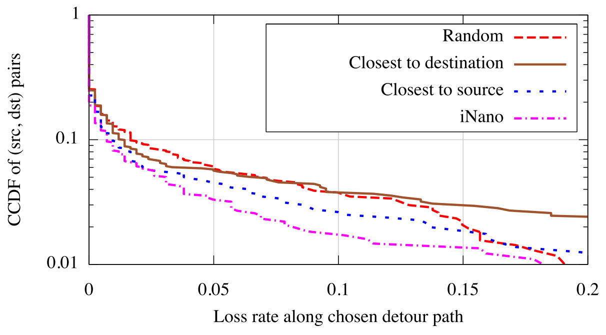


Figure 10: Evaluation of relay selection for voice-over-IP using *iNano*'s estimates of latency and loss rate.

Unlike our experimental evaluation, in practice, a P2P CDN may perform a transfer in parallel across multiple paths assuming that at least one of those paths will provide good performance. *iNano* can be of benefit to such applications in two ways. First, in applications that transmit video, *iNano* can reduce the bootstrapping time for the video to load by helping prune down a potentially large set of path alternatives to a small set of good paths used for the transfer, without performing any measurements. Second, by enabling the application to focus in on the good paths quickly, *iNano* reduces the redundant traffic sent by the application that either gets dropped on lossy paths or is used just for measurement.

7.2 Voice-over-IP

Voice-over-IP (VoIP) has emerged as a popular peer-to-peer application in recent years. VoIP applications such as Skype [52] allow end-hosts that are both behind NATs to talk to each other by routing packets via another end-host that serves as a relay. Picking the right relay is vital to ensure reasonable quality of the end-to-end call [46].

We emulated a VoIP application by considering 119 PlanetLab nodes as representative end-hosts. We chose 1200 (source, destination) pairs at random and emulated a VoIP call between each such pair by sending a 10KBps constant bitrate UDP packet stream from the source to the destination. For each call, we consider all end-hosts other than the source and destination to be potential relays. We use *iNano* to pick the 10 relays that minimize the predicted loss rate and then choose the one amongst these that minimizes end-to-end latency. We compare this strategy of choosing relays with three other strategies—1) closest to source based on measured latency, 2) closest

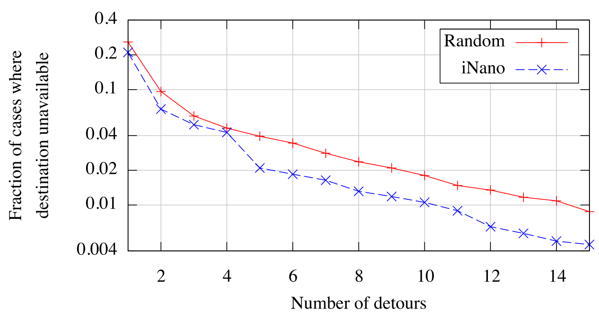


Figure 11: Ability to route around failures using *iNano*'s path predictions and using detour nodes at random. Note y axis is on log scale to the base 2.

to destination based on measured latency, and 3) random.

Figure 10 compares the quality of the relay nodes chosen by using *iNano*'s estimates of latency and loss rate with the choices made using the other strategies. Paths via relay nodes chosen by *iNano* see significantly less packet loss compared to the alternatives.

7.3 Detouring around failures

Several Internet measurement studies [40, 60, 15, 20] have shown that the typical availability of an Internet path is "two-nines", i.e., 99%. This level of availability falls well short of that measured for the telephone network [26]. One of the solutions proposed to mitigate this problem is detour routing [48]. When a source is unable to reach a destination, the source can attempt to contact the destination instead by routing its packets via another end-host that serves as a detour. Previous solutions for improving availability with detour routing implement one of three approaches—1) constantly monitor paths between all pairs of end-hosts [7], 2) constantly monitor paths between all pairs of detour nodes [1] and have end-hosts route through nearby detour nodes, or 3) detour via a small randomly chosen set of end-hosts [20]. All-pairs monitoring is infeasible at Internet-scale, monitoring paths only between detour nodes ignores failures on paths from end-hosts to nearby detour nodes, and a small randomly chosen set of detours will not suffice for widespread outages.

We explore a new way of routing around failures by choosing detour nodes that maximize the disjointness between the detour path and the direct path. When a source is unable to reach a destination, we use *iNano* to predict the direct path from the source to the destination as well as the detour path via each of the available intermediaries. We then rank the detour paths based on the number of PoPs and ASes shared by their predicted paths. We choose the $(k+1)^{th}$ detour node in this ranking to be the one that minimizes first the number of PoPs and second the number of ASes in common with the direct path and the k previously chosen detours. A strategy for recovering from failures by using N detours would try the first N detours in the ranking; the lower the value of N the less overhead incurred.

To compare the efficacy of the above strategy for routing around failures with SOSR's [20] strategy of using a few detours at random, we gathered the following measurements of path availability. We used 35 PlanetLab nodes and performed traceroutes continually for a week from each of them to destinations in 1000 randomly chosen prefixes, once every 15 minutes. Whenever a PlanetLab node was unable to reach a destination, we measured the availability of the detour path via the other 34 PlanetLab nodes. We consider for our analysis only the cases when at least 10% of our sources were simultaneously unable to reach the destination but at least 10% could.

Figure 11 compares our ability to route around failures by intelligently choosing detours using *iNano*'s path predictions as opposed to choosing detours at random. For the same number of detour paths, using *iNano* reduces the fraction of cases when the destination is unreachable by roughly a factor of 2. For example, the use of 5 detour paths leaves the destinations unreachable in 2% of cases compared to 4% of cases with the random strategy.

8 Related Work

Our work benefits from a decade of work in Internet performance prediction [49, 17] and network measurement [51, 55]. Compared to most prior work, our goal is different: accurate prediction of sophisticated Internet performance metrics from lightweight end-hosts, which requires us to aggressively explore the trade-off between accuracy and representation size.

8.1 Latency prediction

IDMaps [17] pioneered the idea of a network information service that provides latency information between arbitrary end-hosts on the Internet. IDMaps issues pings from a set of vantage points to all participating end-hosts and also measures latencies between all pairs of vantage points. As more vantage points are added, the size of IDMaps' measurement data grows proportional to the square of the number of vantage points. Therefore, IDMaps uses a spanner-graph representation to compress its data. *iNano* tackles a different compression problem, that of compactly representing information encoded in the forwarding tables of all routers in the Internet.

Ng et al. [35] showed that Internet nodes could be embedded in a Euclidean coordinate space. The strength of the approach is that it is 1) simple because it treats the underlying network as a blackbox, and 2) lightweight because only a few bytes of coordinates per node need be stored. A large body of work has since refined this basic approach to provide decentralization [36, 13], improved computational efficiency [56], resilience to measurement error [12, 13], security [12], and accuracy. The techniques used to minimize error include Simplex minimization [12, 36], Principal Component Analysis (PCA) [27, 56], and spring relaxation [13].

The network coordinates approach poses two problems for our goals. First, the approach has been shown capable of predicting latencies, but it is unclear how to adapt the approach to other metrics that do not obey linear composition, such as loss rate. Second, the approach is fundamentally limited in accuracy. For example, about half of all Internet routes are known to be asymmetric [40] and a significant fraction are known to possess shorter detour routes [48]. However, common embedding techniques based on metric spaces will predict symmetric latencies and fail to predict detour routes when triangle inequality is violated. This limits the applicability of the approach for many applications.

8.2 Prediction of multiple metrics

Sequoia [43] attempts to embed nodes on to a "virtual prediction tree". Edges of the tree are annotated with latency and the latency between two nodes is predicted as the length of the path connecting them. Unlike other coordinate systems, Sequoia is also extensible to bandwidth. However, it continues to use metric embeddings that predict symmetric routes with no detour routes. Akamai's SureRoute [1] service optimizes transfers between end-hosts by routing through a mesh of detour nodes. End-hosts are routed through nearby detour nodes and the optimal path through the mesh of detour nodes is determined by constant monitoring. However, the performance along a path between two end-hosts is not necessarily the same as on the path via their nearby detour nodes.

8.3 Structural inference

iNano's structural inference approach has been previously used in iPlane. However, unlike *iNano*, iPlane adopts a centralized architecture that scales poorly to 1) Internet-scale query loads, and 2) more vantage points. iPlane uses an atlas of observed paths, whose size is proportional to the number of vantage points times the number of destinations probed times the average path length. With iPlane's current set of vantage points and destinations, the size of its atlas is already over 1GB. As more vantage points contribute measurements, iPlane's accuracy will increase, but at the cost of blowing up the size of its atlas. iPlane's large atlas has the implication that its query engine can only be hosted on dedicated servers but not on typical end-hosts. *iNano*'s atlas instead comprises link-level, not path-level, information of the Internet structure. Routing policies encoded in iPlane's set of observed paths are replaced by *iNano*'s compact representation of the same.

8.4 AS path inference

iNano's main focus is on predicting path performance between arbitrary end-hosts, while predicting the path between them. Prior work has looked at a part of this problem, inference of AS paths.

Mao et al. [32] describe a structural inference approach, RouteScope, to infer AS-level paths. They use constrained optimization to model aspects of interdomain policy routing such as customer<peer<provider and valley-free routing, and use additional measurement techniques to observe routes from multihomed prefixes. Our evaluation in Section 6 shows that *iNano*'s ability to predict AS paths is significantly better than that of RouteScope, with *iNano* predicting the AS path correctly for more than twice as many paths in our validation set.

Qiu and Gao [42] build on RouteScope by using observed AS paths as constraints in predicting paths. Muhlbauer et al. [34] attempt to develop a hybrid model of Internet routing that lies in between a blackbox and a structure inference approach. They introduce "quasi-routers" to model the presence of multiple border routers in an AS based on an observed set of routes. Their approach can predict the training set exactly and achieves 50% prediction accuracy for unobserved routes. Both these pieces of work require a set of AS paths to make predictions; an atlas of paths is not compact enough to serve *iNano*'s goal of distributing the atlas to end-hosts.

9 Conclusions

Our contribution is a practical one. Today, there is a gap between research techniques for Internet performance prediction, and the scalability and low-overhead desired by large-scale P2P applications. *iPlane Nano* is a lightweight Internet path performance prediction engine that applications can use today at low cost. To make this work, we develop a model of Internet routing that can predict PoP-level paths between arbitrary end-hosts with an atlas that is less than 7MB in size and can be updated with roughly 1MB/day. The compact nature of the atlas enables applications to have their clients download the atlas and process queries locally. Furthermore, because the atlas is the same for all end-hosts, it can be disseminated to clients at low cost by using common P2P filesharing protocols, and thus largely using client bandwidths. Our evaluation of *iPlane Nano* demonstrated the accuracy of its predictions and its utility in improving the performance of P2P applications.

Acknowledgments

We thank the DIMES team for their help in gathering data used in our evaluation from the DIMES infrastructure. We would like to thank Jay Lorch and the anonymous NSDI reviewers for their valuable feedback on earlier versions of this paper. We also thank Venugopalan Ramasubramanian for serving as our shepherd. This research was partially supported by Cisco, Google, and by the National Science Foundation under Grants CNS-0435065 and CNS-0721779.

References

[1] Akamai SureRoute. www.akamai.com/dl/feature_sheets/fs_edgesuite_sureroute.pdf.

[2] PPLive. www.pplive.com.

[3] Sopcast. www.sopcast.com.

[4] World of Warcraft. www.blizzard.co.uk/wow/faq/bittorrent.shtml.

[5] Methods for subjective determination of transmission quality. ITU-T Recommendation P.800, 1996.

[6] Akamai, Inc. home page. http://www.akamai.com.

[7] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *SOSP*, 2001.

[8] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *INFOCOM*, 2000.

[9] D. R. Choffnes and F. E. Bustamante. Taming the torrent: A practical approach to reducing cross-ISP traffic in P2P systems. In *SIGCOMM*, 2008.

[10] Y. Chu, S. G. Rao, S. Seshan, and H. Zhang. A case for end system multicast. *IEEE JSAC*, 20(8):1456–1471, 2002.

[11] B. Cohen. Incentives build robustness in BitTorrent. In *P2PEcon*, 2003.

[12] M. Costa, M. Castro, A. Rowstron, and P. Key. PIC: Practical Internet coordinates for distance estimation. In *ICDCS*, 2004.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[14] F. Dabek, F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[15] M. Dahlin, B. Chandra, L. Gao, and A. Nayate. End-to-end WAN service availability. *IEEE/ACM ToN*, 2003.

[16] X. Dimitropoulos, D. Krioukov, M. Fomenkov, B. Huffaker, Y. Hyun, kc claffy, and G. Riley. AS relationships: inference and validation. *ACM CCR*, 2007.

[17] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A global Internet host distance estimation service. *IEEE/ACM ToN*, 2001.

[18] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for any service. In *NSDI*, 2006.

[19] L. Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM ToN*, 2001.

[20] K. P. Gummadi, H. V. Madhyastha, S. D. Gribble, H. M. Levy, and D. J. Wetherall. Improving the reliability of Internet paths with one-hop source routing. In *OSDI*, 2004.

[21] Y. He, M. Faloutsos, S. Krishnamurthy, and B. Huffaker. On routing asymmetry in the Internet. In *Autonomic Networks Symposium in Globecom*, 2005.

[22] N. Hu and P. Steenkiste. Quantifying Internet end-to-end route similarity. In *PAM*, 2006.

[23] E. Katz-Bassett, H. V. Madhyastha, J. P. John, A. Krishnamurthy, and T. Anderson. Studying black holes in the Internet with Hubble. In *NSDI*, 2008.

[24] R. Kokku, A. Bohra, S. Ganguly, and A. Venkataramani. A multipath background network architecture. In *INFOCOM*, 2007.

[25] Kontiki. www.kontiki.com/.

[26] D. R. Kuhn. Sources of failure in the public switched telephone networks. *IEEE Computer*, 1997.

[27] H. Lim, J. C. Hou, and C.-H. Choi. Constructing an Internet coordinate system based on delay measurement. In *IMC*, 2003.

[28] H. V. Madhyastha. *An Information Plane for Internet Applications*. PhD thesis, University of Washington, 2008.

[29] H. V. Madhyastha, T. Anderson, A. Krishnamurthy, N. Spring, and A. Venkataramani. A structural approach to latency prediction. In *IMC*, 2006.

[30] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *OSDI*, 2006.

[31] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. Inferring link weights using end-to-end measurements. In *IMW*, 2002.

[32] Z. M. Mao, L. Qiu, J. Wang, and Y. Zhang. On AS-level path inference. In *SIGMETRICS*, 2005.

[33] D. Meyer. RouteViews. www.routeviews.org.

[34] W. Mühlbauer, A. Feldmann, O. Maennel, M. Roughan, and S. Uhlig. Building an AS-topology model that captures route diversity. In *SIGCOMM*, 2006.

[35] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *INFOCOM*, 2002.

[36] T. S. E. Ng and H. Zhang. A network positioning system for the Internet. In *USENIX*, 2004.

[37] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *SIGCOMM*, 1998.

[38] Pando networks. www.pandonetworks.com/.

[39] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *NSDI*, 2006.

[40] V. Paxson. End-to-end routing behavior in the Internet. *IEEE/ACM ToN*, 1997.

[41] PlanetLab. http://www.planet-lab.org.

[42] J. Qiu and L. Gao. AS path inference by exploiting known AS paths. In *GLOBECOM*, 2006.

[43] V. Ramasubramanian, D. Malkhi, F. Kuhn, I. Abraham, M. Balakrishnan, A. Gupta, and A. Akella. A unified network coordinate system for bandwidth and latency. Technical Report MSR-TR-2008-124, Microsoft Research, 2008.

[44] V. Ramasubramanian and E. G. Sirer. The design and implementation of a next generation name service for the Internet. In *SIGCOMM*, 2004.

[45] Redswoosh. http://en.wikipedia.org/wiki/Red_Swoosh.

[46] S. Ren, L. Guo, and X. Zhang. ASAP: an AS-aware peer-relay protocol for high quality VoIP. In *ICDCS*, 2006.

[47] RIPE Routing Information Service. http://www.ripe.net/ris/.

[48] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: a case for informed Internet routing and transport. *IEEE Micro*, 19(1), 1999.

[49] S. Seshan, M. Stemm, and R. Katz. SPAND: Shared passive network performance discovery. In *USITS*, 1997.

[50] Y. Shavitt and E. Shir. DIMES: Let the Internet measure itself. *ACM CCR*, 2005.

[51] skitter. www.caida.org/tools/measurement/skitter/.

[52] Skype home page. http://www.skype.com.

[53] N. Spring, M. Dontcheva, M. Rodrig, and D. Wetherall. How to resolve IP aliases. Technical report, Univ. of Washington, 2004.

[54] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *SIGCOMM*, 2003.

[55] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with Rocketfuel. *IEEE/ACM ToN*, 2004.

[56] L. Tang and M. Crovella. Virtual landmarks for the Internet. In *IMC*, 2003.

[57] H. Xie, Y. R. Yang, A. Krishnamurthy, Y. Liu, and A. Silberschatz. P4P: Provider portal for (P2P) applications. In *SIGCOMM*, 2008.

[58] M. Zhang, J. Lai, A. Krishnamurthy, L. Peterson, and R. Wang. A transport layer approach for improving end-to-end performance and robustness using redundant paths. In *USENIX*, 2004.

[59] M. Zhang, C. Zhang, V. Pai, L. Peterson, and R. Wang. PlanetSeer: Internet path failure monitoring and characterization in wide-area services. In *OSDI*, 2004.

[60] Y. Zhang, V. Paxson, and S. Shenker. The stationarity of Internet path properties: Routing, loss, and throughput. Technical report, ACIRI, 2000.

Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults

Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin
The University of Texas at Austin

Mirco Marchetti
The University of Modena and Reggio Emilia

Abstract

This paper argues for a new approach to building Byzantine fault tolerant replication systems. We observe that although recently developed BFT state machine replication protocols are quite fast, they don't tolerate Byzantine faults very well: a single faulty client or server is capable of rendering PBFT, Q/U, HQ, and Zyzzyva virtually unusable. In this paper, we (1) demonstrate that existing protocols are dangerously fragile, (2) define a set of principles for constructing BFT services that remain useful even when Byzantine faults occur, and (3) apply these principles to construct a new protocol, Aardvark. Aardvark can achieve peak performance within 40% of that of the best existing protocol in our tests and provide a significant fraction of that performance when up to f servers and any number of clients are faulty. We observe useful throughputs between 11706 and 38667 requests per second for a broad range of injected faults.

1 Introduction

This paper is motivated by a simple observation: although recently developed BFT state machine replication protocols have driven the costs of BFT replication to remarkably low levels [1, 8, 12, 18], the reality is that they don't tolerate Byzantine faults very well. In fact, a single faulty client or server can render these systems effectively unusable by inflicting multiple orders of magnitude reductions in throughput and even long periods of complete unavailability. Performance degradations of such degree are at odds with what one would expect from a system that calls itself Byzantine fault tolerant—after all, if a single fault can render a system unavailable, can that system truly be said to tolerate failures?

To illustrate the the problem, Table 1 shows the measured performance of a variety of systems both in the absence of failures and when a single faulty client submits a carefully crafted series of requests. As we show later, a wide range of other behaviors—faulty primaries, recovering replicas, etc.—can have a similar impact. We believe that these collapses are byproducts of a single-minded focus on designing BFT protocols with ever more impressive best-case performance. While this focus is understandable—after years in which BFT replication was dismissed as too expensive to be practical, it was important to demonstrate that high-performance BFT is not an oxymoron—it has led to protocols whose complexity undermines robustness in two ways: (1) the protocols' *design* includes *fragile optimizations* that allow a faulty client or server to knock the system off of the optimized execution path to an expensive alternative path and (2) the protocols' *implementation* often fails to handle properly all of the intricate corner cases, so that the implementations are even more vulnerable than the protocols appear on paper.

The primary contribution of this paper is to advocate a new approach, *robust BFT* (RBFT), to building BFT systems. Our goal is to change the way BFT systems are designed and implemented by shifting the focus from constructing high-strung systems that maximize best case performance to constructing systems that offer acceptable and predictable performance under the broadest possible set of circumstances—including when faults occur.

System	Peak Throughput	Faulty Client
PBFT [8]	61710	0
Q/U [1]	23850	0[†]
HQ [12]	7629	N/A[‡]
Zyzzyva [18]	65999	0
Aardvark	38667	38667

Table 1: Observed peak throughput of BFT systems in a fault-free case and when a single faulty client submits a carefully crafted series of requests. We detail our measurements in Section 7.2. [†] The result reported for Q/U is for correct clients issuing conflicting requests. [‡] The HQ prototype demonstrates fault-free performance and does not implement many of the error-handling steps required to handle inconsistent MACs.

RBFT explicitly considers performance during both *gracious* intervals—when the network is synchronous, replicas are timely and fault-free, and clients correct—and *uncivil* execution intervals in which network links and correct servers are timely, but up to $f = \lfloor \frac{n-1}{3} \rfloor$ servers and any number of clients are faulty. The last row of Table 1 shows the performance of Aardvark, an RBFT state machine replication protocol whose design and implementation are guided by this new philosophy.

In some ways, Aardvark is very similar to traditional BFT protocols: clients send requests to a primary who relays requests to the replicas who agree (explicitly or implicitly) on the sequence of requests and the corresponding results—not unlike PBFT [8], High throughput BFT [19], Q/U [1], HQ [12], Zyzzyva [18], ZZ [32], Scrooge [28], etc.

In other ways, Aardvark is very different and challenges conventional wisdom. Aardvark utilizes signatures for authentication, even though, as Castro correctly observes, "eliminating signatures and using MACs instead eliminates the main performance bottleneck in previous systems" [7]. Aardvark performs regular view changes, even though view changes temporarily prevent the system from doing useful work. Aardvark utilizes point to point communication, even though renouncing IP-multicast gives up throughput deliberately.

We reach these counter-intuitive choices by following a simple and systematic approach: without ever compromising safety, we deliberately refocus both the design of the system and the engineering choices involved in its implementation on the stress that failures can impose on performance. In applying this strategy for RBFT to construct Aardvark, we choose an extreme position inspired by maxi-min strategies in game theory [26]: we reject any optimization for gracious executions that can decrease performance during uncivil executions.

Surprisingly, these counter-intuitive choices impose only a modest cost on its peak performance. As Table 1 illustrates, Aardvark sustains peak throughput of 38667 requests/second, which is within 40% of the best performance we measure on the same hardware for four stat-of-the-art protocols. At the same time, Aardvark's fault tolerance is dramatically improved. For a broad range of client, primary, and server misbehaviors we prove that Aardvark's performance remains within a constant factor of its best case performance. Testing of the prototype shows that these changes significantly improve robustness under a range of injected faults.

Once again, however, the main contribution of this paper is neither the Aardvark protocol nor implementation. It is instead a new approach that can—and we believe should—be applied to the design of other BFT protocols. In particular, we (1) demonstrate that existing protocols and their implementations are fragile, (2) argue that BFT protocols should be designed and implemented with a focus on robustness, and (3) use Aardvark to demonstrate that the RBFT approach is viable: we gain qualitatively better performance during uncivil intervals at only modest cost to performance during gracious intervals.

In Section 2 we describe our system model and the guarantees appropriate for high assurance systems. In Section 3 we elaborate on the need to rethink Byzantine fault tolerance and identify a set of design principles for RBFT systems. In Section 4 we present a systematic methodology for designing RBFT systems and an overview of Aardvark. In Section 5 we describe in detail the important components of the Aardvark protocol. In Section 6 we present an analysis of Aardvark's expected performance. In Section 7 we present our experimental evaluation. In Section 8 we discuss related work.

2 System model

We assume the Byzantine failure model where faulty nodes (servers or clients) can behave arbitrarily [21] and a strong adversary can coordinate faulty nodes to compromise the replicated service. We do, however, assume the adversary cannot break cryptographic techniques like collision-resistant hashing, message authentication codes (MACs), encryption, and signatures. We denote a message X signed by principal p's public key as $\langle X \rangle_{\sigma_p}$. We denote a message X with a MAC appropriate for principals p and r as $\langle X \rangle_{\mu_{r,p}}$. We denote a message containing a *MAC authenticator*—an array of MACs appropriate for verification by every replica—as $\langle X \rangle_{\vec{\mu}_r}$

Our model puts no restriction on clients, except that their number be finite: in particular, any number of clients can be arbitrarily faulty. However, the system's safety and liveness properties are guaranteed only if at most $f = \lfloor \frac{n-1}{3} \rfloor$ servers are faulty.

Finally, we assume an asynchronous network where *synchronous intervals*, during which messages are delivered with a bounded delay, occur infinitely often.

Definition 1 (Synchronous interval)**.** *During a synchronous interval any message sent between correct processes is delivered within a bounded delay T if the sender retransmits according to some schedule until it is delivered.*

3 Recasting the problem

The foundation of modern BFT state machine replication rests on an impossibility result and on two principles that assist us in dealing with it. The impossibility result, of course, is FLP [13], which states that no solution to consensus can be both safe and live in an asynchronous systems if nodes can fail. The two principles, first applied by Lamport to his Paxos protocol [20], are at the core of Castro and Liskov's seminal work on PBFT [7]. The first states that synchrony must not be needed for safety:

as long as a threshold of faulty servers is not exceeded, the replicated service must always produce linearizable executions, independent of whether the network loses, reorders, or arbitrarily delays messages. The second recognizes, given FLP, that synchrony must play a role in liveness: clients are guaranteed to receive replies to their requests only during intervals in which messages sent to correct nodes are received within some fixed (but potentially unknown) time interval from when they are sent.

Within these boundaries, the engineering of BFT protocols has embraced Lampson's well-known recommendation: "Handle normal and worst case separately as a rule because the requirements for the two are quite different. The normal case must be fast. The worst case must make some progress" [22]. Ever since PBFT, the design of BFT systems has then followed a predictable pattern: first, characterize what defines the normal (common) case; then, pull out all the stops to make the system perform well for that case. While different systems don't completely agree on what defines the common case [16], on one point they are unanimous: the common case includes only *gracious executions*, defined as follows:

Definition 2 (Gracious execution)**.** *An execution is gracious iff (a) the execution is synchronous with some implementation-dependent short bound on message delay and (b) all clients and servers behave correctly.*

The results of this approach continue to be spectacular. Since Zyzzyva last year reported a throughput of over 85,000 null requests per second [18], several new protocols have further improved on that mark [16, 28].

Despite these impressive results, we argue that a single minded focus on aggressively tuning BFT systems for the best case of gracious execution, a practice that we have engaged in with relish [18], is increasingly misguided, dangerous, and even futile.

It is misguided, because it encourages the design and implementation of systems that fail to deliver on their basic promise: to tolerate Byzantine faults. While providing impressive throughput during gracious executions, today's high-performance BFT systems are content to guaranteeing weak liveness guarantees (e.g. "eventual progress") in the presence of Byzantine failures. Unfortunately, as we previewed in Figure 1 and show in detail in Section 7.2, these guarantees are weak indeed. Although current BFT systems can *survive* Byzantine faults without compromising safety, we contend that a system that can be made completely unavailable by a simple Byzantine failure can hardly be said to *tolerate* Byzantine faults.

It is dangerous, because it encourages *fragile optimizations*. Fragile optimizations are harmful in two ways. First, as we will see in Section 7.2, they make it easier for a faulty client or server to knock the system off its hard-won optimized execution path and enter an alternative, much more expensive one. Second, they weigh down the system with subtle corner cases, increasing the likelihood of buggy or incomplete implementations.

It is (increasingly) futile, because the race to optimize common case performance has reached a point of diminishing return where many services' peak demands are already far under the best-case throughput offered by existing BFT replication protocols. For such systems, *good enough is good enough*, and further improvements in best case agreement throughput will have little effect on end-to-end system performance.

In our view, a BFT system fulfills its obligations when it provides acceptable and dependable performance across the broadest possible set of executions, including executions with Byzantine clients and servers. In particular, the temptation of fragile optimizations should be resisted: a BFT system should be designed around an execution path that has three properties: (1) it provides acceptable performance, (2) it is easy to implement, and (3) it is robust against Byzantine attempts to push the system away from it. Optimizations for the common case should be accepted only as long as they don't endanger these properties.

FLP tells us that we cannot guarantee liveness in an asynchronous environment. This is no excuse to cling to gracious executions only. In particular, there is no theoretical reason why BFT systems should not be expected to perform well in what we call *uncivil executions*:

Definition 3 (Uncivil execution)**.** *An execution is* u*ncivil iff (a) the execution is synchronous with some implementation-dependent short bound on message delay, (b) up to f servers and an arbitrary number of clients are Byzantine, and (c) all remaining clients and servers are correct.*

Hence, we propose to build RBFT systems that provide adequate performance during uncivil executions. Although we recognize that this approach is likely to reduce the best case performance, we believe that for a BFT system a limited reduction in peak throughput is preferable to the devastating loss of availability that we report in Figure 1 and Section 7.2.

Increased robustness may come at effectively no additional cost as long as a service's peak demand is below the throughput achievable through RBFT design: as a data point, our Aardvark prototype reaches a peak throughput of 38667 req/s.

Similarly, when systems have other bottlenecks, Amdahl's law limits the impact of changing the performance of agreement. For example, we report in Section 7 that PBFT can execute almost 62,000 null requests per second, suggesting that agreement consumes $16.1 \mu s$ per request. If, rather than a null service, we replicate a service for which executing an average request consumes $100\mu s$ of processing time, then peak throughput with PBFT settles to about 8613 requests per second. For the same service, a protocol with twice the agreement overhead of PBFT (i.e., $32.2\mu s$ per request), would still achieve peak throughput of about 7564 requests/second: in this hypothetical example, doubling agreement overhead would reduce peak end-to-end throughput by about 12%.

4 Aardvark: RBFT in action

Aardvark is a new BFT system designed and implemented to be robust to failures. The Aardvark protocol consists of 3 stages: client request transmission, replica agreement, and primary view change. This is the same basic structure of PBFT [8] and its direct descendants [4, 18, 19, 33, 32], but revisited with the goal of achieving an execution path that satisfies the properties outlined in the previous section: acceptable performance, ease of implementation, and robustness against Byzantine disruptions. To avoid the pitfalls of fragile optimizations, we focus at each stage of the protocol on how faulty nodes, by varying both the nature and the rate of their actions and omissions, can limit the ability of correct nodes to perform in a timely fashion what the protocol requires of them. This systematic methodology leads us to the three main design differences between Aardvark and previous BFT systems: (1) signed client requests, (2) resource isolation, and (3) regular view changes.

Signed client requests. Aardvark clients use digital signatures to authenticate their requests. Digital signatures provide non-repudiation and ensure that all correct replicas make identical decisions about the validity of each client request, eliminating a number of expensive and tricky corner cases found in existing protocols that make use of weaker (though faster) message authentication code (MAC) authenticators [7] to authenticate client requests. The difficulty with utilizing MAC authenticators is that they do not provide the non-repudiation property of digital signatures—one node validating a MAC authenticator does not guarantee that any other nodes will validate that same authenticator [2].

As we mentioned in the Introduction, digital signatures are generally seen as too expensive to use. Aardvark uses them only for client requests where it is possible to push the expensive act of generating the signature onto the client while leaving the servers with the less expensive verification operation. Primary-to-replica, replica-to-replica, and replica-to-client communication rely on MAC authenticators. The quorum-driven nature of server-initiated communication ensures that a single faulty replica is unable to force the system into undesirable execution paths.

Because of the additional costs associated with verifying signatures in place of MACs, Aardvark must guard against new denial-of-service attacks where the system receives a large numbers of requests with signatures that need to be verified. Our implementation limits the number of signature verifications a client can inflict on the system by (1) utilizing a hybrid MAC-signature construct to put a hard limit on the number of *faulty* signature verifications a client can inflict on the system and (2) forcing a client to complete one request before issuing the next.

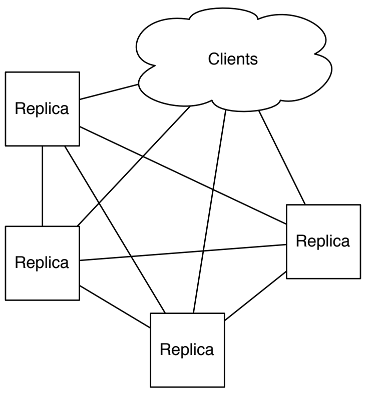


Figure 1: Physical network in Aardvark.

Resource isolation. The Aardvark prototype implementation explicitly isolates network and computational resources.

As illustrated by Fig. 1, Aardvark uses separate network interface controllers (NICs) and wires to connect each pair of replicas. This step prevents a faulty server from interfering with the timely delivery of messages from good servers, as happened when a single broken NIC shut down the immigration system at the Los Angeles International Airport [9]. It also allows a node to defend itself against brute force denial of service attacks by disabling the offending NIC. However, using physically separate NICs for communication between each pair of servers incurs a performance hit, as Aardvark can no longer use hardware multicast to optimize all-to-all communication.

As Figure 2 shows, Aardvark uses separate work queues for processing messages from clients and individual replicas. Employing a separate queue for client requests prevents client traffic from drowning out the replica-to-replica communications required for the system to make progress. Similarly, employing a separate queue for each replica allows Aardvark to schedule message processing fairly, ensuring that a replica is able to efficiently gather the quorums it needs to make progress. Aardvark can also easily leverage separate hardware threads to process incoming client and replica requests. Taking advantage of hardware parallelism allows Aardvark to reclaim part of the costs paid to verify signatures on client requests.

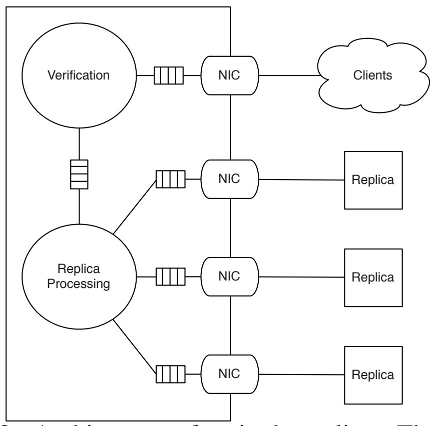


Figure 2: Architecture of a single replica. The replica utilizes a separate NIC for communicating with each other replica and a final NIC to communicate with the collection of clients. Messages from each NIC are placed on separate worker queues.

We use simple brute force techniques for resource scheduling. One could consider network-level scheduling techniques rather than distinct NICs in order to isolate network traffic and/or allow rate-limited multicast. Our goal is to make Aardvark as simple as possible, so we leave exploration of these techniques and optimizations for future work.

Regular view changes. To prevent a primary from achieving tenure and exerting absolute control on system throughput, Aardvark invokes the view change operation on a regular basis. Replicas monitor the performance of the current primary, slowly raising the level of minimal acceptable throughput. If the current primary fails to provide the required throughput, replicas initiate a view change.

The key properties of this technique are:

1. During uncivil intervals, system throughput remains high even when replicas are faulty. Since a primary maintains its position only if it achieves some increasing level of throughput, Aardvark bounds throughput degradation caused by a faulty primary by either forcing the primary to be fast or selecting a new primary.

2. As in prior systems, eventual progress is guaranteed when the system is eventually synchronous.

Previous systems have treated view change as an option of last resort that should only be used in desperate situations to avoid letting throughput drop to zero. However, although the phrase "view change" carries connotations of a complex and expensive protocol, in reality the cost of a view change is similar to the regular cost of agreement. Performing view changes regularly introduces short periods of time during which new requests are not being processed, but the benefits of evicting a misbehaving primary outweigh the periodic costs associated with performing view changes.

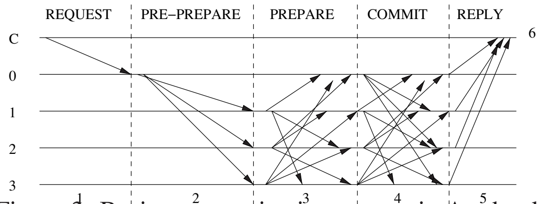


Figure 3: Basic communication pattern in Aardvark.

5 Protocol description

Figure 3 shows the agreement phase communication pattern that Aardvark shares with PBFT. Variants of this pattern are employed in other recent BFT RSM protocols [1, 12, 16, 18, 28, 32, 33], and we believe that, just as Aardvark illustrates how to adapt PBFT via RBFT system design, new Robust BFT systems based on these other protocols can and should be constructed. We organize the following discussion around the numbered steps of the communication pattern of Figure 3.

5.1 Client request transmission

The fundamental challenge in transmitting client requests is ensuring that, upon receiving a client request, every replica comes to the same conclusion about the authenticity of the request. We ensure this property by having clients sign requests.

To guard against denial of service, we break the processing of a client request into a sequence of increasingly expensive steps. Each step serves as a filter, so that more expensive steps are performed less often. For instance, we ask clients to include also a MAC on their signed requests and have replicas verify only the signature of those requests whose MAC checks out. Additionally, Aardvark explicitly dedicates a single NIC to handling incoming client requests so that incoming client traffic does not interfere with replica-to-replica communication.

5.1.1 Protocol Description

The steps taken by an Aardvark replica to authenticate a client request follow.

1. Client sends a request to a replica.

A client c requests an operation o be performed by the replicated state machine by sending a request message $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,p}}$ to the replica p it believes to be the primary. If the client does not receive a timely response to that request, then the client retransmits the request $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,r}}$ to all replicas r. Note that the request contains the client sequence number s and is signed with signature σ_c. The signed message is then authenticated with a MAC $\mu_{c,r}$ for the intended recipient.

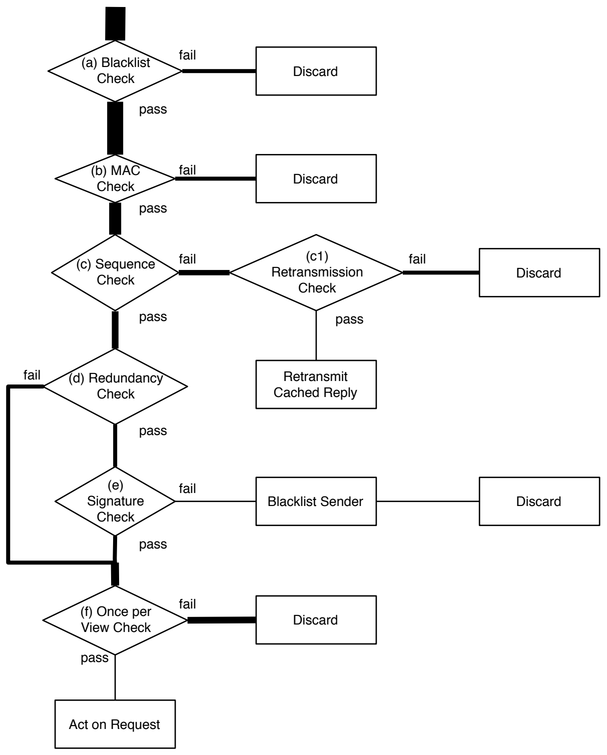


Figure 4: Decision tree followed by replicas while verifying a client request. The narrowing width of the edges portrays the devastating losses suffered by the army of client requests as it marches through the steppes of the verification process. Apologies to Minard.

Upon receiving a client request, a replica proceeds to verify it by following a sequence of steps designed to limit the maximum load a client can place on a server, as illustrated by Figure 4:

(a) **Blacklist check.** If the sender c is not blacklisted, then proceed to step (b). Otherwise discard the message.

(b) **MAC check.** If $\mu_{c,p}$ is valid, then proceed to step (c). Otherwise discard the message.

(c) **Sequence check.** Examine the most recent cached reply to c with sequence number s_{cache}. If the request sequence number s_{req} is exactly $s_{cache} + 1$, then proceed to step (d). Otherwise

(c1) **Retransmission check.** Each replica uses an exponential back off to limit the rate of client reply retransmissions. If a reply has not been sent to c recently, then retransmit the last reply sent to c. Otherwise discard the message.

(d) **Redundancy check.** Examine the most recent cached request from c. If no request from c with sequence number s_{req} has previously been verified or the request does not match the cached request, then proceed to step (e). Otherwise (the request matches the cached request from c) proceed to step (f).

(e) **Signature check.** If σ_c is valid, then proceed to step (f). Additionally, if the request does not match the previously cached request for s_{req}, then blacklist c. Otherwise if σ_c is not valid, then blacklist the node x that authenticated $\mu_{x,p}$ and discard the message.

(f) **Once per view check.** If an identical request has been verified in a previous view, but not processed during the current view, then act on the request. Otherwise discard the message.

Primary and non-primary replicas act on requests in different ways. A primary adds requests to a PRE-PREPARE message that is part of the three-phase commit protocol described in Section 5.2. A non-primary replica r processes a request by authenticating the signed request with a MAC $\mu_{r,p}$ for the primary p and sending the message to the primary. Note that non-primary replicas will forward each request at most once per view, but they may forward a request multiple times provided that a view change occurs between each occurrence.

Note that a REQUEST message that is verified as authentic might contain an operation that the replicated service that runs above Aardvark rejects because of an access control list (ACL) or other service-specific security violation. From the point of view of Aardvark, such messages are valid and will be executed by the service, perhaps resulting in an application level error code.

A node p only blacklists a sender c of a $\langle\langle \text{REQUEST}, o, s, c\rangle_{\sigma_c}, c\rangle_{\mu_{c,p}}$ message if the MAC $\mu_{c,p}$ is valid but the signature σ_c is not. A valid MAC is sufficient to ensure that routine message corruption is not the cause of the invalid signature sent by c, but rather that c has suffered a significant fault or is engaging in malicious behavior. A replica discards all messages it receives from a blacklisted sender and removes the sender from the blacklist after 10 minutes to allow reintegration of repaired machines.

5.1.2 Resource scheduling

Client requests are necessary to provide input to the RSM while replica-to-replica communication is necessary to process those requests. Aardvark leverages separate work queues for providing client requests and replica-to-replica communication to limit the fraction of replica resources that clients are able to consume, ensuring that a flood of client requests is unable to prevent replicas from making progress on requests already received. Of course, as in a non-BFT service, malicious clients can still deny service to other clients by flooding the network between clients and replicas. Defending against these attacks is an area of active independent research [23, 30].

We deploy our prototype implementation on dual core machines. As Figure 2 shows, one core verifies client requests and the second runs the replica protocol. This explicit assignment allows us to isolate resources and take advantage of parallelism to partially mask the additional costs of signature verification.

5.1.3 Discussion

RBFT aims at minimizing the costs that faulty clients can impose on replicas. As Figure 4 shows, there are four actions triggered by the transmission of a client request that can consume significant replica resources: MAC verification (MAC check), retransmission of a cached reply, signature verification (signature check), and request processing (act on request). The cost a faulty client can cause increases as the request passes each successive check in the verification process, but the rate at which a faulty client can trigger this cost decreases at each step.

Starting from the final step of the decision tree, the design ensures that the most expensive message a client can send is a correct request as specified by the protocol, and it limits the rate at which a faulty client can trigger expensive signature checks and request processing to the maximum rate a correct client would. The sequence check step (c) ensures that a client can trigger signature verification or request processing for a new sequence number only after its previous request has been successfully executed. The redundancy check (d) prevents repeated signature verifications for the same sequence number by caching each client's most recent request. Finally, the once per view check (f) permits repeated processing of a request only across different views to ensure progress. The signature check (e) ensures that only requests that will be accepted by all correct replicas are processed. The net result of this filtering is that, for every k correct requests submitted by a client, each replica performs at most $k + 1$ signature verifications, and any client that imposes a $k+1^{st}$ signature verification is blacklisted and unable to instigate additional signature verifications until it is removed from the blacklist.

Moving up the diagram, a replica responds to retransmission of completed requests paired with valid MACs by retransmitting the most recent reply sent to that client. The retransmission check (c1) imposes an exponential back off on retransmissions, limiting the rate at which clients can force the replica to retransmit a response. To help a client learn the sequence number it should use, a replica resends the cached reply at this limited rate for both requests that are from the past but also for requests that are too far into the future.

Any request that fails the MAC check (b) is immediately discarded. MAC verifications occur on every incoming message that claims to have the right format unless the sender is blacklisted, in which case the blacklist check (a) results in the message being discarded. The rate of MAC verification operations is thus limited by the rate at which messages purportedly from non-blacklisted clients are pulled off the network, and the fraction of processing wasted is at most the fraction of incoming requests from faulty clients.

5.2 Replica agreement

Once a request has been transmitted from the client to the current primary, the replicas must agree on the request's position in the global order of operations. Aardvark replicas coordinate with each other using a standard three phase commit protocol [8].

The fundamental challenge in the agreement phase is ensuring that each replica can quickly collect the quorums of PREPARE and COMMIT messages necessary to make progress. Conditioning expensive operations on the gathering of a quorum of messages makes it easier to ensure robustness in two ways. First, it is possible to design the protocol so that incorrect messages sent by a faulty replica will never gain the support of a quorum of replicas. Second, as long as there exists a quorum of timely correct replicas, a faulty replica that sends correct messages too slowly, or not at all, cannot impede progress. Faulty replicas can introduce overhead also by sending messages too quickly: to protect themselves, correct replicas in Aardvark schedule messages from other replicas in a round-robin fashion.

Not all expensive operations in Aardvark are triggered by a quorum. In particular, a correct replica that has fallen behind its peers may ask them for the state it is missing by sending them a *catchup message* (see Section 5.2.1). Aardvark replicas defer processing such messages to idle periods. Note that this state-transfer procedure is self-tuning: if the system is unable to make progress because it cannot assemble quorums of PREPARE and COMMIT messages, then it will devote more time to processing catchup messages.

5.2.1 Agreement protocol

The agreement protocol requires replica-to-replica communication. A replica r filters, classifies, and finally acts on the messages it receives from another replica according to the decision tree shown in Figure 5:

(a) **Volume Check.** If replica q is sending too many messages, blacklist q and discard the message. Otherwise continue to step (b). Aardvark replicas use a distinct NIC for communicating with each replica. Using per-replica NICs allows an Aardvark replica to silence replicas that flood the network and impose excessive interrupt processing load. In our prototype, we disable a network connection when q's rate of message transmission in the current view is a factor of 20 higher than for any other replica. After disconnecting q for flooding, r reconnects q after 10 minutes, or when f other replicas are disconnected for flooding.

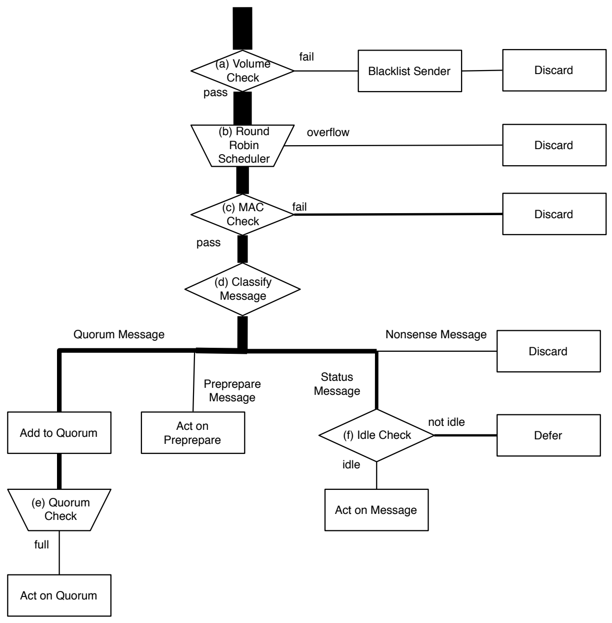


Figure 5: Decision tree followed by a replica when handling messages received from another replica. The width of the edges indicates the rate at which messages reach various stages in the processing.

(b) **Round-Robin Scheduler.** Among the pending messages, select the the next message to process from the available messages in round-robin order based on the sending replica . Discard received messages when the buffers are full.

(c) **MAC Check.** If the selected message has a valid MAC, then proceed to step (d) otherwise, discard the message.

(d) **Classify Message.** Classify the authenticated message according to its type:
- If the message is PRE-PREPARE, then process it immediately in protocol step 3 below.
- If the message is PREPARE or COMMIT, then add it to the appropriate quorum and proceed to step (e).
- If the message is a catchup message, then proceed to step (f).
- If the message is anything else, then discard the message.

(e) **Quorum Check.** If the quorum to which the message was added is complete, then act as appropriate in protocol steps 4-6 below.

(f) **Idle Check.** If the system has free cycles, then process the catchup message. Otherwise, defer processing until the system is idle.

Replica r applies the above steps to each message it receives from the network. Once messages are appropriately filtered and classified, the agreement protocol continues from step 2 of the communication pattern in Figure 3.

2. Primary forms a PRE-PREPARE message containing a set of valid requests and sends the PRE-PREPARE to all replicas.

The primary creates and transmits a $\langle$PRE-PREPARE, v, n, $\langle$REQUEST$, o, s, c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ message where v is the current view number, n is the sequence number for the PRE-PREPARE, and the authenticator is valid for all replicas. Although we show a single request as part of the PRE-PREPARE message, multiple requests can be batched in a single PRE-PREPARE [8, 14, 18, 19].

3. Replica receives PRE-PREPARE from the primary, authenticates the PRE-PREPARE, and sends a PREPARE to all other replicas.

Upon receipt of $\langle$PRE-PREPARE, v, n, $\langle$REQUEST$, o, s, c\rangle_{\sigma_c}\rangle_{\vec{\mu}_p}$ from primary p, replica r verifies the message's authenticity following a process similar to the one described in Section 5.1 for verifying requests. If r has already accepted the PRE-PREPARE message, r discards the message preemptively. If r has already processed a different PRE-PREPARE message with $n' = n$ during view v, then r discards the message. If r has not yet processed a PRE-PREPARE message for n during view v, r first checks that the appropriate portion of the MAC authenticator $\vec{\mu}_p$ is valid. If the replica has not already done so, it then checks the validity of σ_c. If the authenticator is not valid r discards the message. If the authenticator is valid and the client signature is invalid, then the replica blacklists the primary and requests a view change. If, on the other hand, the authenticator and signature are both valid, then the replica logs the PRE-PREPARE message and forms a $\langle$PREPARE, v, n, h,$r\rangle_{\vec{\mu}_r}$ to be sent to all other replicas where h is the digest of the set of requests contained in the PRE-PREPARE message.

4. Replica receives $2f$ PREPARE messages that are consistent with the PRE-PREPARE message for sequence number n and sends a COMMIT message to all other replicas.

Following receipt of $2f$ matching PREPARE messages from non-primary replicas r' that are consistent with a PRE-PREPARE from primary p, replica r sends a $\langle$COMMIT,$v, n, r\rangle_{\vec{\mu}_r}$ message to all replicas. Note that the PRE-PREPARE message from the primary is the $2f + 1^{\text{st}}$ message in the PREPARE quorum.

5. Replica receives $2f + 1$ COMMIT messages, commits and executes the request, and sends a REPLY message to the client.

After receipt of $2f+1$ matching $\langle \text{COMMIT}, v, n, r' \rangle_{\vec{\mu}_{r'}}$ from distinct replicas r', replica r commits and executes the request before sending $\langle \text{REPLY}, v, u, r \rangle_{\mu_{r,c}}$ to client c where u is the result of executing the request and v is the current view.

6. The client receives $f + 1$ matching REPLY messages and accepts the request as complete.

We also support Castro's tentative execution optimization [8], but we omit these details here for simplicity. They do not introduce any new issues for our RBFT design and analysis.

Catchup messages. State catchup messages are not an intrinsic part of the agreement protocol, but fulfill an important logistical priority of bringing replicas that have fallen behind back up to speed. If replica r receives a catchup message from a replica q that has fallen behind, then r sends q the state that q to catch up and resume normal operations. Sending catchup messages is vital to allow temporarily slow replicas to avoid becoming permanently non-responsive, but it also offers faulty replicas the chance to impose significant load on their non-faulty counterparts. Aardvark explicitly delays the processing of catchup messages until there are idle cycles available at a replica—as long as the system is making progress, processing a high volume of requests, there is no need to spend time bringing a slow replica up to speed!

5.2.2 Discussion

We now discuss the Aardvark agreement protocol through the lens of RBFT, starting from the bottom of Figure 5. Because every quorum contains at least a majority of correct replicas, faulty replicas can only marginally alter the rate at which correct replicas take actions (e) that require a quorum of messages. Further, because a correct replica processes catchup messages (f) only when otherwise idle, faulty replicas cannot use catchup messages to interfere with the processing of other messages. When client requests are pending, catchup messages are processed only if too many correct replicas have fallen behind and the processing of quorum messages needed for agreement has stalled—and only until enough correct replicas to enable progress have caught up. Also note that the queue of pending catchup messages is finite, and a replica discards excess catchup messages.

A replica processes PRE-PREPARE messages at the rate they are sent by the primary. If a faulty primary sends them too slowly or too quickly, throughput may be reduced, hastening the transition to a new primary as described in Section 5.3.

Finally, a faulty replica could simply bombard its correct peers with a high volume of messages that are eventually discarded. The round-robin scheduler (b) limits the damage that can result from this attack: if c of its peers have pending messages, then a correct replica wastes at most $\frac{1}{c}$ of the cycles spent checking MACs and classifying messages on what it receives from any faulty replica. The round-robin scheduler also discards messages that overflow a bounded buffer, and the volume check (a) similarly limits the rate at which a faulty replica can inject messages that the round-robin scheduler will eventually discard.

5.3 Primary view changes

Employing a primary to order requests enables batching [8, 14] and avoids the need to trust clients to obey a back off protocol [1, 10]. However, because the primary is responsible for selecting which requests to execute, the system throughput is at most the throughput of the primary. The primary is thus in a unique position to control both overall system progress [3, 4] and fairness to individual clients.

The fundamental challenge to safeguarding performance against a faulty primary is that a wide range of primary behaviors can hurt performance. For example, the primary can delay processing requests, discard requests, corrupt clients' MAC authenticators, introduce gaps in the sequence number space, unfairly delay or drop some clients' requests but not others, etc.

Hence, rather than designing specific mechanism to defend against each of these threats, past BFT systems [8, 18] have relied on view changes to replace an unsatisfactory primary with a new, hopefully better, one. Past systems trigger view changes conservatively, only changing views when it becomes apparent that the current primary is unlikely to allow the system to make even minimal progress.

Aardvark uses the same view change mechanism described in PBFT [8]; in conjunction with the agreement protocol, view changes in PBFT are sufficient to ensure eventual progress. They are not, however, sufficient to ensure acceptable progress.

5.3.1 Adaptive throughput

Replicas monitor the throughput of the current primary. If a replica judges the primary's performance to be insufficient, then the replica initiates a view change. More specifically, replicas in Aardvark expect two things from the primary: a regular supply of PRE-PREPARE messages and high sustained throughput. Following the completion of a view change, each replica starts a heartbeat timer that is reset whenever the next valid PRE-PREPARE message is received. If a replica does not receive the next valid PRE-PREPARE message before the heartbeat timer expires, the replica initiates a view change. To ensure eventual progress, a correct replica doubles the heartbeat interval each time the timer expires. Once the timer is reset because a PRE-PREPARE message is received, the replica resets the heartbeat timer back to its initial value. The value of the heartbeat timer is application and environment specific: our implementation uses a heartbeat of 40ms, so that a system that tolerates f failures demands a minimum of 1 PRE-PREPARE every every $2^f \times 40$ms.

The periodic checkpoints that, at pre-determined intervals, correct replicas must take to bound their state offer convenient synchronization points to assess the throughput that the primary is able to deliver. If the observed throughput in the interval between two successive checkpoints falls below a specified threshold, initially 90% of the maximum throughput observed during the previous n views, the replica initiates a view change to replace the current primary. At each checkpoint interval following an initial grace period at the beginning of each view, 5s in our prototype, the required throughput is increased by a factor of 0.01. Continually raising the bar that the current primary must reach in order to stay in power guarantees that a view change will eventually be replaced, restarting the process with the next primary. Conversely, if the system workload changes, the required throughput adjusts over n views to reflect the performance that a correct primary can provide.

The combined effect of Aardvark's new expectations on the primary is that during the first 5s of a view the primary is required to provide throughput of at least 1 request per 40ms or face eviction. The throughput of any view that lasts longer than 5s is at least 90% of the maximum throughput observed during the previous n views.

5.3.2 Fairness

In addition to hurting overall system throughput, primary replicas can influence which requests are processed. A faulty primary could be unfair to a specific client (or set of clients) by neglecting to order requests from that client. To limit the magnitude of this threat, replicas track fairness of request ordering. When a replica receives from a client a request that it has not seen in a PRE-PREPARE message, it adds the message to its request queue and, before forwarding the request to the primary, it records the sequence number k of the most recent PRE-PREPARE received during the current view. The replica monitors future PRE-PREPARE messages for that request, and if it receives two PRE-PREPAREs for another client before receiving a PREPARE for client c, then it declares the current primary to be unfair and initiates a view change. This ensures that two clients issuing comparable workloads observe throughput values within a constant factor of each other.

5.3.3 Discussion

The adaptive view change and PRE-PREPARE heartbeats leave a faulty primary with two options: it can provide substandard service and be replaced promptly, or it can remain the primary for an extended period of time and provide service comparable to what a non-faulty primary would provide. A faulty primary that does not make any progress will be caught very quickly by the heartbeat timer and summarily replaced. To avoid being replaced, a faulty primary must issue a steady stream of PRE-PREPARE messages until it reaches a checkpoint interval, when it is going to be replaced until it has provided the required throughput. To do *just* what is needed to keep ahead of its reckoning for as long as possible, a faulty primary will be forced to to deliver 95% of the throughput expected from a correct primary.

Periodic view changes may appear to institutionalize overhead, but their cost is actually relatively small. Although the term *view change* evokes images of substantial restructuring, in reality a view change costs roughly as much as a single instance of agreement with respect to message/protocol complexity: when performed every 100+ requests, periodic view changes have marginal performance impact during gracious or uncivil intervals.

6 Analysis

In this section, we analyze the throughput characteristics of Aardvark when the number of client requests is large enough to saturate the system and a fraction g of those requests is correct. We show that Aardvark's throughput during long enough uncivil executions is within a constant factor of its throughput during gracious executions of the same length provided there are sufficient correct clients to saturate the servers.

For simplicity, we restrict our attention to an Aardvark implementation on a single-core machine with a processor speed of κ GHz. We consider only the computational costs of the cryptographic operations—verifying signatures, generating MACs, and verifying MACs, requiring θ, α, and α cycles, respectively. Since these operations occur only when a message is sent or received, and the cost of sending or receiving messages is small, we expect similar results when modeling network costs explicitly.

We begin by computing Aardvark's peak throughput during a gracious view, i.e. a view that occur during a gracious execution, in Theorem 1. We then show in Theorem 2 that during uncivil views, i.e. views that occur during uncivil executions, with a correct primary Aardvark's throughput is at least g times the throughput achieved during a gracious view; as long as the primary is correct faulty replicas are unable to adversely impact Aardvark's throughput. Finally, we show that the throughput of an uncivil execution is at least the fraction of correct replicas times g times the throughput achieved during a gracious view.

We begin in Theorem 1 by computing t_{peak}, Aardvark's peak throughput during a gracious view, i.e. a view that occurs during a gracious execution. We then show in Theorem 2 that during uncivil views in which the primary replica is correct, Aardvark's peak throughput is only reduced to $g \times t_{peak}$: in other words, ignoring low level network overheads faulty replicas are unable to curtail Aardvark's throughput when the primary is correct. Finally, we show in Theorem 3 that the throughput across all views of an uncivil execution is within a constant factor of $\frac{n-f}{n} \times g \times t_{peak}$.

Theorem 1. *Consider a gracious view during which the system is saturated, all requests come from correct clients, and the primary generates batches of requests of size b. Aardvark's throughput is then at least $\frac{\kappa}{\theta+\frac{(4n-2b-4)}{b}\alpha}$ operations per second.*

Proof. We examine the actions required by each server to process one batch of size b. For each request in the batch, every server verifies one signature. The primary also verifies one MAC per request. For each batch, the primary generates $n-1$ MACs to send the PRE-PREPARE and verifies $n-1$ MACs upon receipt of the PREPARE messages; replicas instead verify one MAC in the primary's PRE-PREPARE , generate $(n-1)$ MACs when they send the PREPARE messages, and verify $(n-2)$ MACs when they receive them. Finally, each server first sends and then receives $n-1$ COMMIT messages, for which it generates and verifies a total of $n-2$ MACs, and generates a final MAC for each request in the batch to authenticate the response to the client. The total computational load per request is thus $\theta + \frac{(4n+2b-4)}{b}\alpha$ at the primary, and $\theta + \frac{(4n+b-4)}{b}\alpha$ at a replica. The system's throughput at saturation during a sufficiently long view in a gracious interval is thus at least $\frac{\kappa}{\theta+\frac{(4n+2b-4)}{b}\alpha}$ requests/sec. □

Theorem 2. *Consider an uncivil view in which the primary is correct and at most f replicas are Byzantine. Suppose the system is saturated, but only a fraction of the requests received by the primary are correct. The throughput of Aardvark in this uncivil view is within a constant factor of its throughput in a gracious view in which the primary uses the same batch size.*

Proof. Let θ and α denote the cost of verifying, respectively, a signature and a MAC. We show that if g is the fraction of correct requests, the throughput during uncivil views with a correct primary approaches g of the gracious view's throughput as the ratio $\frac{\alpha}{\theta}$ tends to 0.

In an uncivil view, faulty clients may send unfaithful requests to every server. Before being able to form a batch of b correct requests, the primary may have to verify $\frac{b}{g}$ signatures and MACs, and correct replicas may verify $\frac{b}{g}$ signatures and an additional $(\frac{b}{g})(1-g)$ MACs. Because a correct server processes messages from other servers in round-robin order, it will process at most two messages from a faulty server per message that it would have processed had the server been correct. The total computational load per request is thus $\frac{1}{g}(\theta + \frac{b(1+g)+4g(n-1+f)}{b}\alpha)$ at the primary, and $\frac{1}{g}(\theta + \frac{b+4g(n-1+f)}{b}\alpha)$ at a replica. The system's throughput at saturation during a sufficiently long view in an uncivil interval with a correct primary thus is at least $\frac{g\kappa}{\theta+\frac{(b(1+g)+4g(n-1+f)}{b}\alpha}$ requests per second: as the ratio $\frac{\alpha}{\theta}$ tends to 0, the ratio between the uncivil and gracious throughput approaches g. □

Theorem 3. *For sufficiently long uncivil executions and for small f the throughput of Aardvark, when properly configured, is within a constant factor of its throughput in a gracious execution in which primary replicas use the same batch size.*

Proof. First consider the case in which all the uncivil views have correct primary replicas. Assume that in a properly configured Aardvark $t_{baseViewTimeout}$ is set so that during an uncivil interval, a view change to a correct primary completes within $t_{baseViewTimeout}$. Since a primary's view lasts at least $t_{gracePeriod}$, as the ratio $\frac{\alpha}{\theta}$ tends to 0, the ratio between the throughput during a gracious view and an uncivil interval approaches $g\frac{t_{gracePeriod}}{t_{baseViewTimeout}+t_{gracePeriod}}$

Now consider the general case. If the uncivil interval is long enough, at most $\frac{f}{n}$ of its views will have a Byzantine primary. Aardvark's heartbeat timer provides two guarantees. First, a Byzantine server that does not produce the throughput that is expected of a correct server will not last as primary for longer than a grace period. Second, a correct server is always retained as a primary for at least the length of a grace period. Furthermore, since the throughput expected of a primary at the beginning of a view is a constant fraction of the maximum throughput achieved by the primary replicas of the last n views, faulty primary replicas cannot arbitrarily lower the throughput expected of a new primary. Finally, since the view change timeout is reset after a view change that results in at least one request being executed in the new view, no view change attempt takes longer then $t_{maxViewTimeout} = 2^f t_{baseViewTimeout}$. It follows that, during a sufficiently long uncivil interval, the throughput will be within a factor of $\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}}\frac{n-f}{n}$ of that of Theorem 2, and, as $\frac{\alpha}{\theta}$ tends to 0, the ratio between

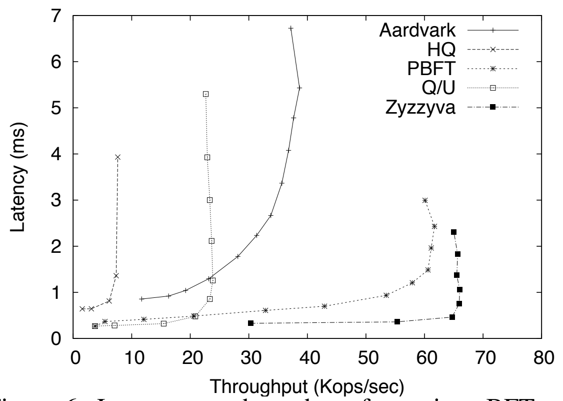


Figure 6: Latency vs. throughput for various BFT systems.

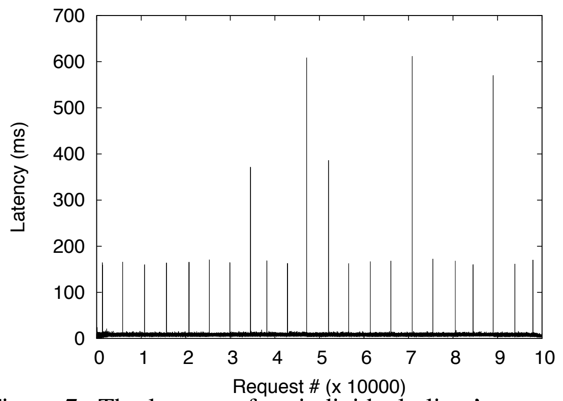


Figure 7: The latency of an individual client's requests running Aardvark with 210 total clients. The sporadic jumps represent view changes in the protocol.

the throughput during uncivil and gracious intervals approaches $g\frac{t_{gracePeriod}}{t_{maxViewTimeout}+t_{gracePeriod}}\frac{(n-f)}{n}$. □

7 Evaluation

We evaluate the performance of Aardvark, PBFT, HQ, Q/U and Zyzzyva on an Emulab cluster [31]. This cluster consists of machines with dual 3GHz Intel Pentium 4 Xeon processors, 1GB of memory, and 1 Gb/s Ethernet connections.

The code bases used to report our results are provided by the respective systems' authors. James Cowling provided us the December 2007 public release of the PBFT code base [5] as well as a copy of the HQ co-debase. We used version 1.3 of the Q/U co-debase, provided to us by Michael Abd-El-Malek in October 2008 [27]. The Zyzzyva co-debase is the version used in the SOSP 2007 paper [18]. Whenever feasible, we rely on the existing pre-configurations for each system to handle $f = 1$ Byzantine failure.

Our evaluation makes three points: (a) despite our choice to utilize signatures, change views regularly, and forsake IP multicast, Aardvark's peak throughput is competitive with that of existing systems; (b) existing systems are vulnerable to significant disruption as a result of a broad range of Byzantine behaviors; and (c) Aardvark is robust to a wide range of Byzantine behaviors. When evaluating existing systems, we attempt to identify places where the prototype *implementation* departs from the published *protocol*.

7.1 Aardvark

Aardvark's peak throughput is competitive with that of state of the art systems as shown in Figure 6. Aardvark's throughput peaks 38667 operations per second, while Zyzzyva and PBFT observe maximum throughputs of 65999 and 61710 operations per second, respectively.

Figures 7 and 8 explore the impact of regular view changes on the latency observed by Aardvark clients in

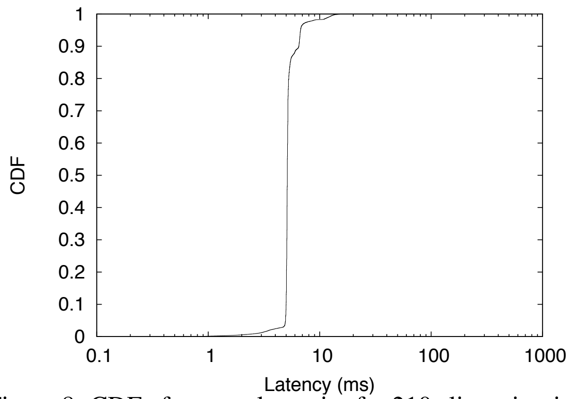


Figure 8: CDF of request latencies for 210 clients issuing 100,000 requests with Aardvark servers.

an experiment with 210 clients each issuing 100,000 requests. Figure 7 shows the per request latency observed by a single client during the run. The periodic latency spikes correspond to view changes. When a client issues a request as the view change is initiated, the request is not processed until the request arrives at the new primary following a client timeout and retransmission. In most cases a single client retransmission is sufficient, but additional retransmissions may be required when multiple view changes occur in rapid succession. Figure 8 shows the CDF for latencies of all client requests in the same experiment. We see that 99.99% of the requests have latency under 15ms, and only a small fraction of all requests incur the higher latencies induced by view changes. We configure an Aardvark client with a retransmission timeout of 150ms and we have not explored other settings.

System	Peak Throughput
Aardvark	38667
PBFT	61710
PBFT w/ client signatures	31777
Aardvark w/o signatures	57405
Aardvark w/o regular view changes	39771

Table 2: Peak throughput of Aardvark and incremental versions of the Aardvark protocol

7.1.1 Putting Aardvark together

Aardvark incorporates several key design decisions that enable it to perform well in the presence of Byzantine failure. We study the performance impact of these decisions by measuring the throughput of several PBFT and Aardvark variations, corresponding to the evolution between these two systems. Table 2 reports these peak throughputs.

While requiring clients in PBFT to sign requests reduces throughput by 50%, we find that the cost of requiring Aardvark clients to use the hybrid MAC-signature scheme imposes a smaller 33% hit to system throughput. Explicitly separating the work queues for client and replica communication makes it easy for Aardvark to utilize the second processor in our test bed machines, which reduces the additional costs Aardvark pays to verify signed client requests. This parallelism is the primary source of the 30% improvement we observe between PBFT with signatures and Aardvark.

Peak throughput for Aardvark with and without regular view changes is comparable. The reason for this is rather straightforward: when both the new and old primary replicas are non-faulty, a view change requires approximately the same amount of work as a single instance of consensus. Aardvark views led by a non-faulty primary are sufficiently long that the throughput costs associated with performing a view change are negligible.

7.2 Evaluating faulty systems

In this section we evaluate Aardvark and existing systems in the context of failures. It is impossible to test every possible Byzantine behavior; consequently we use our knowledge of the systems to construct a set of workloads that we believe to be close to the worst case for Aardvark and other systems. While other faulty behaviors are possible and may stress the evaluated systems in different ways, we believe that our results are indicative of both the frailty of existing systems and the robustness of Aardvark.

7.2.1 Faulty clients

We focus our attention on two aspects of client behavior that have significant impact on system throughput: request dissemination and network flooding.

Request dissemination. Table 1 in the Introduction explores the impact of faulty client behavior related to request distribution on the PBFT, HQ, Zyzzyva, and Aardvark prototypes. We implement different client behaviors for the different systems in order to stress test the design decisions the systems have made.

In PBFT and Zyzzvya, the clients send requests that are authenticated with MAC authenticators. The faulty client includes an inconsistent authenticator on requests so that request verification will succeed at the primary but fail for all other replicas. When the primary includes the client request in a PRE-PREPARE message, the replicas are unable to verify the request.

We developed this workload because, on paper, the protocols specify what appears to be an expensive processing path to handle this contingency. In this situation PBFT specifies a view change while Zyzzyva invokes a conflict resolution procedure that blocks progress and requires replicas to generate signatures. In theory these procedures should have a noticeable, though finite, impact on performance. In particular, PBFT progress should stall until a timeout forces a new view ([6] pp. 42–43), at which point other clients can make some progress until the faulty client stalls progress again. In Zyzzyva, the servers should pay extra overheads for signatures and view changes.

In practice the throughput of both prototype implementations drops to 0. In Zyzzyva the reconciliation protocol is not fully implemented; in PBFT the client behavior results in repeated view changes, and we have not observed our experiment to finish. While the full PBFT and Zyzzyva protocol specifications guarantee liveness under eventual synchrony, the protocol steps required to handle these cases are sufficiently complex to be difficult to implement, easy to overlook, or both.

In HQ, our intended attack is to have clients send certificates during the WRITE-2 phase of the protocol with an inconsistent MAC authenticator. The response specified by the protocol is a signed WRITE-2-REFUSED message which is subsequently used by the client to initiate a call to initiate a request processed by an internal PBFT protocol. This set of circumstances presents a point in the HQ design where a single client, either faulty or simply unlucky, can force the replicas to generate expensive signatures resulting in a degradation in system throughput. We are unable to evaluate the precise impact of this client behavior because the replica processing necessary to handle inconsistent MAC authenticators from clients is not implemented.

Q/U clients, in the lack of contention, are unable to influence each other's operations. During contention, replicas are required to perform barrier and commit operations that are rate limited by a client-initiated exponential back off. During the barrier and commit operations, a faulty client that sends inconsistent certificates to the replicas can theoretically complicate the process further. We implement a simpler scenario in which all clients are correct, yet they issue contending requests to the replicas. In this setting with only 20 clients, Q/U provides 0 throughput. Q/U's focus on performance in the absence of both failures and contention makes it especially vulnerable in practice—clients that issue contending requests can decimate system throughput, whether the clients are faulty or not.

To avoid corner cases where different replicas make different judgments about the legitimacy of a request, Aardvark clients sign requests. In Aardvark, the closest analogous client behaviors to those discussed above for other systems are sending requests with a valid MAC and invalid signature or sending requests with invalid MACs. We implement both attacks and find the results to be comparable. In Table 1 we report the results for requests with invalid MACs.

Network flooding. In Table 3 we demonstrate the impact of a single faulty client that floods the replicas with messages. During these experiments correct clients issue requests sufficient to saturate each system while a single faulty client implements a brute force denial of service attack by repeatedly sending 9KB UDP messages to the replicas. For PBFT and Zyzzyva, 210 clients are sufficient to saturate the servers while Q/U and HQ are saturated with 30 client processes.

The PBFT and Zyzzyva prototypes suffer dramatic performance degradation as their incoming network resources are consumed by the flooding client; processing the incoming client requests disrupt the replica-to-replica communication necessary for the systems to make progress. In both cases, the pending client requests eventually overflows internal queues and crashes the servers. Q/U and HQ suffer smaller degradations in throughput from the spamming replicas. The UDP traffic is dropped by the network stack with minimal processing because they are not valid TCP packets. The slowdowns observed in Q/U and HQ correspond to the displaced network bandwidth.

The reliance on TCP communication in Q/U and HQ changes rather than solves the challenge presented by a flooding client. For example, a single faulty client that repeatedly requests TCP connections crashes both the Q/U and HQ servers.

In each of these systems, the vulnerability to network flooding is a byproduct of the prototype implementation and is not fundamental to the protocol design. Network isolation techniques such as those described in Section 5 could similarly be applied to these systems.

In the case of Aardvark, the decision to use separate NICs and work queues for client and replica requests ensures that a faulty client is unable to prevent replicas from processing requests that have already entered the system. The throughput degradation observed by Aardvark tracks the fraction of requests that replicas receive that were sent by non-faulty clients.

System	Peak Throughput	Network Flooding	
		UDP	TCP
PBFT	61710	crash	-
Q/U	23850	23110	crash
HQ	7629	4470	0
Zyzzyva	65999	crash	-
Aardvark	38667	7873	-

Table 3: Observed peak throughput of BFT systems in the fault free case and under heavy client retransmission load. UDP network flooding corresponds to a single faulty client sending 9KB messages. TCP network flooding corresponds to a single faulty client sending requests to open TCP connections and is shown for TCP based systems.

System	Peak Throughput	1 ms	10 ms	100 ms
PBFT	61710	5041	4853	1097
Zyzzyva	65999	27776	5029	crash
Aardvark	38667	38542	37340	37903

Table 4: Throughput during intervals in which the primary delays sending PRE-PREPARE message (or equivalent) by 1, 10, and 100 ms.

7.2.2 Faulty Primary

In systems that rely on a primary, the primary controls the sequence of requests that are processed during the current view.

In Table 4 we show the impact on PBFT, Zyzzyva, and Aardvark prototypes of a primary that delays sending PRE-PREPARE messages by 1, 10, or 100 ms. The throughput of both PBFT and Zyzzyva degrades dramatically as the slow primary is not slow enough to trigger their view change conditions. This throughput degradation is a consequence of the protocol design and specification of when view changes should occur. With an extremely slow primary, Zyzzyva eventually succumbs to a memory leak exacerbated by holding on to requests for an extended period of time. The throughput achieved by Aardvark indicates that adaptively performing view changes in response to observed throughput is a good technique for ensuring performance.

In addition to controlling the rate at which requests are inserted into the system, the primary is also responsible for controlling which requests are inserted into the system. Table 5 explores the impact that an unfair primary can have on the throughput of a targeted node. In

System	Starved Throughput	Normal Throughput
PBFT	1.25	1446
Zyzzyva	0	1718
Aardvark	358	465

Table 5: Average throughput for a starved client that is shunned by a faulty primary versus the average per-client throughput for any other client.

the case of PBFT and Aardvark, the primary sends a PRE-PREPARE for the targeted client's request only after receiving the the request 9 times. This heuristic prevents the PBFT primary from triggering a view change and demonstrates dramatic degradation in throughput for the targeted client in comparison to the other clients in the system. For Zyzzyva, the unfair primary ignores messages from the targeted client entirely. The resulting throughput is 0 because the implementation is incomplete, and replicas in the Zyzzyva prototype do not forward received requests to the primary as specified by the protocol. Aardvark's fairness detection and periodic view changes limit the impact of the unfair primary.

7.2.3 Non-Primary Replicas

We implement a faulty replica that fails to process protocol messages and insted blasts network traffic at the other replicas and show the results in Table 6. In the first experiments, a faulty replica blasts 9KB UDP messages at the other replicas. The PBFT and Zyzzyva prototypes again show very low performance as the incoming traffic from the spamming replica displaces much of the legitimate traffic in the system, denying the system both requests from the clients and also replica messages required to make progress. Aardvark's use of separate worker queues ensures that the replicas process the messages necessary to make progress. In the second experiment, the faulty The Q/U and HQ replicas again open TCP connections, consuming all of the incoming connections on the other replicas and denying the clients access to the service.

Once again, the shortcomings of the systems are a byproduct of implementation and not protocol design. We speculate that improved network isolation techniques would make the systems more robust.

8 Related work

We are not the first to notice significantly reduced performance for BFT protocols during periods of failures or bad network performance or to explore how timing and failure assumptions impact performance and liveness of fault tolerant systems.

Singh et al. [29] show that PBFT [8], Q/U [1], HQ [12], and Zyzzyva [18] are all sensitive to network performance. They provide a thorough examination of the gracious executions of the four canonical systems through a ns2 [25] network simulator. Singh et al. explore performance properties when the participants are well behaved and the network is faulty; we focus our attention on the dual scenario where the participants are faulty and the network is well behaved.

System	Peak Throughput	Replica Flooding	
		UDP	TCP
PBFT	61710	251	-
Q/U	23850	19275	crash
HQ	7629	crash	crash
Zyzzyva	65999	0	-
Aardvark	38667	11706	-

Table 6: Observed peak throughput and observed throughput when one replica floods the network with messages. UDP flooding consists of a replica sending 9KB messages to other replicas rather than following the protocol. TCP flooding consists of a replica repeatedly attempting to open TCP connections on other replicas.

Aiyer et al. [3] and Amir et al. [4] note that a slow primary can result in dramatically reduced throughput. Aiyer et al. combat this problem by frequently rotating the primary. Amir et al. address the challenge instead by introducing a pre-agreement protocol requiring several all-to-all message exchanges and utilizing signatures for all authentication. Their solution is designed for environments where throughout of 800 requests per second is considered good. Condie et al. [11] address the ability of a well placed adversary to disrupt the performance of an overlay network by frequently restructuring the overlay, effectively changing its view.

The signature processing and scheduling of replica messages in Aardvark is similar in flavor to the early rejection techniques employed by the LOCKSS system [15, 24] in order to improve performance and limit the damage an adversary can inflict on system.

PBFT [8], Q/U [1], HQ [12], and Zyzzyva [18] are recent BFT replication protocols that focus on optimizing performance during gracious executions and collectively demonstrate that BFT replication systems can provide excellent performance during gracious executions. We instead focus on increasing the robustness of BFT systems by providing good performance during uncivil executions. Hendricks et al. [17] explore the use of erasure coding increase the efficiency of BFT replicated storage; they emphasizes increasing the bandwidth and storage efficiency of a replication protocol similar to Q/U and not the fault tolerance of the replication protocol.

9 Conclusion

We claim that high assurance systems require BFT protocols that are more robust to failures than existing systems. Specifically, BFT protocols suitable for high assurance systems must provide adequate throughput during uncivil intervals in which the network is well behaved but an unknown number of clients and up to f servers are faulty. We present Aardvark, the first BFT state machine protocol designed and implemented to provide good performance in the presence of Byzantine faults. Aardvark gives up some throughput during gracious executions, for significant improvement in performance during uncivil executions.

Aardvark is far from being the last word in robust BFT replication: we believe that improvements to the design and implementation of Aardvark, as well as to the methodology that led us to it, are both possible and likely. Specific challenges that remain for future work include formally verifying the design and implementations of BFT systems, developing a notion of optimality for robust BFT systems that captures the fundamental tradeoffs betwee fault-free and fault-full performance, and extending BFT replication to deployable large scale applications.

10 Acknowledgements

The authors would like to thank our shepherd, Petros Maniatis, for his detailed comments and the anonymous reviewers for their insightful reviews. This work was supported in part by NSF grants CSR-PDOS-0509338 and CSR-PDOS-0720649.

References

[1] ABD-EL-MALEK, M., GANGER, G., GOODSON, G., REITER, M., AND WYLIE, J. Fault-scalable Byzantine fault-tolerant services. In *SOSP* (2005).

[2] AIYER, A. S., ALVISI, L., BAZZI, R. A., AND CLEMENT, A. Matrix signatures: From macs to digital signatures in distributed systems. In *DISC* (2008).

[3] AIYER, A. S., ALVISI, L., CLEMENT, A., DAHLIN, M., MARTIN, J.-P., AND PORTH, C. BAR fault tolerance for cooperative services. In *SOSP* (Oct. 2005).

[4] AMIR, Y., COAN, B., KIRSCH, J., AND LANE, J. Byzantine replication under attack. In *DSN* (2008).

[5] BFT project homepage. `http://www.pmg.csail.mit.edu/bft/#sw`.

[6] CASTRO, M. *Practical Byzantine Fault Tolerance*. PhD thesis, 2001.

[7] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance. In *OSDI* (1999).

[8] CASTRO, M., AND LISKOV, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.* (2002).

[9] At LAX, computer glitch delays 20,000 passengers. http://travel.latimes.com/articles/la-trw-lax12aug12.

[10] CHOCKLER, G., MALKHI, D., AND REITER, M. Backoff protocols for distributed mutual exclusion and ordering. In *ICDCS* (2001).

[11] CONDIE, T., KACHOLIA, V., SANKARARAMAN, S., HELLERSTEIN, J. M., AND MANIATIS, P. Induced churn as shelter from routing-table poisoning. In *NDSS* (2006).

[12] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *OSDI* (2006).

[13] FISCHER, M., LYNCH, N., AND PATERSON, M. Impossibility of distributed consensus with one faulty process. *JACM* (1985).

[14] FRIEDMAN, R., AND RENESSE, R. V. Packing messages as a tool for boosting the performance of total ordering protocls. In *HPDC* (1997).

[15] GIULI, T. J., MANIATIS, P., BAKER, M., ROSENTHAL, D. S. H., AND ROUSSOPOULOS, M. Attrition defenses for a peer-to-peer digital preservation system. In *USENIX* (2005).

[16] GUERRAOUI, R., QUÉMA, V., AND VUKOLIC, M. The next 700 bft protocols. Tech. rep., Infoscience — Ecole Polytechnique Federale de Lausanne (Switzerland), 2008.

[17] HENDRICKS, J., GANGER, G. R., AND REITER, M. K. Low-overhead Byzantine fault-tolerant storage. In *SOSP* (2007).

[18] KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. Zyzzyva: speculative Byzantine fault tolerance. In *SOSP* (2007).

[19] KOTLA, R., AND DAHLIN, M. High throughput Byzantine fault tolerance. In *DSN* (June 2004).

[20] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst.*, 2 (1998).

[21] LAMPORT, L., SHOSTAK, R., AND PEASE, M. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.* (1982).

[22] LAMPSON, B. W. Hints for computer system design. *SIGOPS Oper. Syst. Rev. 17* (1983).

[23] MAHIMKAR, A., DANGE, J., SHMATIKOV, V., VIN, H., AND ZHANG, Y. dFence: Transparent network-based denial of service mitigation. In *NSDI* (2007).

[24] MANIATIS, P., ROUSSOPOULOS, M., GIULI, T. J., ROSENTHAL, D. S. H., AND BAKER, M. The LOCKSS peer-to-peer digital preservation system. *ACM Trans. Comput. Syst.* (2005).

[25] NS-2. `http://www.isi.edu/nsnam/ns/`.

[26] OSBORNE, M., AND RUBINSTEIN, A. *A Course in Game Theory*. MIT Press, 1994.

[27] Query/Update protocol. `http://www.pdl.cmu.edu/QU/index.html`.

[28] SERAFINI, M., BOKOR, P., AND SURI, N. Scrooge: Stable speculative byzantine fault tolerance using testifiers. Tech. rep., Darmstadt University of Technology, Department of Computer Science, September 2008.

[29] SING, A., DAS, T., MANIATIS, P., DRUSCHEL, P., AND ROSCOE, T. Bft protocols under fire. In *NSDI* (2008).

[30] WALFISH, M., VUTUKURU, M., BALAKRISHNAN, H., KARGER, D., AND SHENKER, S. DDoS defense by offense. In *SIGCOMM* (2006).

[31] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI* (2002).

[32] WOOD, T., SINGH, R., VENKATARAMANI, A., AND SHENOY, P. Zz: Cheap practical bft using virtualization. Tech. rep., University of Massachussets, 2008.

[33] YIN, J., MARTIN, J.-P., VENKATARAMANI, A., ALVISI, L., AND DAHLIN, M. Separating agreement from execution for Byzantine fault tolerant services. In *SOSP* (2003).

Zeno: Eventually Consistent Byzantine-Fault Tolerance

Atul Singh[1,2], Pedro Fonseca[1], Petr Kuznetsov[3], Rodrigo Rodrigues[1], Petros Maniatis[4]
[1]*MPI-SWS*, [2]*Rice University*,
[3]*TU Berlin/Deutsche Telekom Laboratories*, [4]*Intel Research Berkeley*

Abstract

Many distributed services are hosted at large, shared, geographically diverse data centers, and they use replication to achieve high availability despite the unreachability of an entire data center. Recent events show that non-crash faults occur in these services and may lead to long outages. While Byzantine-Fault Tolerance (BFT) could be used to withstand these faults, current BFT protocols can become unavailable if a small fraction of their replicas are unreachable. This is because existing BFT protocols favor strong safety guarantees (consistency) over liveness (availability).

This paper presents a novel BFT state machine replication protocol called Zeno that trades consistency for higher availability. In particular, Zeno replaces strong consistency (*linearizability*) with a weaker guarantee (*eventual consistency*): clients can temporarily miss each other's updates but when the network is stable the states from the individual partitions are merged by having the replicas agree on a total order for all requests. We have built a prototype of Zeno and our evaluation using micro-benchmarks shows that Zeno provides better availability than traditional BFT protocols.

1 Introduction

Data centers are becoming a crucial computing platform for large-scale Internet services and applications in a variety of fields. These applications are often designed as a composition of multiple services. For instance, Amazon's S3 storage service and its e-commerce platform use Dynamo [15] as a storage substrate, or Google's indices are built using the MapReduce [14] parallel processing framework, which in turn can use GFS [18] for storage.

Ensuring correct and continuous operation of these services is critical, since downtime can lead to loss of revenue, bad press, and customer anger [5]. Thus, to achieve high availability, these services replicate data and computation, commonly at multiple sites, to be able to withstand events that make an entire data center unreachable [15] such as network partitions, maintenance events, and physical disasters.

When designing replication protocols, assumptions have to be made about the types of faults the protocol is designed to tolerate. The main choice lies between a *crash-fault* model, where it is assumed nodes fail cleanly by becoming completely inoperable, or a *Byzantine-fault* model, where no assumptions are made about faulty components, capturing scenarios such as bugs that cause incorrect behavior or even malicious attacks. A crash-fault model is typically assumed in most widely deployed services today, including those described above; the primary motivation for this design choice is that all machines of such commercial services run in the trusted environment of the service provider's data center [15].

Unfortunately, the crash-fault assumption is not always valid even in trusted environments, and the consequences can be disastrous. To give a few recent examples, Amazon's S3 storage service suffered a multi-hour outage, caused by corruption in the internal state of a server that spread throughout the entire system [2]; also an outage in Google's App Engine was triggered by a bug in datastore servers that caused some requests to return errors [19]; and a multi-day outage at the Netflix DVD mail-rental was caused by a faulty hardware component that triggered a database corruption event [28].

Byzantine-fault-tolerant (BFT) replication protocols are an attractive solution for dealing with such faults. Recent research advances in this area have shown that BFT protocols can perform well in terms of throughput and latency [23], they can use a small number of replicas equal to their crash-fault counterparts [9, 37], and they can be used to replicate off-the-shelf, non-deterministic, or even distinct implementations of common services [29, 36].

However, most proposals for BFT protocols have focused on strong semantics such as linearizability [22], where intuitively the replicated system appears to the clients as a single, correct, sequential server. The price to pay for such strong semantics is that each operation must contact a large subset (more than $\frac{2}{3}$, or in some cases $\frac{4}{5}$) of the replicas to conclude, which can cause the system to halt if more than a small fraction ($\frac{1}{3}$ or $\frac{1}{5}$, respectively) of the replicas are unreachable due to maintenance events, network partitions, or other non-Byzantine faults. This contrasts with the philosophy of systems deployed in corporate data centers [15, 21, 34], which favor availability and performance, possibly sacrificing the semantics of the system, so they can provide continuous service and meet tight SLAs [15].

In this paper we propose Zeno, a new BFT replication protocol designed to meet the needs of modern services running in corporate data centers. In particular, Zeno favors service performance and availability, at the cost of providing weaker consistency guarantees than traditional

BFT replication when network partitions and other infrequent events reduce the availability of individual servers.

Zeno offers eventual consistency semantics [17], which intuitively means that different clients can be unaware of the effects of each other's operations, e.g., during a network partition, but operations are never lost and will eventually appear in a linear history of the service—corresponding to that abstraction of a single, correct, sequential server—once enough connectivity is re-established.

In building Zeno we did not start from scratch, but instead adapted Zyzzyva [23], a state-of-the-art BFT replication protocol, to provide high availability. Zyzzyva employs speculation to conclude operations fast and cheaply, yielding high service throughput during favorable system conditions—while connectivity and replicas are available—so it is a good candidate to adapt for our purposes. Adaptation was challenging for several reasons, such as dealing with the conflict between the client's need for a fast and meaningful response and the requirement that each request is brought to completion, or adapting the *view change* protocols to also enable progress when only a small fraction of the replicas are reachable and to merge the state of individual partitions when enough connectivity is re-established.

The rest of the paper is organized as follows. Section 2 motivates the need for eventual consistency. Section 3 defines the properties guaranteed by our protocol. Section 4 describe how Zeno works and Section 5 sketches the proof of its correctness. Section 6 evaluates how our implementation of Zeno performs. Section 7 presents related work, and Section 8 concludes.

2 The Case for Eventual Consistency

Various levels and definitions of weak consistency have been proposed by different communities [16], so we need to justify why our particular choice is adequate. We argue that eventual consistency is both *necessary* for the guarantees we are targetting, and *sufficient* from the standpoint of many applications.

Consider a scenario where a network partition occurs, that causes half of the replicas from a given replica group to be on one side of the partition and the other half on the other side. This is plausible given that replicated systems often spread their replicas over multiple data centers for increased reliability [15], and that Internet partitions do occur in practice [6]. In this case, eventual consistency is *necessary* to offer high availability to clients on both sides of the partition, since it is impossible to have both sides of the partitions make progress and simultaneously achieve a consistency level that provided a total order on the operations ("seen" by all client requests) [7]. Intuitively, the closest approximation from that idealized consistency that could be offered is eventual consistency, where clients on each side of the partition agree on an ordering (that only orders their operations with respect to each other), and, when enough connectivity is re-established, the two divergent states can be merged, meaning that a total order between the operations on both sides can be established, and subsequent operations will reflect that order.

Additionally, we argue that eventual consistency is *sufficient* from the standpoint of the properties required by many services and applications that run in data centers. This has been clearly stated by the designers of many of these services [3, 13, 15, 21, 34]. Applications that use an eventually consistent service have to be able to work with responses that may not include some previously executed operations. To give an example of applications that use Dynamo, this means that customers may not get the most up-to-date sales ranks, or may even see some items they deleted reappear in their shoping carts, in which case the delete operation may have to be redone. However, those events are much preferrable to having a slow, or unavailable service.

Beyond data-center applications, many other examples of eventually consistent services has been deployed in common-use systems, for example, DNS. Saito and Shapiro [30] provide a more thourough survey of the theme.

3 Algorithm Properties

We now informally specify safety and liveness properties of a generic eventually consistent BFT service. The formal definitions appear in a separate technical report due to lack of space [31].

3.1 Safety

Informally, our safety properties say that an eventually consistent system behaves like a centralized server whose service state can be modelled as a multi-set. Each element of the multi-set is a history (a totally ordered subset of the invoked operations), which captures the intuitive notion that some operations may have executed without being aware of each other, e.g., on different sides of a network partition, and are therefore only ordered with respect to a subset of the requests that were executed. We also limit the total number of divergent histories, which in the case of Zeno cannot exceed, at any time, $\lfloor \frac{N-|failed|}{f+1-|failed|} \rfloor$, where $|failed|$ is the current number of failed servers, N is the total number of servers and f is the maximum number of servers that can fail.

We also specify that certain operations are *committed*. Each history has a prefix of committed operations, and the committed prefixes are related by containment.

Hence, all histories agree on the relative order of their committed operations, and the order cannot change in the future. Aside from this restriction, histories can be merged (corresponding to a partition healing) and can be forked, which corresponds to duplicating one of the sets in the multi-set.

Given this state, clients can execute two types of operations, weak and strong, as follows. Any operation begins its execution cycle by being inserted at the end of any non-empty subset of the histories. At this and any subsequent time, a weak operation may return, with the corresponding result reflecting the execution of all the operations that precede it. In this case, we say that the operation is *weakly complete*. For strong operations, they must wait until they are committed (as defined above) before they can return with a similar way of computing the result. We assume that each correct client is *well-formed*: it never issues a new request before its previous (weak or strong) request is (weakly or strongly, respectively) complete.

The merge operation takes two histories and produces a new history, containing all operations in both histories and preserving the ordering of committed operations. However, the weak operations can appear in arbitrary ordering in the merged histories, preserving the causal order of operations invoked by the same client. This implies that weak operations may commit in a different order than when they were weakly completed.

3.2 Liveness

On the liveness side, our service guarantees that a request issued by a correct client is processed and a response is returned to the client, provided that the client can communicate with *enough* replicas in a timely manner.

More precisely, we assume a default round-trip delay Δ and we say that a set of servers $\Pi' \subseteq \Pi$, is *eventually synchronous* if there is a time after which every two-way message exchange within Π' takes at most Δ time units. We also assume that every two correct servers or clients can eventually reliably communicate. Now our progress requirements can be put as follows:

(L1) If there exists an eventually synchronous set of $f+1$ correct servers Π', then every weak request issued by a correct client is eventually weakly complete.

(L2) If there exists an eventually synchronous set of $2f+1$ correct servers Π', then every weakly complete request or a strong request issued by a correct client is eventually committed.

In particular, **(L1)** and **(L2)** imply that if there is a an eventually synchronous set of $2f+1$ correct replicas, then *each* (weak or strong) request issued by a correct client will eventually be committed.

As we will explain later, ensuring **(L1)** in the presence of partitions may require unbounded storage. We will present a protocol addition that bounds the storage requirements at the expense of relaxing **(L1)**.

4 Zeno Protocol

4.1 System model

Zeno is a BFT state machine replication protocol. It requires $N = (3f+1)$ replicas to tolerate f Byzantine faults, i.e., we make no assumption about the behavior of faulty replicas. Zeno also tolerates an arbitrary number of Byzantine clients. We assume no node can break cryptographic techniques like collision-resistant digests, encryption, and signing. The protocol we present in this paper uses public key digital signatures to authenticate communication. In a separate technical report [31], we present a modified version of the protocol that uses more efficient symmetric cryptography based on message authentication codes (MACs).

The protocol uses two kinds of quorums: *strong quorums* consisting of any group of $2f+1$ distinct replicas, and *weak quorums* of $f+1$ distinct replicas.

The system easily generalizes to any $N \geq 3f+1$, in which case the size of *strong quorums* becomes $\lceil \frac{N+f+1}{2} \rceil$, and weak quorums remain the same, independent of N. Note that one can apply our techniques in very large replica groups (where $N \gg 3f+1$) and still make progress as long as $f+1$ replicas are available, whereas traditional (strongly consistent) BFT systems can be blocked unless at least $\lceil \frac{N+f+1}{2} \rceil$ replicas, growing with N, are available.

4.2 Overview

Like most traditional BFT state machine replication protocols, Zeno has three components: *sequence number assignment* (Section 4.4) to determine the total order of operations, *view changes* (Section 4.5) to deal with leader replica election, and *checkpointing* (Section 4.8) to deal with garbage collection of protocol and application state.

The execution goes through a sequence of configurations called *views*. In each view, a designated leader replica (the *primary*) is responsible for assigning monotonically increasing sequence numbers to clients' operations. A replica j is the primary for the view numbered v iff $j = v \mod N$.

At a high level, normal case execution of a request proceeds as follows. A client first sends its request to all replicas. A designated primary replica assigns a sequence number to the client request and broadcasts this proposal to the remaining replicas. Then all replicas execute the request and return a reply to the client.

Name	Meaning
v	current view number
n	highest sequence number executed
h	history, a hash-chain digest of the requests
o	operation to be performed
t	timestamp assigned by the client to each request
s	flag indicating if this is a strong operation
r	result of the operation
$D(.)$	cryptographic digest function
CC	highest commit certificate
ND	non-deterministic argument to an operation
OR	Order Request message

Table 1: Notations used in message fields.

Once the client gathers sufficiently many *matching* replies—replies that agree on the operation result, the sequence number, the view, and the replica history—it returns this result to the application. For weak requests, it suffices that a single correct replica returned the result, since that replica will not only provide a correct weak reply by properly executing the request, but it will also eventually commit that request to the linear history of the service. Therefore, the client need only collect matching replies from a *weak quorum* of replicas. For strong requests, the client must wait for matching replies from a *strong quorum*, that is, a group of at least $2f+1$ distinct replicas. This implies that Zeno can complete many weak operations in parallel across different partitions when only weak quorums are available, whereas it can complete strong operations only when there are strong quorums available.

Whenever operations do not make progress, or if replicas agree that the primary is faulty, a view change protocol tries to elect a new primary. Unlike in previous BFT protocols, view changes in Zeno can proceed with the concordancy of only a weak quorum. This can allow multiple primaries to coexist in the system (e.g., during a network partition) which is necessary to make progress with eventual consistency. However, as soon as these multiple views (with possibly divergent sets of operations) detect each other (Section 4.6), they reconcile their operations via a merge procedure (Section 4.7), restoring consistency among replicas.

In what follows, messages with a subscript of the form σ_c denote a public-key signature by principal c. In all protocol actions, malformed or improperly signed messages are dropped without further processing. We interchangeably use terms "non-faulty" and "correct" to mean system components (e.g., replicas and clients) that follow our protocol faithfully. Table 1 collects our notation.

We start by explaining the protocol state at the replicas. Then we present details about the three protocol components. We used Zyzzyva [23] as a starting point for designing Zeno. Therefore, throughout the presentation, we will explain how Zeno differs from Zyzzyva.

4.3 Protocol State

Each replica i maintains the highest sequence number n it has executed, the number v of the view it is currently participating in, and an ordered history of requests it has executed along with the ordering received from the primary. Replicas maintain a hash-chain digest h_n of the n operations in their history in the following way: $h_{n+1} = D(h_n, D(\text{REQ}_{n+1}))$, where D is a cryptographic digest function and REQ_{n+1} is the request assigned sequence number $n+1$.

A prefix of the ordered history upto sequence number ℓ is called *committed* when a replica gathers a *commit certificate* (denoted *CC* and described in detail in Section 4.4) for ℓ; each replica only remembers the highest CC it witnessed.

To prevent the history of requests from growing without bounds, replicas assemble checkpoints after every *CHKP_INTERVAL* sequence numbers. For every checkpoint sequence number ℓ, a replica first obtains the *CC* for ℓ and executes all operations upto and including ℓ. At this point, a replica takes a snapshot of the application state and stores it (Section 4.8).

Replicas remember the set of operations received from each client c in their *request[c]* buffer and only the last reply sent to each client in their *reply[c]* buffer. The *request* buffer is flushed when a checkpoint is taken.

4.4 Sequence Number Assignment

To describe how sequence number assignment works, we follow the flow of a request.

Client sends request. A correct client c sends a request $\langle \text{REQUEST}, o, t, c, s \rangle_{\sigma_c}$ to all replicas, where o is the operation, t is a sequence number incremented on every request, and s is the strong operation flag.

Primary assigns sequence number and broadcasts order request (OR) message. If the last operation executed for this client has timestamp $t' = t-1$, then primary i assigns the next available sequence number $n+1$ to this request, increments n, and then broadcasts a $\langle \text{OR}, v, n, h_n, D(\text{REQ}), i, s, ND \rangle_{\sigma_i}$ message to backup replicas. ND is a set of non-deterministic application variables, such as a seed for a pseudorandom number generator, used by the application to generate non-determinism.

Replicas receive OR. When a replica j receives an OR message and the corresponding client request, it first checks if both are authentic, and then checks if it is in view v. If valid, it calculates $h'_{n+1} = D(h_n, D(\text{REQ}))$ and checks if h'_{n+1} is equal to the history digest in the OR message. Next, it increments its highest sequence number n, and executes the operation o from REQ on the application state and obtains a reply r. A replica sends the reply $\langle\langle$SPECREPLY$, v, n, h_n, D(r), c, t\rangle_{\sigma_j}, j, r, \mathrm{OR}\rangle$ immediately to the client if s is *false* (i.e., this is a weak request). If s is *true*, then the request must be committed before replying, so a replica first multicasts a $\langle$COMMIT$, \mathrm{OR}, j\rangle_{\sigma_j}$ to all others. When a replica receives at least $2f+1$ such COMMIT messages (including its own) matching in n, v, h_n, $D(\mathrm{REQ})$, it forms a commit certificate *CC* consisting of the set of COMMIT messages and the corresponding OR, stores the *CC*, and sends the reply to the client in a message $\langle\langle$REPLY$, v, n, h_n, D(r), c, t\rangle_{\sigma_j}, j, r, \mathrm{OR}\rangle$. The primary follows the same logic to execute the request, potentially committing it, and sending the reply to the client. Note that the commit protocol used for strong requests will also add all the preceding weak requests to the set of committed operations.

Client receives responses. For weak requests, if a client receives a weak quorum of SPECREPLY messages matching in their v, n, h, r, and OR, it considers the request weakly complete and returns a weak result to the application. For strong requests, a client requires matching REPLY messages from a strong quorum to consider the operation complete.

Fill Hole Protocol. Replicas only execute requests—both weak and strong—in sequence number order. However, due to message loss or other network disruptions, a replica i may receive an OR or a COMMIT message with a higher-than-expected sequence number (that is, $\mathrm{OR}.n > n+1$); the replica discards such messages, asking the primary to "fill it in" on what it has missed (the OR messages with sequence numbers between $n+1$ and $\mathrm{OR}.n$) by sending the primary a $\langle$FILLHOLE$, v, n, \mathrm{OR}.n, i\rangle$ message. Upon receipt, the primary resends all of the requested OR messages back to i, to bring it up-to-date.

Comparison to Zyzzyva. There are four important differences between Zeno and Zyzzyva in the normal execution of the protocol.

First, Zeno clients only need matching replies from a weak quorum, whereas Zyzzyva requires at least a strong quorum; this leads to significant increase in availability, when for example only between $f+1$ and $2f$ replicas are available. It also allows for slightly lower overhead at the client due to reduced message processing requirements, and to a lower latency for request execution when inter-node latencies are heterogeneous.

Second, Zeno requires clients to use sequential timestamps instead of monotonically increasing but not necessarily sequential timestamps (which are the norm in comparable systems). This is required for garbage collection (Section 4.8). This raises the issue of how to deal with clients that reboot or otherwise lose the information about the latest sequence number. In our current implementation we are not storing this sequence number persistently before sending the request. We chose this because the guarantees we obtain are still quite strong: the requests that were already committed will remain in the system, this does not interfere with requests from other clients, and all that might happen is the client losing some of its initial requests after rebooting or oldest uncommitted requests. As future work, we will devise protocols for improving these guarantees further, or for storing sequence numbers efficiently using SSDs or NVRAM.

Third, whereas Zyzzyva offers a single-phase performance optimization, in which a request commits in only three message steps under some conditions (when all $3f+1$ replicas operate roughly synchronously and are all available and non-faulty), Zeno disables that optimization. The rationale behind this removal is based on the view change protocol (Section 4.5) so we defer the discussion until then. A positive side-effect of this removal is that, unlike with Zyzzyva, Zeno does not entrust potentially faulty clients with any protocol step other than sending requests and collecting responses.

Finally, clients in Zeno send the request to all replicas whereas clients in Zyzzyva send the request only to the primary replica. This change is required only in the MAC version of the protocol but we present it here to keep the protocol description consistent. At a high level, this change is required to ensure that a faulty primary cannot prevent a correct request that has weakly completed from committing—the faulty primary may manipulate a few of the MACs in an authenticator present in the request before forwarding it to others, and during commit phase, not enough correct replicas correctly verify the authenticator and drop the request. Interestingly, we find that the implementations of both PBFT and Zyzzyva protocols also require the clients to send the request directly to all replicas.

Our protocol description omits some of the pedantic details such as handling faulty clients or request retransmissions; these cases are handled similarly to Zyzzyva and do not affect the overheads or benefits of Zeno when compared to Zyzzyva.

4.5 View Changes

We now turn to the election of a new primary when the current primary is unavailable or faulty. The key point behind our view change protocol is that it must be able to proceed when only a weak quorum of replicas is available unlike view change algorithms in strongly consistent BFT systems which require availability of a strong quorum to make progress. The reason for this is the following: strongly consistent BFT systems rely on the *quorum* *intersection property* to ensure that if a strong quorum Q decides to change view and another strong quorum Q' decides to commit a request, there is at least one non-faulty replica in both quorums ensuring that view changes do not "lose" requests committed previously. This implies that the sizes of strong quorums are at least $2f+1$, so that the intersection of any two contains at least $f+1$ replicas, including—since no more than f of those can be faulty—at least one non-faulty replica. In contrast, Zeno does not require view change quorums to intersect; a weak request missing from a view change will be eventually committed when the correct replica executing it manages to reach a strong quorum of correct replicas, whereas strong requests missing from a view change will cause a subsequent provable divergence and application-state merge.

View Change Protocol. A client c retransmits the request to all replicas if it times out before completing its request. A replica i receiving a client retransmission first checks if the request is already executed; if so, it simply resends the SPECREPLY/REPLY to the client from its *reply[c]* buffer. Otherwise, the replica forwards the request to the primary and starts a IHateThePrimary timer.

In the latter case, if the replica does not receive an OR message before it times out, it broadcasts $\langle \text{IHATETHEPRIMARY}, v\rangle_{\sigma_i}$ to all replicas, but continues to participate in the current view. If a replica receives such accusations from a weak quorum, it stops participating in the current view v and sends a $\langle \text{VIEWCHANGE}, v+1, CC, \mathcal{O}\rangle_{\sigma_i}$ to other replicas, where CC is the highest commit certificate, and $\mathcal{O}$ is i's ordered request history since that commit certificate, i.e., all OR messages for requests with sequence numbers higher than the one in CC. It then starts the view change timer.

The primary replica j for view $v+1$ starts a timer with a shorter timeout value called the aggregation timer and waits until it collects a set of VIEWCHANGE messages for view $v+1$ from a *strong* quorum, or until its aggregation timer expires. If the aggregation timer expires and the primary replica has collected $f+1$ or more such messages, it sends a $\langle \text{NEWVIEW}, v+1, \mathcal{P}\rangle_{\sigma_j}$ to other replicas, where $\mathcal{P}$ is the set of VIEWCHANGE messages it gathered (we call this a *weak view change*, as opposed to one where a strong quorum of replicas participate which is called a *strong view change*). If a replica does not receive the NEWVIEW message before the view change timer expires, it starts a view change into the next view number.

Note that waiting for messages from a strong quorum is not needed to meet our eventual consistency specification, but helps to avoid a situation where some operations are not immediately incorporated into the new view, which would later create a divergence that would need to be resolved using our merge procedure. Thus it improves the availability of our protocol.

Each replica locally calculates the initial state for the new view by executing the requests contained in $\mathcal{P}$, thereby updating both n and the history chain digest h_n. The order in which these requests are executed and how the initial state for the new view is calculated is related to how we merge divergent states from different replicas, so we defer this explanation to Section 4.7. Each replica then sends a $\langle \text{VIEWCONFIRM}, v+1, n, h_n, i\rangle_{\sigma_i}$ to all others, and once it receives such VIEWCONFIRM messages matching in $v+1$, n, and h from a weak or a strong quorum (for weak or strong view changes, respectively) the replica becomes active in view $v+1$ and stops processing messages for any prior views.

The view change protocol allows a set of $f+1$ correct but slow replicas to initiate a global view change even if there is a set of $f+1$ synchronized correct replicas, which may affect our liveness guarantees (in particular, the ability to eventually execute weak requests when there is a synchronous set of $f+1$ correct servers). We avoid this by prioritizing client requests over view change requests as follows. Every replica maintains a set of client requests that it received but have not been processed (put in an ordered request) by the primary. Whenever a replica i receives a message from j related to the view change protocol (IHATETHEPRIMARY, VIEWCHANGE, NEWVIEW, or VIEWCONFIRM) for a higher view, i first forwards the outstanding requests to the current primary and waits until the corresponding ORs are received or a timer expires. For each pending request, if a valid OR is received, then the replica sends the corresponding response back to the client. Then i processes the original view change related messages from j according to the protocol described above. This guarantees that the system makes progress even in the presence of continuous view changes caused by the slow replicas in such pathological situations.

Comparison to Zyzzyva. View changes in Zeno differ from Zyzzyva in the size of the quorum required for a view change to succeed: we require $f+1$ view change messages before a new view can be announced, whereas previous protocols required $2f+1$ messages. Moreover, the way a new view message is processed is also different in Zeno. Specifically, the start state in a new view must incorporate not only the highest CC in the VIEWCHANGE messages, but also all ORDERREQ that appear in any VIEWCHANGE message from the previous view. This guarantees that a request is incorporated within the state of a new view even if only a single replica reports it; in contrast, Zyzzyva and other similar protocols require support from a weak quorum for every request moved forward through a view change. This is required in Zeno since it is possible that only one replica supports an operation that was executed in a weak view and no other non-faulty replica has seen that operation, and because bringing such operations to a higher view is needed to ensure that weak requests are eventually committed.

The following sections describe additions to the view change protocols to incorporate functionality for detecting and merging concurrent histories, which are also exclusive to Zeno.

4.6 Detecting Concurrent Histories

Concurrent histories (i.e., divergence in the service state) can be formed for several reasons. This can occur when the view change logic leads to the presence of two replicas that simultaneously believe they are the primary, and there are a sufficient number of other replicas that also share that belief and complete weak operations proposed by each primary. This could be the case during a network partition that splits the set of replicas into two subsets, each of them containing at least $f+1$ replicas.

Another possible reason for concurrent histories is that the base history decided during a view change may not have the latest committed operations from prior views. This is because a view change quorum (a weak quorum) may not share a non-faulty replica with prior commitment quorums (strong quorums) and remaining replicas; as a result, some committed operations may not appear in VIEWCHANGE messages and, therefore, may be missing from the new starting state in the NEWVIEW message.

Finally, a misbehaving primary can also cause divergence by proposing the same sequence numbers to different operations, and forwarding the different choices to disjoint sets of replicas.

Basic Idea. Two request history orderings $h^i_1, h^i_2, \ldots$ and $h^j_1, h^j_2, \ldots$, present at replicas i and j respectively, are called *concurrent* if there exists a sequence number n such that $h^i_n \neq h^j_n$; because of the collision resistance of the hash chaining mechanism used to produce history digests, this means that the sequence of requests represented by the two digests differ as well. A replica compares history digests whenever it receives protocol messages such as OR, COMMIT, or CHECKPOINT (described in Section 4.8) that purport to share the same history as its own.

For clarity, we first describe how we detect divergence within a view and then discuss detection across views. We also defer details pertaining to garbage collection of replica state until Section 4.8.

4.6.1 Divergence between replicas in same view

Suppose replica i is in view v_i, has executed up to sequence number n_i, and receives a properly authenticated message $\langle \text{OR}, v_i, n_j, h_{n_j}, D(\text{REQ}), p, s, ND \rangle_{\sigma_p}$ or $\langle \text{COMMIT}, \langle \text{OR}, v_i, n_j, h_{n_j}, D(\text{REQ}), p, s, ND \rangle_{\sigma_p}, j \rangle_{\sigma_j}$ from replica j.

If $n_i < n_j$, i.e., j has executed a request with sequence number n_j, then the fill-hole mechanism is started, and i receives from j a message $\langle \text{OR}, v', n_i, h_{n_i}, D(\text{REQ}'), k, s, ND \rangle_{\sigma_k}$, where $v' \leq v_i$ and $k = primary(v')$.

Otherwise, if $n_i \geq n_j$, both replicas have executed a request with sequence number n_j and therefore i must have the some $\langle \text{OR}, v', n_j, h_{n_j}, D(\text{REQ}'), k, s, ND \rangle_{\sigma_k}$ message in its log, where $v' \leq v_i$ and $k = primary(v')$.

If the two history digests match (the local h_{n_j} or h_{n_i}, depending on whether $n_i \geq n_j$, and the one received in the message), then the two histories are consistent and no concurrency is deduced.

If instead the two history digests differ, the histories must differ as well. If the two OR messages are authenticated by the same primary, together they constitute a *proof of misbehavior (POM)*; through an inductive argument it can be shown that the primary must have assigned different requests to the same sequence number n_j. Such a POM is sufficient to initiate a view change and a merge of histories (Section 4.7).

The case when the two OR messages are authenticated by different primaries indicates the existence of divergence, caused for instance by a network partition, and we discuss how to handle it next.

4.6.2 Divergence across views

Now assume that replica i receives a message from replica j indicating that $v_j > v_i$. This could happen due to a partition, during which different subsets changed views independently, or due to other network and replica asynchrony. Replica i requests the NEWVIEW message for v_j from j. (The case where $v_j < v_i$ is similar, with the exception that i pushes the NEWVIEW message to j instead.)

When node i receives and verifies the $\langle \text{NEWVIEW}, v_j, \mathcal{P} \rangle_{\sigma_p}$ message, where p is the issuing primary of view v_j, it compares its local history to the sequence of OR messages obtained after ordering the OR message present in the NEWVIEW message (according to the procedure described in Section 4.7). Let n_l and n_h be the lowest and highest sequence numbers of those OR messages, respectively.

Case 1: $[n_i < n_l]$ Replica i is missing future requests, so it sends j a FILLHOLE message requesting the OR messages between n_i and n_l. When these are received, it compares the OR message for n_i to detect if there was divergence. If so, the replica obtained a *proof of divergence (POD)*, consisting of the two OR messages, which it can use to initiate a new view change. If not, it executes the operations from n_i to n_l and ensures that its history after executing n_l is consistent with the *CC* present in the NEWVIEW message, and then handles the NEWVIEW message normally and enters v_j. If the histories do not match this also constitutes a POD.

Case 2: $[n_l \leq n_i \leq n_h]$ Replica i must have the corresponding ORDERREQ for all requests with sequence numbers between n_l and n_i and can therefore check if its history diverges from that which was used to generate the new view. If it finds no divergence, it moves to v_j and calculates the start state based on the NEWVIEW message (Section 4.5). Otherwise, it generates a *POD* and initiates a merge.

Case 3: $[n_i > n_h]$ Replica i has corresponding OR messages for all sequence numbers appearing in the NEWVIEW and can check for divergence. If no divergence is found, the replica has executed more requests in a lower view v_i than v_j. Therefore, it generates a *Proof of Absence (POA)*, consisting of all OR messages with sequence numbers in $[n_l, n_i]$ and the NEWVIEW message for the higher view, and initiates a merge. If divergence is found, i generates a *POD* and also initiates a merge.

Like traditional view change protocols, a replica i does not enter v_j if the NEWVIEW message for that view did not include all of i's committed requests. This is important for the safety properties providing guarantees for strong operations, since it excludes a situation where requests could be committed in v_j without seeing previously committed requests.

4.7 Merging Concurrent Histories

Once concurrent histories are detected, we need to merge them in a deterministic order. The solution we propose is to extend the view change protocol, since many of the functionalities required for merging are similar to those required to transfer a set of operations across views.

We extend the view change mechanism so that view changes can be triggered by either PODs, POMs or POAs. When a replica obtains a POM, a POD, or a POA after detecting divergence, it multicasts a message of the form $\langle \text{POMMSG}, v, POM \rangle_{\sigma_i}$, $\langle \text{PODMSG}, v, POD \rangle_{\sigma_i}$, or $\langle \text{POAMSG}, v, POA \rangle_{\sigma_i}$ in addition to the VIEWCHANGE message for v. Note here that v in *POM* and *POD* is one higher than the highest view number present in the conflicting ORDERREQ messages, or one higher than the view number in the NEWVIEW component in the case of a *POA*.

Upon receiving an authentic and valid POMMSG or PODMSG or a POAMSG, a replica broadcasts a VIEWCHANGE along with the triggering POM, POD, or POA message.

The view change mechanism will eventually lead to the election of a new primary that is supposed to multicast a NEWVIEW message. When a node receives such a message, it needs to compute the start state for the next view based on the information contained in that message. The new start state is calculated by first identifying the highest *CC* present among all VIEWCHANGE messages; this determines the new base history digest h_n for the start sequence number n of the new view.

But nodes also need to determine how to order the different OR messages that are present in the NEWVIEW message but not yet committed. Contained OR messages (potentially including concurrent requests) are ordered using a deterministic function of the requests that produces a total order for these requests. Having a fixed function allows all nodes receiving the NEWVIEW message to easily agree on the final order for the concurrent OR present in that message. Alternatively, we could let the primary replica propose an ordering, and disseminate it as an additional parameter of the NEWVIEW message.

Replicas receiving the NEWVIEW message then execute the requests in the OR messages according to that fixed order, updating their histories and history digests. If a replica has already executed some weak operations in an order that differs from the new ordering, it first rolls back the application state to the state of the last checkpoint (Section 4.8) and executes all operations after the checkpoint, starting with committed requests and then with the weak requests ordered by the NEWVIEW message. Finally, the replica broadcasts a VIEWCONFIRM message. As mentioned, when a replica collects matching VIEWCONFIRM messages on v, n, and h_n it becomes active in the new view.

Our merge procedure re-executes the concurrent operations sequentially, without running any additional or alternative application-specific conflict resolution procedure. This makes the merge algorithm slightly simpler, but requires the application upcall that executes client operations to contain enough information to identify and resolve concurrent operations. This is similar to the design choice made by Bayou [33] where special concurrency detection and merge procedure are part of each service operation, enabling servers to automatically detect and resolve conflicts.

Limiting the number of merge operations. A faulty replica can trigger multiple merges by producing a new POD for each conflicting request in the same view, or generating PODs for requests in old views where itself or a colluding replica was the primary. To avoid this potential performance problem, replicas remember the last POD, POM, or a POA every other replica initiated, and reject a POM/POD/POA from the same or a lower view coming from that replica. This ensures that a faulty replica can initiate a POD/POM/POA only once from each view it participated in. This, as we show in Section 5, helps establish our liveness properties.

Recap comparison to Zyzzyva. Zeno's view changes motivate our removal of the single-phase Zyzzyva optimization for the following reason: suppose a strong client request REQ was executed (and committed) at sequence number n at $3f+1$ replicas. Now suppose there was a weak view change, the new primary is faulty, and only $f+1$ replicas are available. A faulty replica among those has the option of reporting REQ in a different order in its VIEWCHANGE message, which enables the primary to order REQ arbitrarily in its NEWVIEW message; this is possible because only a single—potentially faulty—replica need report any request during a Zeno view change. This means that linearizability is violated for this strong, committed request REQ. Although it may be possible to design a more involved view change to preserve such orderings, we chose to keep things simple instead. As our results show, in many settings where eventual consistency is sufficient for weak operations, our availability under partitions tramps any benefits from increased throughput due to the Zyzzyva's optimized single-phase request commitment.

4.8 Garbage Collection

The protocol we have presented so far has two important shortcomings: the protocol state grows unboundedly, and weak requests are never committed unless they are followed by a strong request.

To address these issues, Zeno periodically takes checkpoints, garbage collecting its logs of requests and forcing weak requests to be committed.

When a replica receives an ORDERREQ message from the primary for sequence number M, it checks if $M \bmod \text{CHKP\_INTERVAL} = 0$. If so, it broadcasts the COMMIT message corresponding to M to other replicas. Once a replica receives $2f+1$ COMMIT messages matching in v, M, and h_M, it creates the commit certificate for sequence number M. It then sends a $\langle \text{CHECKPOINT}, v, M, h_M, App \rangle_{\sigma_j}$ to all other replicas. The *App* is a snapshot of the application state after executing requests upto and including M. When it receives $f+1$ matching CHECKPOINT messages, it considers the checkpoint stable, stores this proof, and discards all ordered requests with sequence number lower than n along with their corresponding client requests.

Also, in case the checkpoint procedure is not run within the interval of T_{CHKP} time units, and a replica has some not yet committed ordered requests, the replica also initiates the commit step of the checkpoint procedure. This is done to make sure that pending ordered requests are committed when the service is rarely used by other clients and the sequence numbers grow very slowly.

Our checkpoint procedure described so far poses a challenge to the protocol for detecting concurrent histories. Once old requests have been garbage-collected, there is no way to verify, in the case of a slow replica (or a malicious replica pretending to be slow) that presents an old request, if that request has been committed at that sequence number or if there is divergence.

To address this, clients send sequential timestamps to uniquely identify each one of their own operations, and we added a list of per-client timestamps to the checkpoint messages, representing the maximum operation each client has executed up to the checkpoint. This is in contrast with previous BFT replication protocols, including Zyzzyva, where clients identified operations using timestamps obtained by reading their local clocks. Concretely, a replica sends $\langle \text{CHECKPOINT}, v, M, h_M, App, CSet \rangle_{\sigma_j}$, where *CSet* is a vector of $\langle c, t \rangle$ tuples, where t is the timestamp of the last committed operation from c.

This allows us to detect concurrent requests, even if some of the replicas have garbage-collected that request. Suppose a replica i receives an OR with sequence number n that corresponds to client c's request with timestamp t_1. Replica i first obtains the timestamp of the last executed operation of c in the highest checkpoint $t_c = CSet[c]$. If $t_1 \leq t_c$, then there is no divergence since the client request with timestamp t_1 has already been committed. But if $t_1 > t_c$, then we need to check if some other request was assigned n, providing a proof of divergence. If $n < M$, then the CHECKPOINT and the OR form a POD since some other request was assigned n. Else, we can perform regular conflict detection procedure to identify concurrency (see Section 4.6).

Note that our checkpoints become stable only when there are at least $2f+1$ replicas that are able to agree. In the presence of partitions or other unreachability situations where only weak quorums can talk to each other, it may not be possible to gather a checkpoint, which implies that Zeno must either allow the state concerning tentative operations to grow without bounds, or weaken its liveness guarantees. In our current protocol we chose the latter, and so replicas stop participating once they reach a maximum number of tentative operations they can execute, which could be determined based on their available storage resources (memory as well as the disk space). Garbage collecting weak operations and the resulting impact on conflict detection is left as a future work.

5 Correctness

In this section, we sketch the proof that Zeno satisfies the safety properties specified in Section 3. A proof sketch for liveness properties is presented in a separate technical report [31].

In Zeno, a (weak or strong) response is based on identical histories of at least $f+1$ replicas, and, thus, at least one of these histories belongs to a correct replica. Hence, in the case that our garbage collection scheme is not initiated, we can reformulate the safety requirements as follows: **(S1)** the local history maintained by a correct replica consists of a prefix of committed requests extended with a sequence of speculative requests, where no request appears twice, **(S2)** a request associated with a correct client c appears, in a history at a correct replica only if c has previously issued the request, and **(S3)** the committed prefixes of histories at every two correct replicas are related by containment, and **(S4)** at any time, the number of conflicting histories maintained at correct replica does not exceed $maxhist = \lfloor (N-f')/(f-f'+1) \rfloor$, where f' is the number of currently failed replicas and N is the total number of replicas required to tolerate a maximum of f faulty replicas. Here we say that two histories are conflicting if none of them is a prefix of the other.

Properties **(S1)** and **(S2)** are implied by the state maintenance mechanism of our protocol and the fact that only properly signed requests are put in a history by a correct replica. The special case when a prefix of a history is hidden behind a checkpoint is discussed later.

A committed prefix of a history maintained at a correct replica can only be modified by a commitment of a new request or a merge operation. The sub-protocol of Zeno responsible for committing requests are analogous to the two-phase conservative commitment in Zyzzyva [23], and, similarly, guarantees that all committed requests are totally ordered. When two histories are merged at a correct replica, the resulting history adopts the longest committed prefix of the two histories. Thus, inductively, the committed prefixes of all histories maintained at correct replicas are related by containment **(S3)**.

Now suppose that at a given time, the number of conflicting histories maintained at correct replica is more than *maxhist*. Our weak quorum mechanism guarantees that each history maintained at a correct process is supported by at least $f+1$ distinct processes (through sending SPECREPLY and REPLY messages). A correct process cannot concurrently acknowledge two conflicting histories. But when f' replicas are faulty, there can be at most $\lfloor (n-f')/(f-f'+1) \rfloor$ sets of $f+1$ replicas that are disjoint in the set of correct ones. Thus, at least one correct replica acknowledged two conflicting histories — a contradiction establishes **(S4)**.

Checkpointing. Note that our garbage collection scheme may affect property **(S1)**: the sequence of tentative operations maintained at a correct replica may potentially include a committed but already garbage-collected operation. This, however, cannot happen: each round of garbage collection produces a checkpoint that contains the latest committed service state and the logical timestamp of the latest committed operation of every client. Since no correct replica agrees to commit a request from a client unless its previous requests are already committed, the checkpoint implies the set of timestamps of all committed requests of each client. If a replica receives an ordered request of a client c corresponding to a sequence number preceding the checkpoint state, and the timestamp of this request is no later than the last committed request of c, then the replica simply ignores the request, concluding that the request is already committed. Hence, no request can appear in a local history twice.

6 Evaluation

We have implemented a prototype of Zeno as an extension to the publicly available Zyzzyva source code [24].

Our evaluation tries to answer the following questions: (1) Does Zeno incur more overhead than existing protocols in the normal case? (2) Does Zeno provide higher availability compared to existing protocols when there are more than f unreachable nodes? (3) What is the cost of merges?

Experimental setup. We set $f=1$, and the minimum number of replicas to tolerate it, $N=3f+1=4$. We vary the number of clients to increase load. Each physical machine has a dual-core 2.8 GHz AMD processor with 4GB of memory, running a 2.6.20 Linux kernel. Each replica as well as a client runs on a dedicated physical machine. We use Modelnet [35] to simulate a network topology consisting of two hubs connected via a bi-directional link unless otherwise mentioned. Each hub has two servers in all of our experiments but client location varies as per the experiment. Each link has one-way latency of 1 ms and a 100 Mbps bandwidth.

Transport protocols. Zyzzyva, like PBFT, uses multicast to reduce the cost of sending operations from clients to all replicas, so it uses UDP as a transport protocol and implements a simple backoff and retry policy to handle message loss. This is not optimized for periods of congestion and high message loss, such as those we antecipate during merges when the replicas that were partitioned need to bring each other up-to-date. To address this, Zeno uses TCP as the transport layer during the merge procedure but continues to use Zyzzyva's UDP-based transport during normal operation and multicasting communication that is sent to all replicas.

Partition. We simulate network partitions by separating the two hubs from each other. We vary the duration of the partitions from 1 to 5 minutes, based on the observation by Chandra et al. [12] that a large fraction ($> 75\%$) of network disconnectivity events range from 30 to 500 seconds.

6.1 Implementation

Replacing PKI with MACs. Our Zeno prototype uses MACs instead of the slower digital signatures to implement message authentication for the common-case, but still uses signatures for view changes. Using MACs induces some small mechanistic design changes over the protocol description in Section 4; these changes are standard practice in similar protocols including Zyzzyva, and are presented in [31].

Merge. Replicas detect divergence by following the algorithm specified in Section 4.7. We implemented an optimization to the merge protocol where replicas first move to the higher view and then propagate their local uncommitted requests to the primary of the higher view. The primary of the higher view orders these requests as if they are received from the client and hence merges these requests in the history.

6.2 Results

We generate a workload with a varying fraction of strong and weak operations. If each client issued both strong and weak operations, then most clients would block soon after network partitions started. Instead, we simulate two kind of clients: (i) weak clients only issue weak requests and (ii) strong clients always pose strong requests. This allows us to vary the ratio of weak operations (denoted by α) in the total workload with a limited number of clients in the system and long network partitions. We use a micro-benchmark that executes a no-op when the *execute* upcall for the client operation is invoked.

We have also built a simple application on top of Zeno, emulating a shopping cart service with operations to add, remove, and checkout items based on a *key-value* data store. We also implement a simple conflict detection and merge procedure. Due to lack of space, the design and evaluation of this service is presented in the technical report [31].

Protocol	Batch=1	Batch=10
Zyzzyva (single phase)	62 Kops/s	88 Kops/s
Zeno (weak)	60 Kops/s	86 Kops/s
Zeno (strong)	40 Kops/s	82 Kops/s
Zyzzyva (commit opt)	40 Kops/s	82 Kops/s

Table 2: Peak throughput of Zeno and Zyzzyva.

6.2.1 *Maximum throughput in the normal case*

We compare the normal case performance of Zeno with Zyzzyva. In both systems we used the optimization of batching requests to reduce protocol overhead. In this experiment, the clients and servers are connected by a 1 Gbps switch with 0.1 ms round trip latency. We expect the peak throughput of Zeno with weak operations to approximately match the peak throughput of Zyzzyva since both can be completed in a single phase. However, the performance of Zeno with strong operations will be lower than the peak throughput of Zyzzyva since Zeno requires an extra phase to commit a strong operation.

Our results presented in Table 2 show that Zeno and Zyzzyva's throughput are similar, with Zyzzyva achieving slightly (3–6%) higher throughput than Zeno's throughput for weak operations. The results also show that, with batching, Zeno's throughput for strong operations is also close to Zyzzyva's peak throughput: Zyzzyva has 7% higher throughput when the single phase optimization is employed. However, when a single replica is faulty or slow, Zyzzyva cannot achieve the single phase throughput and Zeno's throughput for strong operations is identical to Zyzzyva's performance with a faulty replica.

6.2.2 *Partition with no concurrency*

For all the remaining experiments, we use Modelnet setup and disable multicast since Modelnet does not support it. We use a client population of 4 nodes, each sending a new request of minimal payload (2 Bytes) as soon as it has completed the previous request. This generates a steady load of approximately 500 requests/sec on the system. This is similar to an example SLA provided in Dynamo [15]. We use a batch size of 1 for both Zyzzyva and Zeno, since it is sufficient to handle the incoming request load.

In this experiment, all clients reside in the first LAN. We initiate a partition at 90 seconds which continues for a minute. Since there are no clients in the second LAN, there are no requests processed in it and hence there is no concurrency, which avoids the cost of merging. Replicas with id 0 (primary for view initial view 0) and 1 reside in the first LAN while replicas with ids 2 and 3 reside in the second LAN. We also present the results of Zyzzyva to compare the performance in both normal cases as well as under the given failure.

Varying α. We vary the mix of weak and strong operations in the workload, and present the results in Figure 1. First, strong operations block as soon as the failure starts which is expected since not enough replicas are reachable from the first LAN to complete the strong operation. However, as soon as the partition heals, we observe that strong operations start to be completed. Note also

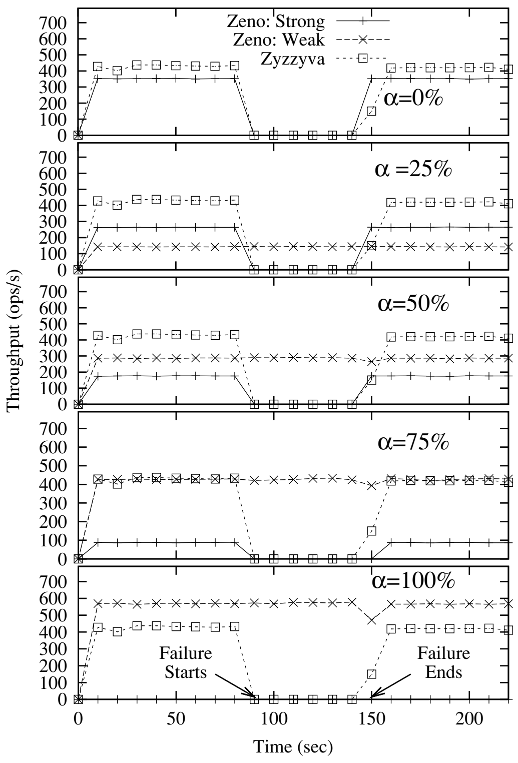


Figure 1: Two replicas are disconnected via a partition, that starts at time 90 and continues for 60 seconds. Parameter α represents the fraction of weak operations in the workload. Note that the throughput of weak and strong operations in Zeno is presented separately for clarity.

that Zyzzyva also blocks as soon as the failure starts and resumes as soon as it ends.

Second, weak operations continue to be processed and completed during the partition and this is because Zeno requires (for $f = 1$) only 2 non-faulty replicas to complete the operation. The fraction of total requests completed increases as α increases, essentially improving the availability of such operations despite network partitions.

Third, when replicas in the other LAN are reachable again, they need to obtain the missing requests from the first LAN. Since the number of weak operations performed in the first LAN increases as α increases, the time to update the lagging replicas in the other partition also goes up; this puts a temporary strain on the network, evidenced by the dip in the throughput of weak operations when the partition heals. However, this dip is brief compared to the duration of the partition. We explore the impact of the duration of partitions next.

Varying partition duration. Using the same setup, we now vary partition durations between 1 and 5 minutes for $\alpha = 75\%$. For each partition duration, we measure the period of unavailability for both weak and strong operations. The unavailability is measured as the number of seconds for which the observed throughput, on either side of the partition, was less than 10% of the average throughput observed before the partition started. Also, the distance from the "Strong" line to the baseline ($x = y$) indicates how soon after healing the partition can strong operations be processed again.

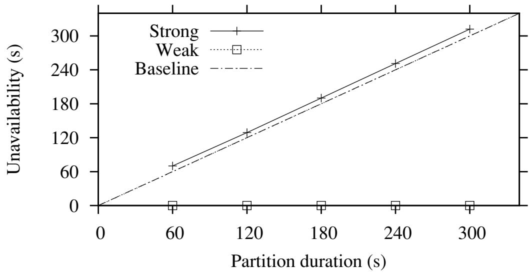


Figure 2: Varying partition durations with no concurrent operations. Baseline represents the minimal unavailability expected for strong operations, which is equal to the partition duration.

Figure 2 presents the results. We observe that weak operations are always available in this experiment since all weak operations were completed in the first LAN and the replicas in the first LAN are up-to-date with each other to process the next weak operation. Strong operations are unavailable for the entire duration of the partition due to unavailability of the replicas in the second LAN and the additional unavailability is introduced by Zeno due to the operation transfer mechanism. However, the additional delay is within 4% of the partition duration (12 seconds for a 5 minute partition). Our current prototype is not yet optimized and we believe that the delay could be further reduced.

Varying request size. In this experiment, we simulate a partition for 60 seconds but increase the payload sizes from 2 Bytes to 1 KB, with an equally sized reply. The cumulative bandwidth of requests to be transferred from one LAN to the other is a function of the weak request offered load, the size of the requests, and the duration of the partition. With 60 seconds of partition and an offered load of 500 req/s, the cumulative request payload ranges from approximately 60 KB to 30 MB for 2 Bytes and 1 KB request size respectively. The results we obtained are very similar to those in Figure 1 so we do not repeat them. These show that the time to bring replicas in the second LAN up-to-date does not increase significantly with the increase in request size. Given that we have 100 Mbps links connecting replicas to each other, bandwidth is not a limiting resource for shipping operations at these offered loads.

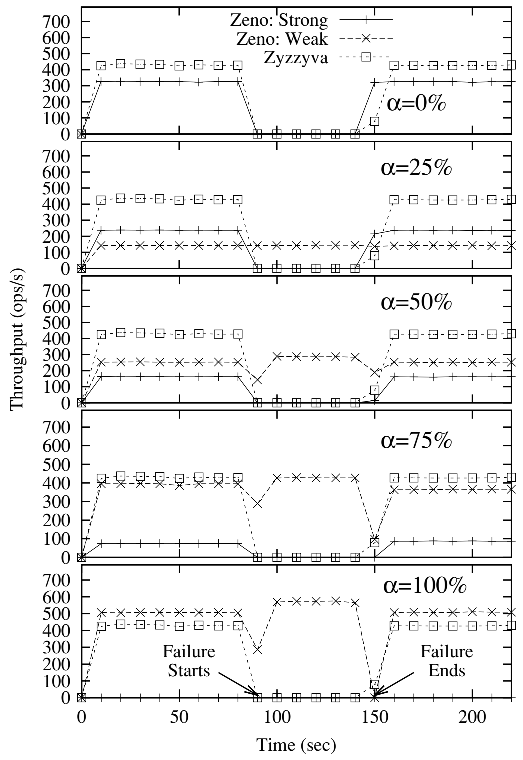


Figure 3: Network partition for 60 seconds starting at time 90 seconds. Note that the throughput of weak and strong operations in Zeno is presented separately for clarity.

6.2.3 *Partition with concurrency*

In this experiment, we keep half the clients on each side of a partition. This ensures that both partitions observe a steady load of weak operations that will cause Zeno to first perform a weak view change and later merge the concurrent weak operations completed in each partition. Hence, this microbenchmark additionally evaluates the cost of weak view changes and the merge procedure. As before, the primary for the initial view resides in the first LAN. We measure the overall throughput of weak and strong operations completed in both partitions. Again, we compare our results to Zyzzyva.

Varying α. Figure 3 presents the results for the throughput of different systems while varying the value of α. We observe three main points.

When $\alpha = 0$, Zeno does not give additional benefits since there are no weak operations to be completed. Also, as soon as the partition starts, strong operations are blocked and resume after the partition heals. As above, Zyzzyva provides greater throughput thanks to its single-phase execution of client requests, but it is as powerless to make progress during partitions as Zeno in the face of strong operations only.

When $\alpha = 25\%$, we have only one client sending *weak* operations in one LAN. Since there are no conflicts, this graph matches that of Figure 1.

When $\alpha \geq 50\%$, we have at least two weak clients, at least one in each LAN. When a partition starts, we observe that the throughput of weak operations first drops; this happens because weak clients in the second partition cannot complete operations as they are partitioned from the current primary. Once they perform the necessary view changes in the second LAN, they resume processing weak operations; this is observed by an increase in the overall throughput of weak operations completed since both partitions can now complete weak operations in parallel – in fact, faster than before the partition due to decreased cryptographic and message overheads and reduced round trip delay of clients in the second partition from the primary in their partition. The duration of the weak operation unavailability in the non-primary partition is proportional to the number of view changes required. In our experiment, since replicas with ids 2 and 3 reside in the second LAN, two view changes were required (to make replica 2 the new primary).

When the partition heals, replicas in the first view detect the existence of concurrency and construct a POD, since replicas in the second LAN are in a higher view (with $v = 2$). At this point, they request a NEWVIEW from the primary of view 2, move to view 2, and then propagate their locally executed weak operations to the primary of view 2. Next, replicas in the first LAN need to fetch the weak operations that completed in the second LAN and needs to complete them before the strong operations can make progress. This results in additional delay before the strong operations can complete, as observed in the figure.

Varying partition duration. Next, we simulate partitions of varying duration as before, for $\alpha = 75\%$. Again, we measure the unavailability of both strong and weak operations using the earlier definition: unavailability is the duration for which the throughput in either partition was less than 10% of average throughput before the failure. With a longer partition duration, the cost of the merge procedure increases since the weak operations from both partitions have to be transferred prior to completing the new client operations.

Figure 4 presents the results. We observe that weak operations experience some unavailability in this scenario, whose duration increases with the length of the partition. The unavailability for weak operations is within 9% of the total time of the partition.

The unavailability of strong operations is at least the duration of the network partition plus the merge cost (similar to that for weak operations). The additional unavailability due to the merge operation is within 14% of the total time of the partition.

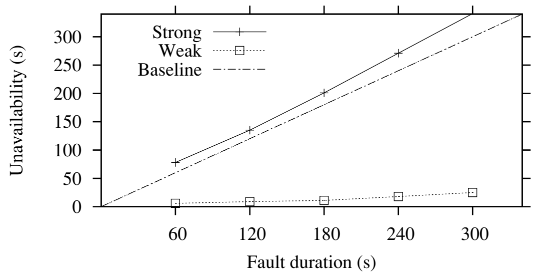


Figure 4: Varying partition durations with concurrent operations. Baseline represents the minimal unavailability expected for strong operations, which is equal to the partition duration.

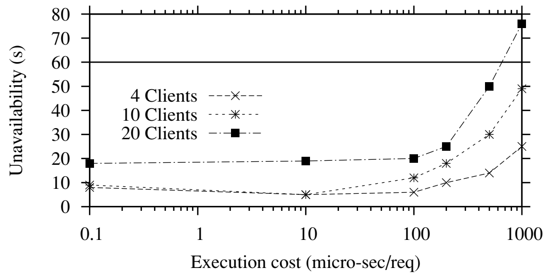


Figure 5: Varying execution cost of operations with increasing request load. 60 second partition duration.

Varying execution cost and request load. In this experiment, we vary the execution cost of each operation as well as increase the request load, by increasing the number of clients, to estimate the cost of merges when the system is loaded. For example, the system was operating at peak cpu utilization with 20 clients and operations with 200 μs/operation or more. Here, we set $\alpha = 100\%$. We present results with a partition duration of 60 seconds in Figure 5. We observe that as the cost of operations system load increases, the unavailability of weak operations also goes up. This is expected because the set of weak operations performed in one partition must be re-executed at the replicas in the other partition during the merge procedure. As the client load and the cost of operation execution increases, the time taken to re-execute the operation also increases. In particular, when the system is operating at 100% cpu utilization, the cost of re-executing the operations will take as much as time as the duration of the partition, and therefore the unavailability in these cases is higher than the partition duration. If, however, the system is not operating at peak utilization, the cost of merging is lower than the partition duration.

Varying request size. We ran an experiment with a 5 minute partition, and varying request sizes from 2 Bytes to 1 KB. The results with different request sizes were similar to those shown in Figure 3 so we do not plot them. We observed that increasing the payload size does not significantly affect the merge duration. This is due to the high speed network connection between replicas.

Summary. Our microbenchmark results show that Zeno significantly improves the availability of weak operations and the cost of merging is reasonable as long as the system is not overloaded. This allows Zeno to quickly start processing strong operations soon after partitions heal.

6.2.4 *Mix of strong and weak operations*

In this experiment, we allow each client to issue a mix of strong and weak operations. Note that as soon as a client issues a strong operation in a partition, it will be blocked until the partition heals. We use a client population of 40 nodes. Each client issues a strong operation with probability p, weak operations with probability $0.8 - p$, and exits from the system with a fixed probability of 0.2. We implement a fixed think time of 10 seconds between operations issued by each client. The think times and the exit probability are obtained from the SpecWeb2005 banking benchmark [10]. Next, we vary p to estimate the impact of failure events such as network partitions on the overall user experience. To give an idea of reference values for p, we looked into the types and frequencies of distinct operations in existing benchmarks. In an e-banking benchmark, and assigning the billing operations to be strong operations, the recommended frequency of such operations follows $p = 0.13$ [10]. In the case of an e-commerce benchmark, if the checkout operation is considered strong while the remaining, such as login, accessing account information and customizations are considered as weak operations, then we obtain $p = 0.05$ [1]. Our experimental results cover these values.

We simulate a partition duration of 60 seconds and calculate the number of clients blocked and the length of time they were blocked during the partition. Figure 6 presents the cumulative distribution function of clients on the y-axis and the maximum duration a client was blocked on the x-axis. This metric allows us to see how clients were affected by the partition. With Zyzzyva, all clients will be blocked for the entire duration of the partition. However, with Zeno, a large fraction of clients do not observe any wait time and this is because they exit from the system after doing a few weak operations. For example, more than 70% of clients do not observe any wait time as long as the probability of performing a strong operation is less than 15%. In summary, this result shows that Zeno significantly improves the user experience and masks the failure events from being exposed to the user as long as the workload contains few strong operations.

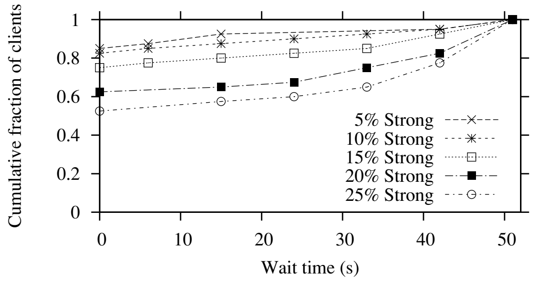


Figure 6: Wait time per client with varying probability p of issuing strong operations.

7 Related Work

The trade-off between consistency, availability and tolerance to network partitions in computing services has become folklore long ago [7].

Most replicated systems are designed to be "strongly" consistent, i.e., provide clients with consistency guarantees that approximate the semantics of a single, correct server, such as single-copy serializability [20] or linearizability [22].

Weaker consistency criteria, which allow for better availability and performance at the expense of letting replicas temporarily diverge and users see inconsistent data, were later proposed in the context of replicated services tolerating crash faults [17, 30, 33, 38]. We improve on this body of work by considering the more challenging Byzantine-failure model, where, for instance, it may not suffice to apply an update at a single replica, since that replica may be malicious and fail to propagate it.

There are many examples of Byzantine-fault tolerant state machine replication protocols, but the vast majority of them were designed to provide linearizable semantics [4, 8, 11, 23]. Similarly, Byzantine-quorum protocols provide other forms of strong consistency, such as safe, regular, or atomic register semantics [27]. We differ from this work by analyzing a new point in the consistency-availability tradeoff, where we favor high availability and performance over strong consistency.

There are very few examples of Byzantine-fault tolerant systems that provide weak consistency.

SUNDR [25] and BFT2F [26] provide similar forms of weak consistency (fork and fork*, respectively) in a client-server system that tolerates Byzantine servers. While SUNDR is designed for an unreplicated service and is meant to minimize the trust placed on that server, BFT2F is a replicated service that tolerates a subset of Byzantine-faulty servers. A system with fork consistency might conceal users' actions from each other, but if it does, users get divided into groups and the members of one group can no longer see any of another group's file system operations.

These two systems propose quite different consistency guarantees from the guarantees provided by Zeno, because the weaker semantics in SUNDR and BFT2F have very different purposes than our own. Whereas we are trying to achieve high availability and good performance with up to f Byzantine faults, the goal in SUNDR and BFT2F is to provide the best possible semantics in the presence of a large fraction of malicious servers. In the case of SUNDR, this means the single server can be malicious, and in the case of BFT2F this means tolerating arbitrary failures of up to $\frac{2}{3}$ of the servers. Thus they associate client signatures with updates such that, when such failures occur, all the malicious servers can do is conceal client updates from other clients. This makes the approach of these systems orthogonal and complementary to our own.

Another example of a system that provides weak consistency in the presence of some Byzantine failures can be found in [32]. However, the system aims at achieving extreme availability but provides almost no guarantees and relies on a trusted node for auditing.

To our knowledge, this paper is the first to consider eventually-consistent Byzantine-fault tolerant generic replicated services.

8 Future Work and Conclusions

In this paper we presented Zeno, a BFT protocol that privileges availability and performance, at the expense of providing weaker semantics than traditional BFT protocols. Yet Zeno provides eventual consistency, which is adequate for many of today's replicated services, e.g., that serve as back-ends for e-commerce websites. Our evaluation of an implementation of Zeno shows it provides better availability than existing BFT protocols, and that overheads are low, even during partitions and merges.

Zeno is only a first step towards liberating highly available but Byzantine-fault tolerant systems from the expensive burden of linearizability. Our eventual consistency may still be too strong for many real applications. For example, the shopping cart application does not necessarily care in what order cart insertions occur, now or eventually; this is probably the case for all operations that are associative and commutative, as well as operations whose effects on system state can easily be reconciled using snapshots (as opposed to merging or totally ordering request histories). Defining required consistency per *operation type* and allowing the replication protocol to relax its overheads for the more "best-effort" kinds of requests could provide significant further benefits in designing high-performance systems that tolerate Byzantine faults.

Acknowledgements

We would like to thank our shepherd, Miguel Castro, the anonymous reviewers, and the members of the MPI-SWS for valuable feedback.

References

[1] TPC-W Benchmark White Paper. `http://www.tpc.org/tpcw/TPC-W_Wh.pdf`.

[2] Amazon S3 Availability Event: July 20, 2008. `http://status.aws.amazon.com/s3-20080720.html`, 2008.

[3] FaceBook's Cassandra: A Structured Storage System on a P2P Network. `http://code.google.com/p/the-cassandra-project/`, 2008.

[4] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. Reiter, and J. J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Brighton, United Kingdom, 2005.

[5] Amazon. Discussion Forum: Thread: Massive (500) Internal Server Error. Outage started 35 minutes ago. `http://developer.amazonwebservices.com/connect/thread.jspa?threadID=19714&start=90&tstart=0`, 2008.

[6] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Banff, Canada, 2001.

[7] E. Brewer. Towards Robust Distributed Systems (Invited Talk). Proceedings of ACM Symposium on Principles of Distributed Computing (PODC), 2000.

[8] M. Castro and B. Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, New Orleans, LA, USA, 1999.

[9] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested Append-Only Memory: Making Adversaries Stick to their Word. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[10] S. P. E. Corporation. Specweb2005 release 1.20 banking workload design document. `http://www.spec.org/web2005/docs/1.20/design/BankingDesign.html`, 2006.

[11] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, Seattle, WA, USA, 2006.

[12] M. Dahlin, B. B. V. Chandra, L. Gao, and A. Nayate. End-to-end wan service availability. *IEEE/ACM Transactions on Networking*, 11(2), 2003.

[13] J. Dean. Handling large datasets at Google: Current systems and future designs. In Data-Intensive Computing Symposium, Mar. 2008.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, San Francisco, CA, USA, 2004.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[16] A. Fekete. Weak consistency conditions for replicated data. Invited talk at 'A 30-year perspective on replication', Nov. 2007.

[17] A. Fekete, D. Gupta, V. Luchangco, N. Lynch, and A. Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220(1), 1999.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, 2003.

[19] Google. App Engine Outage today. `http://groups.google.com/group/google-appengine/browse_thread/thread/f7ce559b3b8b303b?pli=1`, 2008.

[20] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[21] J. Hamilton. Internet-Scale Service Efficiency. In *Proceedings of 2nd Large-Scale Distributed Systems and Middleware Workshop (LADIS)*, New York, USA, 2008.

[22] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3), 1990.

[23] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[24] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. `http://cs.utexas.edu/~kotla/RESEARCH/CODE/ZYZZYVA/`, 2008.

[25] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, 2004.

[26] J. Li and D. Mazières. Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems. In *Proceedings of USENIX Networked Systems Design and Implementation (NSDI)*, Cambridge, MA, USA, 2007.

[27] D. Malkhi and M. Reiter. Byzantine quorum systems. In *Symposium on Theory of Computing (STOC)*, El Paso, TX, USA, May 1997.

[28] Netflix Blog. Shipping Delay. `http://blog.netflix.com/2008/08/shipping-delay-recap.html`, 2008.

[29] R. Rodrigues, M. Castro, and B. Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Banff, Canada, 2001.

[30] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1), 2005.

[31] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually Consistent Byzantine Fault Tolerance. *MPI-SWS, Technical Report: TR-09-02-01*, 2009.

[32] M. Spreitzer, M. Theimer, K. Petersen, A. J. Demers, and D. B. Terry. Dealing with server corruption in weakly consistent replicated data systems. *Wireless Networks*, 5(5), 1999.

[33] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Cooper Mountain Resort, Colorado, USA, 1995.

[34] F. Travostino and R. Shoup. eBay's Scalability Odyssey: Growing and Evolving a Large eCommerce Site. In *Proceedings of 2nd Large-Scale Distributed Systems and Middleware Workshop (LADIS)*, New York, USA, 2008.

[35] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proceedings of USENIX Operating System Design and Implementation (OSDI)*, Boston, MA, USA, 2002.

[36] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating Byzantine Faults in Database Systems using Commit Barrier Scheduling. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Stevenson, WA, USA, 2007.

[37] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating Agreement from Execution for Byzantine Fault Tolerant Services. In *Proceedings of ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, USA, 2003.

[38] H. Yu and A. Vahdat. Design and Evaluation of a Conit-Based Continuous Consistentcy Model for Replicated Services. *ACM Transactions on Computer Systems*, 20(3), 2002.

SPLAY: Distributed Systems Evaluation Made Simple[*]
(or how to turn ideas into live systems in a breeze)

Lorenzo Leonini[†] Étienne Rivière[‡] Pascal Felber
University of Neuchâtel, Switzerland

Abstract

This paper presents SPLAY*, an integrated system that facilitates the design, deployment and testing of large-scale distributed applications. Unlike existing systems,* SPLAY *covers all aspects of the development and evaluation chain. It allows developers to express algorithms in a concise, simple language that highly resembles pseudo-code found in research papers. The execution environment has low overheads and footprint, and provides a comprehensive set of libraries for common distributed systems operations.* SPLAY *applications are run by a set of daemons distributed on one or several testbeds. They execute in a sandboxed environment that shields the host system and enables* SPLAY *to also be used on non-dedicated platforms, in addition to classical testbeds like PlanetLab or ModelNet. A controller manages applications, offering multi-criterion resource selection, deployment control, and churn management by reproducing the system's dynamics from traces or synthetic descriptions.* SPLAY*'s features, usefulness, performance and scalability are evaluated using deployment of representative experiments on PlanetLab and ModelNet clusters.*

1 Introduction

Developing large-scale distributed applications is a highly complex, time-consuming and error-prone task. One of the main difficulties stems from the lack of appropriate tool sets for quickly prototyping, deploying and evaluating algorithms in real settings, when facing unpredictable communication and failure patterns. Nonetheless, evaluation of distributed systems over real testbeds is highly desirable, as it is quite common to discover discrepancies between the expected behavior of an application as modeled or simulated and its actual behavior when deployed in a live network.

While there exist a number of experimental testbeds to address this demand (e.g., PlanetLab [11], ModelNet [35], or Emulab [38]), they are unfortunately not used as systematically as they should. Indeed, our first-hand experience has convinced us that it is far from straightforward to develop, deploy, execute and monitor applications for them and the learning curve is usually slow. Technical difficulties are even higher when one wants to deploy an application on several testbeds, as deployment scripts written for one testbed may not be directly usable for another, e.g., between PlanetLab and ModelNet. As a side effect of these difficulties, the performance of an application can be greatly impacted by the technical quality of its implementation and the skills of the person who deploys it, overshadowing features of the underlying algorithms and making comparisons potentially unsound or irrelevant. More dramatically, the complexity of using existing testbeds discourages researchers, teachers, or more generally systems practitioners from fully exploiting these technologies.

These various factors outline the need for novel development-deployment systems that would straightforwardly exploit existing testbeds and bridge the gap between algorithmic specifications and live systems. For researchers, such a system would significantly shorten the delay experienced when moving from simulation to evaluation of large-scale distributed systems ("time-to-paper" gap). Teachers would use it to focus their lab work on the core of distributed programming—algorithms and protocols—and let students experience distributed systems implementation in real settings with little effort. Practitioners could easily validate their applications in the most adverse conditions.

There already exist several systems to ease the development or deployment process of distributed applications. Tools like Mace [23] or P2 [26] assist the developer by generating code from a high-level description, but do not provide any facility for its deployment or evaluation. Tools such as Plush [9] or Weevil [37] help for the deployment process, but are restricted to situations where the user has control over the nodes composing the testbed (i.e., the ability to run programs remotely using `ssh` or similar).

To address these limitations, we propose SPLAY, an infrastructure that simplifies the prototyping, development, deployment and evaluation of large-scale systems. Unlike existing tools, SPLAY covers the whole chain of distributed systems design and evaluation. It allows developers to specify distributed applications in a concise manner using a platform-independent, lightweight and efficient language based on Lua [20]. For instance, a complete implementation of the Chord [33] distributed hash table (DHT) requires approximately 100 lines of code.

SPLAY provides a secure and safe environment for executing and monitoring applications, and allows for a simplified and unified usage of testbeds such as PlanetLab, ModelNet, networks of idle workstations, or per-

[*]This work is supported in part by the Swiss National Foundation under agreement number 102819.

[†]Contact author: `lorenzo.leonini@unine.ch`

[‡]This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme.

sonal computers. SPLAY applications execute in a safe, sandboxed environment with controlled access to local resources (file system, network, memory) and can be instantiated on a large set of nodes with a single command. SPLAY supports multi-user resource reservation and selection, orchestrates the deployment and monitors the whole system. It is particularly easy with SPLAY to reproduce a given live experiment or to control several experiments at the same time.

An important component of SPLAY is its churn manager, which can reproduce the dynamics of a distributed system based on real traces or synthetic descriptions. This aspect is of paramount importance, as natural churn present in some testbeds such as PlanetLab is not reproducible, hence preventing a fair comparison of protocols under the very same conditions.

SPLAY is designed for a broad range of usages, including: (i) deploying distributed systems whose lifetime is specified at runtime and usually short, e.g., distributing a large file using BitTorrent [17]; (ii) executing long-running applications, such as an indexing service based on a DHT or a cooperative web cache, for which the population of nodes may dynamically evolve during the lifetime of the system (and where failed nodes must be replaced automatically); or (iii) experimenting with distributed algorithms, e.g., in the context of hands-on networking class, by leveraging the isolation properties of SPLAY to enable execution of (possibly buggy) code on a shared testbed without interference.

Contributions. This paper introduces a distributed infrastructure that greatly simplifies the prototyping, development, deployment, and execution of large-scale distributed systems and applications. SPLAY includes several original features—notably churn management, support for mixed deployments, and platform-independent language and libraries—that make the evaluation and comparison of distributed systems much easier and fairer than with existing tools.

We show how SPLAY applications can be concisely expressed with a specialized language that closely resembles the pseudo-code usually found in research papers. We have implemented several well-known systems: Chord [33], Pastry [31], Scribe [15], SplitStream [14], BitTorrent [17], Cyclon [36], Erdös-Renyi epidemic broadcast [19] and various types of distribution trees [13].

Our system has been thoroughly evaluated along all its aspects: conciseness and ease of development, efficiency, scalability, stability and features. Experiments convey SPLAY's good properties and the ability of the system to help practitioner and researcher alike through the whole distributed system design, implementation and evaluation chain.

Roadmap. The remaining of this paper is organized as follows. We first discuss related work in Section 2. Section 3 gives an overview of the SPLAY architecture and elaborates on its design choices and rationales. In Section 4, we illustrate the development process of a complete application (the Chord DHT [33]). Section 5 presents a complete evaluation of SPLAY, using representative experiments and deployments (including tests of the Chord implementation of Section 4). Finally, we conclude in Section 6.

2 Related Work

SPLAY shares similarities with a large body of work in the area of concurrent and distributed systems. We only present systems that are closely related to our approach.

Development tools. On the one hand, a set of new languages and libraries have been proposed to ease and speed up the development process of distributed applications.

Mace [23] is a toolkit that provides a wide set of tools and libraries to develop distributed applications using an event-driven approach. Mace defines a grammar to specify finite state machines, which are then compiled to C++ code, implementing the event loop, timers, state transitions, and message handling. The generated code is platform-dependent: this can prove to be a constraint in heterogeneous environments. Mace focuses on application development and provides good performance results but it does not provide any built-in facility for deploying or observing the generated distributed application.

P2 [26] uses a declarative logic language named OverLog to express overlays in a compact form by specifying data flows between nodes, using logical rules. While the resulting overlay descriptions are very succinct, specifications in P2 are not natural to most network programmers (programs are largely composed of table declaration statements and rules) and produce applications that are not very efficient. Similarly to Mace, P2 does not provide any support for deploying or monitoring applications: the user has to write his/her own scripts and tools.

Other domain-specific languages have been proposed for distributed systems development. In RTAG [10], protocols are specified as a context-free grammar. Incoming messages trigger reduction of the rules, which express the sequence of events allowed by the protocol. Morpheus [8] and Prolac [24] target network protocols development. All these systems share the goal of SPLAY to provide easily readable yet efficient implementations, but are restricted to developing low-level network protocols, while SPLAY targets a broader range of distributed systems.

Deployment tools. On the other hand, several tools have been proposed to provide runtime facilities for distributed applications developers by easing the deployment and monitoring phase.

Neko [34] is a set of libraries that abstract the network substrate for Java programs. A program that uses Neko can be executed without modifications either in simulations or in a real network, similarly to the NEST testbed [18]. Neko addresses simple deployment issues, by using daemons on distant nodes to launch the virtual machines (JVMs). Nonetheless, Neko's network library has been designed for simplicity rather than efficiency (as a result of using Java's RMI), provides no isolation of deployed programs, and does not have built-in support for monitoring. This restricts its usage to controlled settings and small-scale experiments.

Plush [9] is a set of tools for automatic deployment and monitoring of applications on large-scale testbeds such as PlanetLab [11]. Applications can be remotely compiled from source code on the target nodes. Similarly to Neko and SPLAY, Plush uses a set of application controllers (daemons) that run on each node of the system, and a centralized controller is responsible for managing the execution of the distributed application.

Along the same lines, Weevil [37] automates the creation of deployment scripts. A set of models is provided by the user to describe the experiment. An interesting feature of Weevil lies in its ability to replay a distributed workload (such as a set of request for a distributed middleware infrastructure). These inputs can either be synthetically generated, or recorded from a previous run or simulation. The deployment phase does not include any node selection mechanism: the set of nodes and the mapping of application instances to these nodes must be provided by the user. The created scripts allow deployment and removal of the application, as well as the retrieval of outputs at the end of an experiment.

Plush and Weevil share a set of limitations that make them unsuitable for our goals. First, and most importantly, these systems propose high-end features for experienced users on experimental platforms such as PlanetLab, but cannot provide resource isolation due to their script-based nature. This restricts their usage to controlled testbeds, i.e., platforms on which the user has been granted some access rights, as opposed to non-dedicated environments such as networks of idle workstations where it might not be desirable or possible to create accounts on the machines, and where the nature of the testbed imposes to restrict the usage of their resources (e.g., disk or network usage).Second, they do not provide any management of the dynamics (churn) of the system, despite its recognized usefulness [29] for distributed system evaluation.

Testbeds. A set of experimental platforms, hereafter denoted as *testbeds*, have been built and proposed to the community. These testbeds are complementary to the languages and deployment systems presented in the first part of this section: they are the *medium* on which these tools operate.

Distributed simulation platforms such as WiDS [25] allow developers to run their application on top of an event-based network simulation layer. Distributed simulation is known to scale poorly, due to the high load of synchronization between nodes of the testbed hosting communicating processes. WiDS alleviates this limitation by relaxing the synchronization model between processes on distinct nodes. Nonetheless, event-based simulation testbeds such as WiDS do not provide mechanisms to deploy or manage the distributed application under test.

Network emulators such as Emulab [38], ModelNet [35], FlexLab [30] or P2PLab [28] can reproduce some of the characteristics of a networked environment: delays, bandwidth, packet drops, etc. They basically allow users to evaluate unmodified applications across various network models. Applications are typically deployed in a local-area cluster and all communications are routed through some proxy node(s), which emulate the topology. Each machine in the cluster can host several end-nodes from the emulated topology.

The PlanetLab [11] testbed (and forks such as Everlab [22]) allows experimenting in live networks by hosting applications on a large set of geographically dispersed hosts. It is a very valuable infrastructure for testing distributed applications in the most adverse conditions.

SPLAY is designed to complement these systems. Testbeds are useful but, often, complex platforms. They require the user to know how to deploy applications, to have a good understanding of the target topology, and to be able to properly configure the environment for executing his/her application (for instance, one needs to use a specific library to override the IP address used by the application in a ModelNet cluster). In PlanetLab, it is time-consuming and error-prone to choose a set of non-overloaded nodes on which to test the application, to deploy and launch the program, and to retrieve the results. Finally, considering mixed deployments that use several testbeds at the same time for a single experiment would require to write even much more complex scripts (e.g., taking into account problems such as port range forwarding). With SPLAY, as soon as the administrator who deployed the infrastructure has set up the network, using a complex testbed is as straightforward for the user as running an application on a local machine.

3 The SPLAY Framework

We present the architecture of our system: its main components, its programming language, libraries and tools.

3.1 Architecture

The SPLAY framework consists of about 15,000 lines of code written in C, Lua, Ruby, and SQL, plus some third-party support libraries. Roughly speaking, the architecture is made of three major components. These components are depicted in Figure 1.

- The **controller**, `splayctl`, is a trusted entity that controls the deployment and execution of applications.
- A lightweight **daemon** process, `splayd`, runs on every machine of the testbed. A `splayd` instantiates, stops, and monitors SPLAY applications when instructed by the controller.
- SPLAY **applications** execute in sandboxed processes forked by `splayd` daemons on participating hosts.

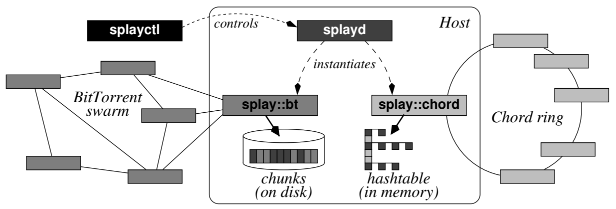


Figure 1: An illustration of two SPLAY applications (BitTorrent and Chord) at runtime.

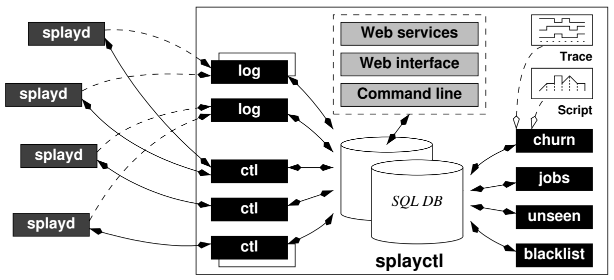


Figure 2: Architecture of the SPLAY controller (note that all components may be distributed on different machines).

Many SPLAY applications can run simultaneously on the same host. The testbed can be used transparently by multiple users deploying different applications on overlapping sets of nodes, unless the controller has been configured for a single-user testbed. Two SPLAY applications on the same node are unaware of each other (they cannot even exchange data via the file system); they can only communicate by message passing as for remote processes. Figure 1 illustrates the deployment of multiple applications with a host participating to both a Chord DHT and a BitTorrent swarm.

An important point is that SPLAY applications can be run locally with no modification to their code, while still using all libraries and language features proposed by SPLAY. Users can simply and quickly debug and test their programs locally, prior to deployment.

We now discuss in more details the different components of the SPLAY architecture.

Controller. The controller plays an essential role in our system. It is implemented as a set of cooperating processes and executes on one or several trusted servers. The only central component is a database that stores all data pertaining to participating hosts and applications.

The controller (see Figure 2) keeps track of all active SPLAY daemons and applications in the system. Upon startup, a daemon initiates a secure connection (SSL) to a `ctl` process. For scalability reasons, there can be many `ctl` processes spread across several trusted hosts. These processes only need to access the shared database.

SPLAY daemons open connections to `log` processes on behalf of the applications, if the logging library is used. This library is described in section 3.4.

The deployment of a distributed application is achieved by submitting a job through a command-line or Web-based interface. SPLAY also provides a Web services API that can be used by other projects. Once registered in the database, jobs are handled by `jobs` processes. The nodes participating in the deployment can be specified explicitly as a list of hosts, or one can simply indicate the number of nodes on which deployment has to take place, regardless of their identity. One can also specify requirements in terms of resources that must be available at the participating nodes (e.g., bandwidth) or in terms of geographical location (e.g., nodes in a specific country or within a given distance from a position). Incremental deployment, i.e., adding nodes at different times, can be performed using several jobs or with the churn manager.

Each daemon is associated with records in the database that store information about the applications and active hosts running them, or scheduled for execution. The controller monitors the daemons and uses a session mechanism to tolerate short-term disconnections (i.e., a daemon is considered alive if it shows activity at least once during a given time period). Only after a long-term disconnection (typically one hour) does the controller reset the status of the daemon and clean up the associated entries in the database. This task is under the responsibility of the `unseen` process. The `blacklist` process manages in the database a list of forbidden network addresses and masks; it piggybacks updates of this list onto messages sent to connected daemons.

Communication between the daemon and the controller follows a simple request/answer protocol. The first request originates from the daemon that connects to the controller. Every subsequent command comes from the controller. For brevity, we only present here a minimal set of commands.

The `jobs` process dequeues jobs from the database and searches for a set of hosts matching the constraints specified by the user. The controller sends a `REGISTER` message to the daemons of every selected node. In case the identity of the nodes is not explicitly specified, the system selects a set larger than the one originally requested to account for failed or overloaded nodes. Upon accepting the job, a daemon sends to the controller the range of ports that are available to the application. Once it receives enough replies, the controller first sends to every selected daemon a `LIST` message with the addresses of some participating nodes (e.g., a single *rendez-vous* node or a random subset , depending on the application) to bootstrap the application, followed by a `START` message to begin execution. Supernumerary daemons that are slow to answer and active applications that must be terminated receive a `FREE` message. The state machine of a SPLAY job is as follows:

$$\begin{tikzcd}
\textit{idle} \arrow[r, bend left, "\texttt{REGISTER}"] & \textit{selected} \arrow[l, bend left, "\texttt{FREE}"] \arrow[loop above, "\texttt{LIST}"] \arrow[r, bend left, "\texttt{START}"] & \textit{running} \arrow[l, bend left, "\texttt{STOP}"] \arrow[ll, bend left=50, "\texttt{FREE}"]
\end{tikzcd}$$

The reason why we initially select a larger set of nodes than requested clearly appears when considering the availability of hosts on testbeds like PlanetLab, where transient failures and overloads are the norm rather than the exception. Figure 3 shows both the cumulative and discretized distributions of round-trip times (RTT) for a 20 KB message over an already established TCP connection from the controller to PlanetLab hosts. One can observe that only 17.10% of the nodes reply within 250 milliseconds, and over 45% need more than 1 second. Selecting a larger set of candidates allows us to choose the most responsive nodes for deploying the application.

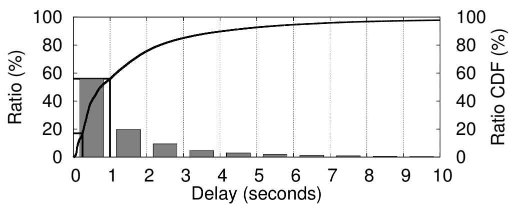


Figure 3: RTT between the controller and PlanetLab hosts over pre-established TCP connections, with a 20 KB payload.

Daemons. SPLAY daemons are installed on participating hosts by a local user or administrator. The local administrator can configure the daemon via a configuration file, specifying various instance parameters (e.g., daemon name, access key, etc.) and restrictions on the resources available for SPLAY applications. These restrictions encompass memory, network, and disk usage. If an application exceeds these limitations, it is killed (memory usage) or I/O operations fail (disk or network usage). The controller can specify stricter—but not weaker—restrictions at deployment time.

Upon startup, a SPLAY daemon receives a blacklist of forbidden addresses expressed as IP or DNS masks. By default, the addresses of the controllers are blacklisted so that applications cannot actively connect to them. Blacklists can be updated by the controller at runtime (e.g., when adding a new daemon or for protecting a particular machine).

The daemon also receives the address of a `log` process to connect to for logging, together with a unique identification key. SPLAY applications instantiated by the local daemon can only connect to that log process; other processes will reject any connection request.

3.2 Churn Management

In order to fully understand the behavior and robustness of a distributed protocol, it is necessary to evaluate it under different churn conditions. Theses conditions can range from rare but unpredictable hardware failures, to frequent application-level disconnections, as usually found in user-driven peer-to-peer systems, or even to massive failures scenarios. It is also important to allow comparison of competing algorithms under the very same churn scenarios. Relying on the natural, non-reproducible churn of testbeds such as PlanetLab often proves to be insufficient.

There exist several characterizations of churn that can be leveraged to reproduce realistic conditions for the protocol under test. First, synthetic descriptions issued from analytical studies [27] can be used to generate churn scenarios and replay them in the system. Second, several traces of the dynamics of real networks have been made publicly available by the community (e.g., see the repository at [1]); they cover a wide range of applications such as a highly churned file-sharing system [12] or high-performance computing clusters [32].

```
1  at 30s join 10
2  from 5m to 10m inc 10
3  from 10m to 15m const
       churn 50%
4  at 15m leave 50%
5  from 15m to 20m inc 10
       churn 150%
6  at 20m stop
```

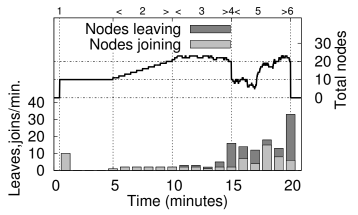


Figure 4: Example of a synthetic churn description: script (left), binned number of joins/leave (right, bottom) and total number of nodes (right, top).

SPLAY incorporates a component, `churn` (see Figure 2), dedicated to churn management. This component can send instructions to the daemons for stopping and starting processes on-the-fly. Churn can be specified as a trace, in a format similar to that used by [1], or as a synthetic description written in a simple script language. The trace indicates explicitly when each node enters or leaves the system while the script allows users to express phases of the application's lifetime, such as a steady increase or decrease of the number of peers over a given time duration, periods with continuous churn, massive failures, join flash crowds, etc. An example script is shown in Figure 4 together with a representation of the evolution of the node population and the number of arrivals and departures during each one-minute period: an initial set of nodes joins after 30 seconds, then the system stabilizes before a regular increase, a period with a constant population but a churn that sees half of the nodes leave and an equal number join, a massive failure of half of the nodes, another increase under high churn, and finally the departure of all the nodes.

Section 5.5 presents typical uses of the churn management mechanism in the evaluation of a large-scale distributed system. It is noteworthy that the churn management system relieves the need for fault injection systems such as Loki [16]. Another typical use of the churn management system is for long-running applications, e.g., a DHT that serves as a substrate for some other distributed application under test and needs to stay available for the whole duration of the experiments. In such a scenario, one can ask the churn manager to maintain a fixed-size population of nodes and to automatically bootstrap new ones as faults occur in the testbed.

3.3 Language and Applications

SPLAY applications are written in the Lua language [20], whose features are extended by SPLAY's libraries. This design choice was dictated by four majors factors. First, the most important reason is that Lua has unique features that allow to simply and efficiently implement sandboxing. As mentioned earlier, sandboxing is a sound basis for execution in non-dedicated environments, where resources need to be constrained and where the hosting operating system must be shielded from possibly buggy or ill-behaved code. Second, one of SPLAY's goals is to support large numbers of processes within a single host of the testbed. This calls for a low footprint for both the daemons and the associated libraries. This excludes languages such as Java that require several megabytes of memory just for their execution environment. Third, SPLAY must ensure that the achieved performance is as good as the host system permits, and the features offered to the distributed system designer shall not interfere with the performance of the application. Fourth, SPLAY allows deployment of applications on any hardware and on any operating systems. This requires a "write-once, run everywhere" approach that calls for either an interpreted or bytecode-based language. Lua's unique features allow us to meet these goals of lightness, simplicity, performance, security and generality.

Lua was designed from the ground up to be an efficient scripting language with very low footprint. According to recent benchmarks [2], Lua is among the fastest interpreted scripting languages. It is reflective, imperative, and procedural with extensible semantics. Lua is dynamically typed and has automatic memory management with incremental garbage collection. The small footprint from Lua results from its design that provides flexible and extensible meta-features, rather than a complete set of general-purpose facilities. The full interpreter is less than 200 kB and can be easily embedded. Applications can use libraries written in different languages (especially C/C++). This allows for low-level programming if need be. Our experiments (Section 5) highlight the lightness of SPLAY applications using Lua, in terms of memory footprint, load, and scalability.

Lua's interpreter can directly execute source code, as well as hardware-dependent (but operating system-independent) bytecode. In SPLAY, the favored way of submitting applications is in the form of source code, but bytecode programs are also supported (e.g., for intellectual property protection).

Isolation and sandboxing are achieved thanks to Lua's support for first-class functions with lexical scoping and closures, which allow us to restrict access to I/O and networking libraries. We modify the behavior of these functions to implement the restrictions imposed by the administrator or by the user at the time he/she submits the application for deployment over SPLAY.

Lua also supports cooperative multitasking by the means of coroutines, which are at the core of SPLAY's event-based model (discussed below).

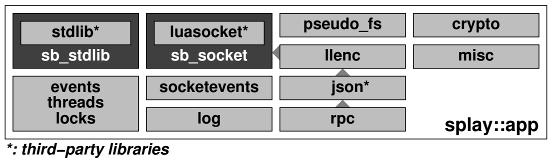


Figure 5: Overview of the main SPLAY libraries.

3.4 The Libraries

SPLAY includes an extensible set of shared libraries (see Figure 5) tailored for the development of distributed applications and overlays. These libraries are meant to be also used outside of the deployment system, when developing the application. We briefly describe the major components of these libraries.

Networking. The `luasocket` library provides basic networking facilities. We have wrapped it into a restricted socket library, `sb_socket`, which includes a security layer that can be controlled by the local administrator (the person who has instantiated the local daemon process) and further restricted remotely by the controller. This secure layer allows us to limit: (1) the total bandwidth available for SPLAY applications (instantaneous bandwidth can be limited using shaping tools if need be); (2) the maximum number of sockets used by an application and (3) the addresses that an application can or cannot connect to. Restrictions are specified declaratively in configuration files by the local user that starts the daemon, or at the controller via the command-line and Web-based APIs.

We have implemented higher-level abstractions for simplifying communication between remote processes. Our API supports message passing over TCP and UDP, as well as access to remote function and variables using RPCs. Calling a remote function is almost as simple as calling a local one (see code in next section). All arguments and return values are transparently serialized. Communication errors are reported using a second return value, as allowed by Lua.

Finally, communication libraries can be instructed to drop a given proportion of the packets (specified upon deployment): this can be used to simulate lossy links and study their impact on an application.

Sandboxed virtual filesystem. Overlays and distributed applications often need to use the local file system. For instance, when instantiating the BitTorrent protocol to replicate a large file on a set of nodes, temporary data must be written to disk as chunks are being received. Following our goal to not impact the hosting operating system, we need to ensure that a SPLAY application cannot access or overwrite any data on the host file system. To this end, SPLAY includes a library, `sb_fs`, that wraps the standard `io` library and provides restricted access to the file system in an OS-independent fashion.

Our wrapped library simulates a file system inside a single directory. The library transparently maps a complete path name to the underlying files that stores the actual data, and applications can only read the files located in their private directory. The wrapped file handles enforce additional restrictions, such as limitations on the disk space and the number of opened files.

Events, threads and locks. SPLAY proposes a threading model based on Lua's coroutines combined with event-based programming. Unlike preemptive threads, coroutines yield the processor to each other (cooperative multitasking). This happens at special points in base libraries, typically when performing an operation that may block (e.g., disk or network I/O). This is typically transparent to the application developer. Although a *single* SPLAY application will not benefit from a multicore processor, coroutines are preferable to system-level threads for two reasons: their portability and their recognized efficiency (low latency and high throughput) for programs that use many network connections (using either non-blocking or RPC-based programming), which is typical of distributed systems programming. Moreover, using a single process (at the operating system level) has a lower footprint, especially from a sandboxing perspective, and allows deploying more applications on each `splayd`.

Shared data accesses are also safer with coroutines, as race conditions can only occur if the current thread yields the processor. This requires, however, a good understanding of the behavior of the application (we illustrate a common pitfall in Section 4). SPLAY provides a lock library as a simple alternative to protect shared data from concurrent accesses by multiple coroutines.

We have also developed an event library, `events`, that controls the main execution loop of the application, the scheduler, the communication between coroutines, timeouts, as well as event generation, waiting, and reception. To integrate with the event library, we have wrapped the socket library to produce a non-blocking, coroutine-aware version `sb_socket`. All these layers are transparent to the SPLAY developer who only sees a restricted, non-blocking socket library.

Logging. An important objective of SPLAY is to be able to quickly prototype and experiment with distributed algorithms. To that end, one must be able to easily debug and collect statistics about the SPLAY application at runtime. The `log` library allows the developer to print information either locally (screen, file) or, more interestingly, send it over the network to a log collector managed by the controller. If need be, the amount of data sent to the log collector can be restricted by a `splayd`, as instructed by the controller. As with most log libraries, facilities are provided to manage different log levels and dynamically enable or disable logging.

Other libraries. SPLAY provides a few other libraries with facilities useful for developing distributed systems and applications. The `llenc` and `json` libraries [3] support automatic and efficient serialization of data to be sent to remote nodes over the network. We developed the first one, `llenc`, to simplify message passing over stream-oriented protocols (e.g., TCP). The library automatically performs message demarcation, computing buffer sizes and waiting for all packets of a message before delivery. It uses the `json` library to automate encoding of any type of data structures using a compact and standardized data-interchange format. The `crypto` library includes cryptographic functions for data encryption and decryption, secure hashing, signatures, etc. The `misc` library provides common containers, functions for format conversion, bit manipulation, high-precision timers and distributed synchronization.

The memory footprint of these libraries is remarkably small. The base size of a SPLAY application is less than 600 kB with all the abovementioned libraries loaded. It is easy for administrators to deploy additional third-party software with the daemons, in the form of libraries. Lua has been design to seamlessly interact with C/C++, and other languages that bind to C can be used as well. For instance, we successfully linked some Splay application code with a third-party video transcoding library in C, for experimenting with adaptive video multicast. Obviously, the administrator is responsible for providing sandboxing in these libraries if required.

4 Developing Applications with SPLAY

This section illustrates the development of an application for SPLAY. We use the well-known Chord overlay [33] for its familiarity to the community. As we will see, the specification of this overlay is remarkably concise and close to the pseudo-code found in the original paper. We have successfully deployed this implementation on a ModelNet cluster and PlanetLab; results are presented in Section 5.2. The goal here is to provide the reader with a complete chain of development, deployment, and monitoring of a well-known distributed application. Note that local testing and debugging is generally done outside of the deployment framework (but still, using SPLAY libraries).

Chord is a distributed hash table (DHT) that maps keys to nodes in a peer-to-peer infrastructure. Any node can use the DHT substrate to determine the current live node that is responsible for a given key. When joining the network, a node receives a unique identifier (typically by hashing its IP address and port number) that determines its position in the identifier space. Nodes are organized in a ring according to their identifiers, and every node is responsible for the keys that fall between itself (inclusive) and its predecessor (exclusive). In addition to keeping track of their successors and predecessors on the ring, each node maintains a "finger" table whose entries point to nodes at an exponentially increasing distance from the current node's position. More precisely, the i^{th} entry of a node with identifier n designates the live node responsible for key $n + 2^i$. Note that the successor is effectively the first entry in the finger table.

```
 4 function join(n0)                                      -- n0: some node in the ring
 5   predecessor = nil
 6   finger[1] = call(n0, {'find_successor', n.id})
 7   call(finger[1], {'notify', n})
 8 end
 9 function stabilize()                          -- periodically verify n's successor
10   local x = call(finger[1], 'predecessor')
11   if x and between(x.id, n.id, finger[1].id, false, false) then
12     finger[1] = x                                                -- new successor
13   end
14   call(finger[1], {'notify', n})
15 end
16 function notify(n0)                       -- n0 thinks it might be our predecessor
17   if not predecessor or between(n0.id, predecessor.id, n.id, false, false) then
18     predecessor = n0                                           -- new predecessor
19   end
20 end
21 function fix_fingers()                                         -- refresh fingers
22   refresh = (refresh % m) + 1                                  -- 1 ≤ refresh ≤ m
23   finger[refresh] = find_successor((n.id + 2^(refresh − 1)) % 2^m)
24 end
25 function check_predecessor()                   -- checks if predecessor has failed
26   if predecessor and not ping(predecessor) then
27     predecessor = nil
28   end
29 end
```

Listing 1: SPLAY code for Chord overlay (stabilization).

Listing 1 shows the code for the construction and maintenance of the Chord overlay. For clarity, we only show here the basic algorithm that was proposed in [33] (the reader can appreciate the similarity between this code and Figure 6 of the referenced paper).

Function `join()` allows a node to join the Chord ring. Only its successor is set: its predecessor and successor's predecessor will be updated as part of the stabilization process. Function `stabilize()` periodically verifies that a node is its own successor's predecessor and notifies the successor. SPLAY base library's `between` call determines the inclusion of a value in a range, on a ring. Function `notify()` tells a node that its predecessor might be incorrect. Function `fix_fingers()` iteratively refreshes fingers. Finally, function `check_predecessor()` periodically checks if a node's predecessor has failed.

These functions are identical in their behavior and very similar in their form to those published in [33]. Yet, they correspond to executable code that can be readily deployed. The implementation of Chord illustrates a subtle problem that occurs frequently when developing distributed applications from a high-level pseudo-code description: the reception of multiple messages may trigger concurrent operations that perform conflicting modifications on the state of the node. SPLAY's coroutine model alleviates this problem in some, but not all, situations. During the blocking call to `ping()` on line 26 of Listing 1, a remote call to `notify()` can update the predecessor, which may be erased on line 27 until the next remote call to `notify()`. This is not a major issue as it may only delay stabilization, not break consistency. It can be avoided by adding an extra check after the ping or, more generally, by using the locks provided by the SPLAY standard libraries (not shown here).

```
30 function find_successor(id)                       -- ask node to find id's successor
31   if between(id, n.id, finger[1].id, false, true)   -- inclusive for second bound
32     return finger[1]
33   end
34   local n0 = closest_preceding_node(id)
35   return call(n0, {'find_successor', id})
36 end
37 function closest_preceding_node(id)                          -- finger preceding id
38   for i = m, 1, −1 do
39     if finger[i] and between(finger[i].id, n.id, id, false, false) then
40       return finger[i]
41     end
42   end
43   return n
44 end
```

Listing 2: SPLAY code for Chord overlay (lookup).

Listing 2 shows the code for Chord lookup. Function `find_successor()` looks for the successor of a given identifier, while function `closest_preceding_node()` returns the highest predecessor of a given identifier found in the finger table. Again, one can appreciate the similarity with the original pseudo-code.

This almost completes our minimal Chord implementation, with the exception of the initialization code shown in Listing 3. One can specifically note the registration of periodic stabilization tasks and the invocation of the main event loop.

```
 1 require "splay.base"                          -- events, misc, socket (core libraries)
 2 rpc = require "splay.rpc"                                     -- rpc (optional library)
 3 between, call, ping = misc.between_c, rpc.call, rpc.ping                     -- aliases

45 timeout = 5                                                   -- stabilization frequency
46 m = 24                               -- 2^m nodes and key with identifiers of length m
47 n = job.me                                                    -- our node {ip, port, id}
48 n.id = math.random(1, 2^m)                                  -- random position on ring
49 predecessor = nil                                -- previous node on ring {id, ip, port}
50 finger = {[1] = n}                                      -- finger table with m entries
51 refresh = 0                                                    -- next finger to refresh
52 n0 = job.nodes[1]                                         -- first peer is rendez−vous node
53 rpc.server(n.port)                                                  -- start rpc server
54 events.thread(function() join(n0) end)                              -- join chord ring
55 events.periodic(stabilize, timeout)                      -- periodically check successor, ...
56 events.periodic(check_predecessor, timeout)                          -- predecessor, ...
57 events.periodic(fix_fingers, timeout)                                    -- and fingers
58 events.loop()                                                       -- execute main loop
```

Listing 3: SPLAY code for Chord overlay (initialization).

While this code is quite classical in its form, the remarkable features are the conciseness of the implementation, the closeness to pseudo-code, and the ease with which one can communicate with other nodes of the system by RPC. Of course, most of the complexity is hidden inside the SPLAY infrastructure.

The presented implementation is not fault-tolerant. Although the goal of this paper is not to present the design of a fault-tolerant Chord, we briefly elaborate below on some steps needed to make Chord robust enough for running on error-prone platforms such as PlanetLab. The first step is to take into account the absence of a reply to an RPC. Consider the call to `predecessor` in method `stabilize()`. One simply needs to replace this call by the code of Figure 4.

```
function stabilize()                      -- rpc.a_call() returns both status and results
  local ok, x = rpc.a_call(finger[1], 'predecessor', 60)    -- RPC, 1m timeout
  if not ok then
    suspect(finger[1])          -- will prune the node out of local routing tables
  else
    (...)
```

Listing 4: Fault-tolerant RPC call

We omit the code of function `suspect()` for brevity. Depending on the reliability of the links, this function prunes the suspected node after a configurable number of missed replies. One can tune the RPC timeout according to the target platform (here, 1 minute instead of the standard 2 minutes), or use an adaptive strategy (e.g., exponentially increasing timeouts). Finally, as suggested by [33] and similarly to the leafset structure used in Pastry [31], we replace the single successor and predecessor by a list of 4 peers in each direction on the ring.

Our Chord implementation without fault-tolerance is only 58 lines long, which represents an increase of 18% over the pseudo-code from the original paper (which does not contain initialization code, while our code does). Our fault-tolerant version is only 100 lines long, i.e., 73% more than the base implementation (29% for fault tolerance, and 44% for the leafset-like structure). We detail the procedure for deployment and the results obtained with both versions on a ModelNet cluster and on PlanetLab, respectively, in Section 5.2.

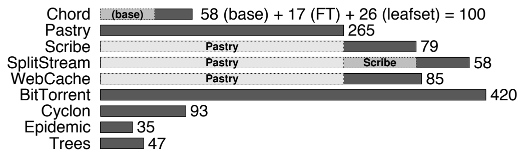
5 Evaluation

This section presents a thorough evaluation of SPLAY performance and capabilities. Evaluating such an infrastructure is a challenging task as the way users will use it plays an important role. Therefore, our goal in this evaluation is twofold: (1) to present the implementation, deployment and observation of real distributed systems by using SPLAY's capability to easily reproduce experiments that are commonly used in evaluations and (2) to study the performance of SPLAY itself, both by comparing it to other widely-used implementations and by evaluating its costs and scalability. The overall objective is to demonstrate the usefulness and benefits of SPLAY rather than evaluate the distributed applications themselves. We first demonstrate in Section 5.1 SPLAY's capabilities to easily express complex system in a concise manner. We present in Section 5.2 the deployment and performance evaluation of the Chord DHT proposed in Section 4, using a ModelNet [35] cluster and PlanetLab [11]. We then compare in Section 5.3 the performance and scalability of the Pastry [31] DHT written with SPLAY against a legacy Java implementation, FreePastry [4]. Sections 5.4 and 5.5 evaluate SPLAY's ability to easily (1) deploy applications in complex network settings (mixed PlanetLab and ModelNet deployment) and (2) reproduce arbitrary churn conditions. Section 5.6 focuses on SPLAY performance for deploying and undeploying applications on a testbed. We conclude in Section 5.7 with an evaluation of SPLAY's performance with resource-intensive applications (tree-based content dissemination and long-term running of a cooperative Web cache).

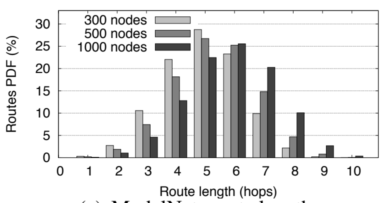
Experimental setup. Unless specified otherwise, our experimentations were performed either on PlanetLab, using a set of 400 to 450 hosts, or on our local cluster (11 nodes, each equipped with a 2.13 Ghz Core 2 Duo processor and 2 GB of memory, linked by a 1 Gbps switched network). All nodes run GNU/Linux 2.6.9. A separate node running FreeBSD 4.11 is used as a ModelNet router, when required by the experiment. Our ModelNet configuration emulates 1,100 hosts connected to a 500-node transit-stub topology. The bandwidth is set to 10Mbps for all links. RTT between nodes of the same domain is 10 ms, stub-stub and stub-transit RTT is 30 ms, and transit-transit (i.e., long range links) RTT is 100 ms. These settings result in delays that are approximately twice those experienced in PlanetLab.

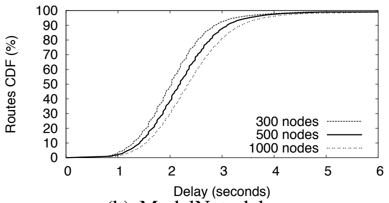
5.1 Development complexity

We developed the following applications using SPLAY: Chord [33] and Pastry [31], two DHTs; Scribe [15], a publish-subscribe system; SplitStream [14], a bandwidth-intensive multicast protocol; a cooperative web-cache based on Pastry; BitTorrent [17], a content distribution infrastructure;[1] and Cyclon [36], a gossip-based membership management protocol. We have also implemented a number of classical algorithms, such as epidemic diffusion on Erdös-Renyi random graphs [19] and various types of distribution trees [13] (n-ary trees, parallel trees). As one can note from the following figure, all implementations are extremely concise in terms of lines of code (LOC). Note that we did not try to compact the code in a way that would impair readability. Numbers and darker bars represent LOC for the protocol, while lighter bars represent protocols acting as a substrate (Scribe and our Web cache are based on Pastry, SplitStream is based on both Pastry and Scribe):



Although the number of lines is clearly just a rough indicator of the expressiveness of a system, it is still a valuable metric to estimate programming efforts. Our implementations are systematically more compact than those written with Mace [23] (by approximately a factor of two) and comparable to P2's [26] specifications. A well-documented protocol such as Chord only took a few hours to implement and debug. In contract, BitTorrent, being a complex and underspecified protocol, required several days of development. In both cases, the development process greatly benefited from the short deployment and testing phase, made almost trivial by SPLAY.


(a) ModelNet: route lengths


(b) ModelNet: delays

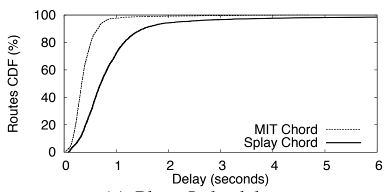

(c) PlanetLab: delays

Figure 6: Performance results of Chord, deployed on a ModelNet cluster and on PlanetLab.

5.2 Testing the Chord Implementation

This section presents the deployment and performance results of the Chord implementation from Section 4. We proceed with two deployments. First, the exact code presented in this paper is deployed in a ModelNet testbed with no node failure. Second, a slightly modified version of this code is run on PlanetLab. This version includes the extensions presented at the end of Section 4: use of a leaf set instead of a single successor and a single predecessor, fault-tolerant RPCs, and shorter stabilization intervals.

Chord on ModelNet. To parameterize the deployment of the Chord implementation presented in Section 4 on a testbed, we create a descriptor that describes resources requirements and limitations. The descriptor allows to further restrict memory, disk and network usage, and it specifies what information an application should receive when instantiated:

```
--[[ BEGIN SPLAY RESOURCES RESERVATION
  nb_splayd  1000
  nodes      head    1
END SPLAY RESOURCES RESERVATION ]]
```

This descriptor requests 1,000 instances of the application and specifies that each instance will receive three essential pieces of information: (1) a single-element list containing the first node in the deployment sequence (to act as rendezvous node); (2) the rank of the current process in the deployment sequence; and (3) the identity of the current process (host and port). This information is useful to bootstrap the system without having to rely on external mechanisms such as a directory service. In the case of Chord, we use this information to have hosts join the network one after the other, with a delay between consecutive joins to ensure that a single ring is created. A staggered join strategy allows better experiments reproducibility, but a massive join scenario would succeed as well. The following code is added:

```
events.sleep(job.position)                          -- 1s between joins
if #job.position > 1 then               -- first node is rendez-vous node
  join(job.nodes[1])
end
```

Finally, we register the Lua script and the deployment descriptor using one of the command line, Web service or Web-based interfaces.

Each host runs 27 to 91 Chord nodes (we show in Section 5.3 that SPLAY can handle many more instances on a single host). During the experiment, each node injects 50 random lookup requests in the system. We then undeploy the overlay, and process the results obtained from the logging facility. Figure 6(a) presents the distribution of route lengths. Figure 6(b) presents the cumulative distribution of latencies. The average number of hops is below $\frac{\log_2 N}{2}$ and the look-up time remains small. This supports our observations that SPLAY is efficient and does not introduce additional delays or overheads.

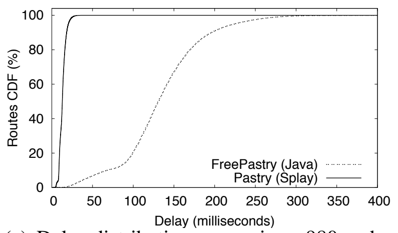
Chord on PlanetLab. Next, we deploy our Chord implementation with extensions on 380 PlanetLab nodes and compare its performance with MIT's fine-tuned C++ Chord implementation [5] in terms of delays when looking up random keys in the DHT. In both cases, we let the Chord overlay stabilizes before starting the measurements. Figure 6(c) presents the cumulative distribution of delays for 5000 random lookups (average route length is 4.1 for both systems). We observe that MIT Chord outperforms Chord for SPLAY, because it relies on a custom network layer that uses, amongst other optimizations, network coordinates for constructing latency-aware finger tables. In contrast, we did not include such optimizations in our implementation.

5.3 SPLAY Performance

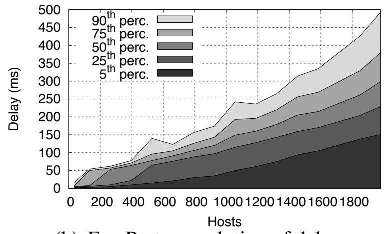
We evaluate the performance of applications using SPLAY in two ways. First, we evaluate the efficiency of the network libraries, based on the delays experienced by a sample application on a high-performance testbed. Second, we evaluate scalability: how many nodes can be run on a single host and what is the impact on performance. For these tests we chose Pastry [31] because: (i) it combines both TCP and UDP communications; (ii) it requires efficient network libraries and transport layers, each node being potentially opening sockets and sending data to a large number of other peers; (iii) it supports network proximity-based peer selection, and as such can be affected by fluctuating or unstable delays (for instance due to overload or scheduling issues).

We compare our version of Pastry with FreePastry 2.0 [4], a complete implementation of the Pastry protocol in Java. Our implementation is functionally identical to FreePastry and uses the very same protocols, e.g., locality-aware routing table construction and stabilization mechanisms to repair broken routing table entries. The only notable differences reside in the message formats (no wire compatibility) and the choice of alternate routes upon failure.

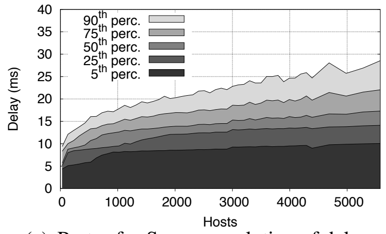
We deployed FreePastry using all optimizations ad-



(a) Delay distribution comparison, 980 nodes



(b) FreePastry, evolution of delays



(c) Pastry for SPLAY, evolution of delays

Figure 7: Comparisons of two implementations of Pastry: FreePastry and Pastry for SPLAY.

vised by the authors, that is, running multiple nodes within the same JVM, replacing Java serialization with raw serialization, and keeping a pool of opened TCP connections to peers to avoid reopening recently used connections. We used 3 JVMs on our dual cores machines, each running multiple Pastry nodes. With a large set of nodes, our experiments have shown that this configuration yields slightly better results than using a single JVM, both in terms of delay and load.

Figure 7(a) presents the cumulative delay distribution in a converged Pastry ring. The distribution of route lengths (not shown) is slightly better with FreePastry thanks to optimizations in the routing table management. Delays obtained with Pastry on SPLAY are much lower than the delays obtained with FreePastry. This experiment shows that SPLAY, while allowing for concise and readable protocol implementations, does not trade simplicity for efficiency. We also notice that Java-based programs are often too heavyweight to be used with multiple instances on a single host.[2] This is further conveyed by our second experiment that compares the evolution of delays of FreePastry (Figure 7(b)) and Pastry for SPLAY (Figure 7(c)) as the number of nodes on the testbed increases. We use a percentile-based plotting method that allows expressing the evolution of a cumulative distribution of delays with respect to the number of nodes. We can observe that: (1) delays start increasing exponentially for FreePastry when there are more than 1,600 nodes running in the cluster, that is 145 nodes per host (recall that all nodes on a single host are hosted by only 3 JVMs and share most of their memory footprint); (2) it is not possible to run more than 1,980 FreePastry nodes, as the system will start swapping, degrading performance dramatically; (3) SPLAY can handle 5,500 nodes (500 on each host) without significant drop in performance (other than the $\mathcal{O}(\log N)$ route lengths evolution, N being the number of nodes).

Figure 8 presents the load (i.e., average number of processes with “runnable” status, as reported by the Linux scheduler) and memory consumption per instance for varying number of instances. Each process is a Pastry node and issues a random request every minute. We observe that the memory footprint of an instance is lower than 1.5 MB, with just a slight increase during the experiment as nodes fill their routing table. It takes 1,263 Pastry instances before the host system starts swapping memory to disk. Load (averaged over the last minute) remains reasonably low, which explains the small delays presented by Figure 7(c).

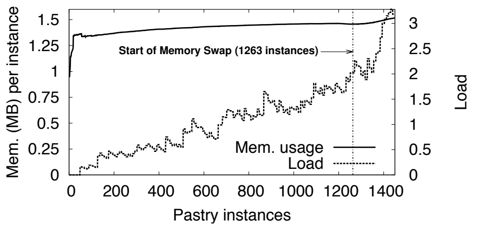


Figure 8: Memory consumption and load evolution on a single node hosting several instances of Pastry for SPLAY.

5.4 Complex Deployments

SPLAY is designed to be used within a large set of different testbeds. Despite this diversity, it is sometimes also desirable to experiment with more than a single testbed at a time. For instance, one may want to evaluate a complex system with a set of peers linked by high bandwidth, non-lossy links, emulated by ModelNet, and a set of peers facing adverse network conditions on PlanetLab. A typical usage would be to test a broker-based publish-subscribe infrastructure deployed on reliable nodes, along with a set of client nodes facing churn and lossy network links.

Such a mixed deployment requires a deep understanding of the system for setting it up using scripting and common tools, as the user has to care about NAT and firewalls traversal, port forwarding, etc. The experiment presented in this section shows that such a complex mixed deployment can be achieved using SPLAY as if it were on a single testbed. The only precondition is that the administrator of the part of the testbed that is behind a NAT or firewall defines (and opens) a range of ports that all `splayds` will use to communicate with other daemons outside the testbed. Notably for a ModelNet cluster, this operation can easily be done at the time Modelnet is installed on the nodes of the testbed and it does not requires additional access rights. All other communication details are dealt with by SPLAY itself: no modification is needed to the application code.

Figure 9 presents the delay distribution for a deployment of 1,000 nodes on PlanetLab, on ModelNet, and in a mixed deployment over both testbeds at the same time (i.e., 500 nodes on each). We notice that the delays of the mixed deployment are distributed between the delays of

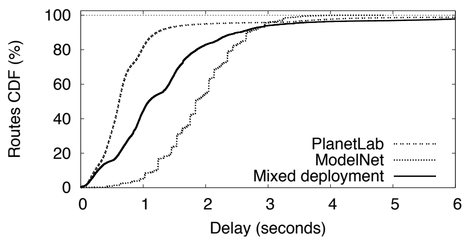


Figure 9: Pastry on PlanetLab, ModelNet, and both.

PlanetLab and the higher delays of our ModelNet cluster. The "steps" on the ModelNet cumulative delays representation are a result of routes of increasing number of hops (both in Pastry and in the emulated topology), and the fixed delays for ModelNet links.

5.5 Using Churn Management

This section evaluates the use of the churn management module, both using traces and synthetic descriptions. Using churn is as simple as launching a regular SPLAY application with a trace file as extra argument. SPLAY provides a set of tools to generate and process trace files. One can, for instance, speed-up a trace, increase the churn amplitude whilst keeping its statistical properties, or generate a trace from a synthetic description.

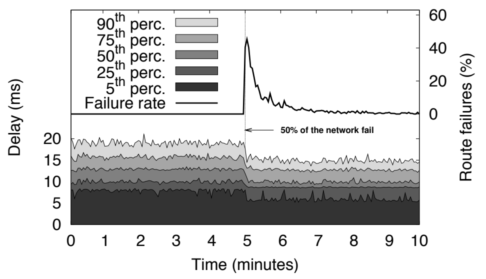


Figure 10: Using churn management to reproduce massive churn conditions for the SPLAY Pastry implementation.

Figure 10 presents a typical experiment of a massive failure using the synthetic description. We ran Pastry on our local cluster with 1,500 nodes and, after 5 minutes, triggered a sudden failure of half of the network (750 nodes). This models, for example, the disconnection of a inter-continental link or a WAN link between two corporate LANs. We observe that the number of failed lookups reaches almost 50% after the massive failure due to routing table entries referring to unreachable nodes. Pastry recovers all its routing capabilities in about 5 minutes and we can observe that delays actually decrease after the failure because the population has shrunk (delays are shown for successful routes only). While this scenario is amongst the simplest ones, churn descriptions allow users to experiment with much more complex scenarios, as discussed in Section 3.2.

Our second experiment is representative of a complex test scenario that would usually involve much engineering, testing and post-processing. We use the churn trace observed in the Overnet file sharing peer-to-peer system [12]. We want to observe the behavior of Pastry, deployed on PlanetLab, when facing churn rates that are much beyond the natural churn rates suffered in PlanetLab. As we want increasing levels of Churn, we simply "speed-up" the trace, that is, with a speed-up factor of 2x, 5x or 10, a minute in the original trace is mapped to 30, 12 or 6 seconds respectively. Figure 11 presents both the churn description and the evolution of delays and failure rates, for increasing levels of churn. The churn description shows the population of nodes and the number of joins/leaves as a function of time, and performance observations plot the evolution of the delay distribution as a function of time. We observe that (1) Pastry handles churn pretty well as we do not observe a significant failure rate when as much as 14% of the nodes are changing state within a single minute; (2) running this experiment is neither more complex nor longer than on a single cluster without churn, as we did for Figure 7(a). Based on our own experience, we estimate that it takes at least one order of magnitude less human efforts to conduct this experiment using SPLAY than with any other deployment tools. We strongly believe that the availability of tools such as SPLAY will encourage the community to further test and deploy their protocols under adverse conditions, and to compare systems using published churn models.

5.6 Deployment Performance

This section presents an evaluation of the deployment time of an application on an adversarial testbed, PlanetLab. This further conveys our position from Section 3.1 that one needs to initially select a larger set of nodes than requested to ensure that one can rely on reasonably responsive nodes for deploying the application. Traditionally, such a selection process is done by hand, or using simple heuristics based on the load or response time of the nodes. SPLAY relieves the need for the user to proceed with this selection. Figure 12 presents the deployment time for the Pastry application on PlanetLab. We vary the number of additionally probed daemons from 10% to 100% of the requested nodes. We observe that a larger set results in lower delays for deploying an application (hence, presumably, lower delays for subsequent application communications). Nonetheless, the selection of a reasonably large superset for a proper selection of peers is a tradeoff between deployment delay and redundant messages sent over the network. Based on experiments, we use by default an initial superset of 125% of requested nodes.

5.7 Resource-intensive Experiments

Our two last experimental demonstrations deal with resource-intensive applications, both for short-term and long-term runs. They further conveys SPLAY's ability to run in high performance settings and production environments, as well as demonstrating that the obtained performance is similar to the one achieved with a dedicated implementation (particularly from the network point of view). We run the following two experiments: (1) the

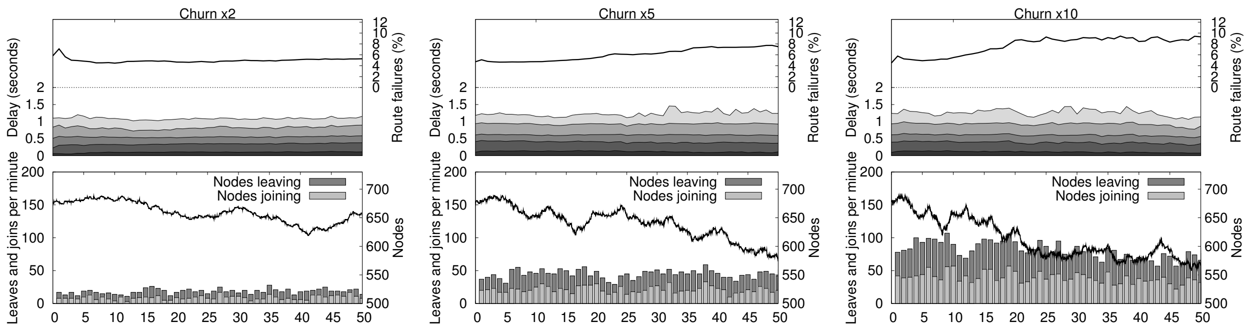


Figure 11: Study of the effect of churn on Pastry deployed on PlanetLab. Churn is derived from the trace of the Overnet file sharing system and sped up for increasing volatility.

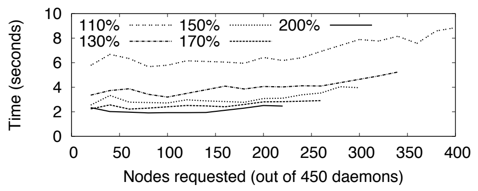


Figure 12: Deployment times of Pastry for SPLAY, as a function of (1) the number of nodes requested and (2) the size of the superset of daemons used.

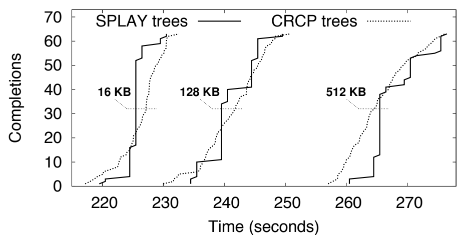


Figure 13: File distribution using trees.

evaluation of a cooperative data distribution algorithm based on parallel trees using both SPLAY and a native (C) implementation on ModelNet and (2) a distributed cooperative Web cache for HTTP accesses, which has been running for several weeks under a constant and significant load.

Dissemination using trees. This experiment compares two versions of a simple cooperative protocol [13] based on parallel n-ary trees written with SPLAY and in C. We create $n = 2$ distinct trees in the same manner as SplitStream [14] does: each of the 63 nodes is an inner member in one tree and a leaf in the other. The data to be transmitted is split into blocks, which are propagated along one of the 2 trees according to a round-robin policy. This experiment allows us to observe how SPLAY compares against a native application, CRCP, written in C [6]. Using a tree for this comparison bears the advantage of highlighting the additional delays and overheads of the platform and its network libraries (such as the sandboxing of network operations). These overheads accumulate at each level of the tree, from the root to the leaves.

Tests were run in a ModelNet testbed configured with a symmetric bandwidth of 1 Mbps for each node. Results are shown in Figure 13 for binary trees, a 24 MB file, and different block sizes (16 KB, 128 KB, 512 KB). We observe that both implementations produce similar results, which tends to demonstrate that the overhead of SPLAY's language and libraries is negligible. Differences in shape are due to CRCP nodes sending chunks sequentially to their children, while SPLAY nodes send chunks in parallel. In our settings (i.e., homogeneous bandwidth), this should not change the completion time of the last peer as links are saturated at all times.

Long-term experiment: cooperative Web cache. Our last experiment presents the performance over time of a cooperative Web cache built using SPLAY following the same base design as Squirrel [21]. This experiment highlights the ability of SPLAY to support long-run applications under constant load. The cache uses our Pastry DHT implementation deployed in a cluster, with 100 nodes that proxy requests and store remote Web resources for speeding up subsequent accesses. For this experiment, we limit the number of entries stored by each nodes to 100. Cached resources are evicted according to an LRU policy or when they are older than 120 seconds. The cooperative Web cache has been run for three weeks. Figure 14 presents the evolution of HTTP requests delay distribution for a period of 100 hours along with the cache hit ratio. We injected a continuous stream of 100 requests per second extracted from real Web access traces [7] corresponding to 1.7 million hits to 42,000 different URLs. We observe a steady cache hit ratio of 77.6%. The experienced delays distribution has remained stable throughout the whole run of the application. Most accesses (75th percentile) are cached and served in less than 25 to 100 ms, compared to non-cached accesses that require 1 to 2 seconds on average.

6 Conclusion

SPLAY is an infrastructure that aims at simplifying the development, deployment and evaluation of large-scale distributed applications. It incorporates several novel features not found in existing tools and testbeds. SPLAY applications are specified using in a high-level, efficient scripting language very close to pseudo-code commonly used by researchers in their publications. They execute

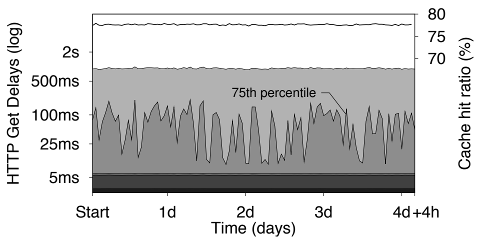


Figure 14: Cooperative Web cache: evolution of delays and cache hit ratios during a 4 days period.

in a sandboxed environment and can thus be readily deployed on non-dedicated hosts. SPLAY also includes a comprehensive set of shared libraries tailored for the development of distributed protocols. Application specifications are based on an event-driven model and are extremely concise.

SPLAY can seamlessly deploy applications in real (e.g., PlanetLab) or emulated (e.g., ModelNet) networks, as well as mixed environments. An original feature of SPLAY is its ability to inject churn in the system using a trace or a synthetic description to test applications in the most realistic conditions. Our thorough evaluation of SPLAY demonstrates that it allows developers to easily express complex systems in a concise yet readable manner, scales remarkably well thanks to its low footprint, exhibits very good performance in various deployment scenarios, and compares favorably against native applications in our experiments. SPLAY is publicly available from `http://www.splay-project.org`.

Acknowledgments: We would like to thank the anynymous reviewers for their constructive comments, as well as Petros Maniatis whose help was invaluable for preparing the final version of this paper.

References

[1] `http://www.cs.berkeley.edu/~pbg/availability/`.

[2] `http://shootout.alioth.debian.org/`.

[3] `http://www.ietf.org/rfc/rfc4627.txt`.

[4] `http://freepastry.rice.edu/`.

[5] `http://pdos.csail.mit.edu/chord/`.

[6] `http://www.crossflux.org/crcp/`.

[7] `http://ftp.ircache.net/Traces/`.

[8] ABBOTT, M., PETERSON, L. A language-based approach to protocol implementation. *IEEE/ACM Trans. Netw. 1*, 1 (Feb. 1993), 4–19.

[9] ALBRECHT, J., BRAUD, R., DAO, D., TOPILSKI, N., TUTTLE, C., SNOEREN, A. C., VAHDAT, A. Remote Control: Distributed Application Configuration, Management, and Visualization with Plush. In *LISA'07*.

[10] ANDERSON, D. Automated protocol implementation with rtag. *IEEE Trans. Soft. Eng. 14*, 3 (1988), 291–300.

[11] BAVIER, A., BOWMAN, M., CHUN, B., CULLER, D., KARLIN, S., MUIR, S., PETERSON, L., ROSCOE, T., SPALINK, T., WAWRZONIAK, M. Operating system support for planetary-scale network services. In *NSDI'04*.

[12] BHAGWAN, R., SAVAGE, S., VOELKER, G. M. Understanding availability. In *IPTPS* (Feb. 2003).

[13] BIERSACK, E., RODRIGUEZ, P., FELBER, P. Performance analysis of peer-to-peer networks for file distribution. In *QofIS'04*.

[14] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., NANDI, A., ROWSTRON, A., SINGH, A. SplitStream: High-bandwidth multicast in a cooperative environment. In *SOSP'03*.

[15] CASTRO, M., DRUSCHEL, P., KERMARREC, A.-M., ROWSTRON, A. SCRIBE: A large-scale and decentralized publish-subscribe infrastructure. *IEEE J. Sel. Areas Commun. 20*, 8 (Oct. 2002).

[16] CHANDRA, R., LEFEVER, R. M., CUKIER, M., SANDERS, W. H. Loki: A state-driven fault injector for distributed systems. In *DSN'00*.

[17] COHEN, B. Incentives to build robustness in BitTorrent. Tech. rep., `http://www.bittorrent.org/`, May 2003.

[18] DUPUY, A., SCHWARTZ, J., YEMINI, Y., BACON, D. Nest: a network simulation and prototyping testbed. *Commun. ACM 33*, 10 (1990), 63–74.

[19] ERDÖS, P., RÉNYI, A. On the evolution of random graphs. *Mat. Kuttató. Int. Közl. 5* (1960), 17–60.

[20] IERUSALIMSCHY, R., DE FIGUEIREDO, L., CELES, W. The implementation of lua 5.0. *J. of Univ. Comp. Sc. 11*, 7 (2005), 1159–1176.

[21] IYER, S., ROWSTRON, A., DRUSCHEL, P. Squirrel: a decentralized peer-to-peer web cache. In *PODC'02*.

[22] JAFFE, E., BICKSON, D., KIRKPATRICK, S. Everlab: a production platform for research in network experimentation and computation. In *LISA'07*.

[23] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., VAHDAT, A. M. Mace: language support for building distributed systems. In *PLDI'07*.

[24] KOHLER, E., KAASHOEK, M., MONTGOMERY, D. A readable TCP in the prolac protocol language. *SIGCOMM Comput. Commun. Rev. 29*, 4 (1999).

[25] LIN, S., PAN, A., GUO, R., ZHANG, Z. Simulating large-scale P2P systems with the wids toolkit. In *MASCOTS'05*.

[26] LOO, B. T., CONDIE, T., HELLERSTEIN, J., MANIATIS, P., ROSCOE, T., STOICA, I. Implementing declarative overlays. In *SOSP'05*, pp. 75–90.

[27] MICKENS, J. W., NOBLE, B. D. Exploiting availability prediction in distributed systems. In *NSDI'06*.

[28] NUSSBAUM, L., RICHARD, O. Lightweight emulation to study peer-to-peer systems. *Concurr. Comput. : Pract. Exper. 20*, 6 (2008), 735–749.

[29] RHEA, S., GEELS, D., ROSCOE, T., KUBIATOWICZ, J. Handling churn in a dht. In *2004 USENIX Annual Technical Conference*.

[30] RICCI, R., DUERIG, J., SANAGA, P., GEBHARDT, D., HIBLER, M., ATKINSON, K., ZHANG, J., KASERA, S., LEPREAU, J. The Flexlab approach to realistic evaluation of networked systems. In *NSDI'07*.

[31] ROWSTRON, A., DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware'01*.

[32] SCHROEDER, B., GIBSON, G. Large-scale study of failures in high-performance-computing systems. In *DSN'06*.

[33] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D., KAASHOEK, M., DABEK, F., BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11*, 1 (2003), 17–32.

[34] URBAN, P., DEFAGO, X., SCHIPER, A. Neko: A single environment to simulate and prototype distributed algorithms. In *ICOIN'01*.

[35] VAHDAT, A., YOCUM, K., WALSH, K., MAHADEVAN, P., KOSTIC, D., CHASE, J., BECKER, D. Scalability and accuracy in a large-scale network emulator. In *OSDI'02*.

[36] VOULGARIS, S., GAVIDIA, D., VAN STEEN, M. CYCLON: Inexpensive membership management for unstructured P2P overlays. *J. Network Syst. Manage. 13*, 2 (2005).

[37] WANG, Y., CARZANIGA, A., WOLF, A. L. Four enhancements to automated distributed system experimentation methods. In *ICSE'08*.

[38] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. In *OSDI'02*.

Notes

[1]Note that, without the requirement for binary compatibility, the size of our implementation could be significantly reduced. Our BitTorrent implementation has been successfully used for downloading several times the Ubuntu Linux disk in official swarms.

[2]This possibility is notably useful to test characteristics that do not depend much on the performance of individual nodes with a limited-size testbed, e.g., to evaluate the scalability of routing in an overlay.

Modeling and Emulation of Internet Paths

Pramod Sanaga Jonathon Duerig Robert Ricci Jay Lepreau

University of Utah, School of Computing
{*pramod, duerig, ricci, lepreau*}@*cs.utah.edu*

Abstract

Network emulation subjects real applications and protocols to controlled network conditions. Most existing network emulators are fundamentally *link* emulators, not *path* emulators: they concentrate on faithful emulation of the transmission and queuing behavior of individual network hops in isolation, rather than a path as a whole. This presents an obstacle to constructing emulations of observed Internet paths, for which detailed parameters are difficult or impossible to obtain on a hop-by-hop basis. For many experiments, however, the experimenter's primary concern is the end-to-end behavior of paths, not the details of queues in the interior of the network.

End-to-end measurements of many networks, including the Internet, are readily available and potentially provide a good data source from which to construct realistic emulations. Directly using such measurements to drive a link emulator, however, exposes a fundamental disconnect: link emulators model the *capacity* of resources such as link bandwidth and router queues, but when reproducing Internet paths, we generally wish to emulate the measured *availability* of these resources.

In this paper, we identify a set of four principles for emulating entire paths. We use these principles to design and implement a path emulator. All parameters to our model can be measured or derived from end-to-end observations of the Internet. We demonstrate our emulator's ability to accurately recreate conditions observed on Internet paths.

1 Introduction

In network emulation, a real application or protocol, running on real devices, is subjected to artificially induced network conditions. This gives experimenters the opportunity to develop, debug, and evaluate networked systems in an environment that is more representative of the Internet than a LAN, yet more controlled and predictable than running live across deployed networks such as the Internet. Due to these properties, network emulation has become a popular tool in the networking and distributed systems communities.

Network emulators work by forwarding packets from an application under test through a set of queues that approximate the behavior of router queues. By adjusting the parameters of these queues, an experimenter can control the emulated capacity of a link, delay packets, and introduce packet loss. Popular network emulators include Dummynet [22], ModelNet [27], NIST Net [7], and Emulab (which uses Dummynet) [32]. These emulators focus on *link emulation*, meaning that they concentrate on faithful emulation of individual links and queues.

In many cases, particularly in distributed systems, the system under test runs on hosts at the edges of the network. Experiments on these systems are concerned with the end-to-end characteristics of the *paths* between hosts, not with the behavior of individual queues in the network. For such experiments, detailed modeling of individual queues is not a necessity, so long as end-to-end properties are preserved. One way to create emulations with realistic conditions is to use parameters from real networks, such as the Internet, but it can be difficult or impossible to obtain the necessary level of detail to recreate real networks on a hop-by-hop basis. Thus, in order to run experiments using conditions from real networks, there is a clear need for a new type of emulator that models paths as a whole rather than individual queues.

In this paper, we identify a set of principles for path emulation and present the design and implementation of a new path emulator. This emulator uses an abstract and straightforward model of path behavior. Rather than requiring parameters for each hop in the path, it uses a much smaller set of parameters to describe the entire path. The parameters for our model can be estimated or derived from end-to-end measurements of Internet paths. In addition to the simplicity and efficiency benefits, this end-to-end focus makes our emulator suitable for recreating observed Internet paths inside a network testbed, such as Emulab, where experiments are predictable, repeatable, and controlled.

1.1 Path Emulation Approaches

One approach to emulating paths is to use multiple instances of a link emulator, creating a series of queues for the traffic under test to pass through, much like the series of routers it would pass through on a real path. ModelNet and Emulab in particular are designed for use in this fashion. Building a path emulator in this way, however, requires a router-level topology. While such topologies can be generated from models or obtained for particular networks, obtaining detailed topologies for arbitrary Internet paths is very difficult. Worse, to construct an accurate emulation, capacity, queue size, and background traffic for each link in the path must be known, making reconstruction of Internet paths intractable.

Another alternative is to approximate a path as a single link, using the desired end-to-end characteristics such as available bandwidth, and observed round-trip time, to set the parameters of a single link emulator. Because these properties can be measured from the edges of the network, this is an attractive approach. A recent survey of the distributed systems literature [29] shows that many distributed or network systems papers published in top venues [4, 5, 8, 18, 19, 23, 26, 30]—nearly one third of those surveyed—include a topology in which a single hop is used to approximate a path.

On the surface, this seems like a reasonable approximation: distributed systems tend to be sensitive to high-level network characteristics such as bandwidth, latency, and packet loss rather than the fine-grained queuing behavior of every router along a path. However, as we discuss in Section 2 and demonstrate in Section 4, using a single link emulator to model a measured path can often fail even simple tests of accuracy. This is due to a fundamental mismatch between the fact that link emulators model the *capacity* of links, and the fact that end-to-end measurements reveal the *availability* of resources on those links. This difference can result in flows being unable to achieve the bandwidth set by the experimenter or seeing unrealistic round-trip times, and these errors can be quite large. This model also does not capture interactions *between* paths, such as shared bottlenecks, or *within* paths, such as the reactivity of background flows.

1.2 Path Emulation Principles

What is needed is a new approach to emulation that models entire Internet paths rather than individual links within those paths. We have identified four principles for designing such an emulator:

- **Model capacity and available bandwidth separately.** Existing link emulators model links with limited *capacity*. We show why this is not always sufficient to create a path emulation with a particular target *available bandwidth*. We provide the mathematical basis for deciding how much capacity and how much cross-traffic are necessary to produce the desired effect.

- **Pick appropriate queue sizes.** Much work has been done in choosing "good" values for queue sizes in real routers, but the issues that apply to emulation are somewhat different. We define concrete upper and lower bounds for queue sizes in emulation and simulation. These bounds are derived from the delay and available bandwidth parameters of the emulated paths to ensure that the configured bandwidth is actually achievable.

- **Use an abstracted model of the reactivity of background flows.** Real networks have cross-traffic that reacts in complex ways to foreground traffic. Available bandwidth can change in reaction to foreground flows, and thus is a function of the load offered by the system under test. Discovering the characteristics of background traffic from the edge of the network is very difficult—even the degree of statistical multiplexing is obscured by TCP unfairness in the presence of disparate RTTs [15]. We show that we can model reactivity by concentrating only on the effect that the reactivity of the background flows has on foreground flows.

- **Model shared bottlenecks.** When modeling a set of paths, it is likely that some of those paths share bottlenecks, and that this will affect the properties seen by foreground flows. Such bottleneck sharing occurs naturally in router-level emulation, but must be explicitly modeled in an abstracted emulation.

Note that any of these principles can, individually, be applied to a link emulator; indeed, our path emulator implementation, presented in Section 3, is based on the Dummynet link emulator. Our contribution lies in identifying all four principles as being fundamental to path emulation, and in implementing a path emulator based on them so that they can be empirically evaluated. Although our focus in this paper is on emulation, these principles are also applicable to simulation.

2 Path Modeling

Our path model grows out of these four principles. It takes as input a set of five parameters: base round-trip time (RTT), available bandwidth (ABW), capacity, shared bottlenecks, and functions describing the reactivity of background traffic. As shown in Section 3.3, it is possible to measure each of these parameters from end hosts on the Internet, making it feasible to build reconstructions of real paths. We discuss the ways in which these parameters are interrelated, and contrast our model with the approach of using end-to-end measurements as input to a single link emulator, showing the deficiencies of such an approach and how our model corrects them.

Our model focuses on accommodating foreground TCP flows, leaving emulation for other types of foreground flows as future work. We also concentrate on emulating stationary conditions for paths; in principle, any or all parameters to our model can be made time-varying to capture more dynamic network behavior.

2.1 Base RTT

The round-trip time (RTT) of a path is the time it takes for a packet to be transferred in one direction plus the time for an acknowledgment to be transferred in the opposite direction. We model the RTT of a path by breaking it into two components: the "base RTT" [6] (RTT_{base}) and the queuing delay of the bottleneck link.

The base RTT includes the propagation, transmission, and processing delay for the entire path and the queuing delay of all non-bottleneck links. When the queue on the bottleneck link is empty, the RTT of the path is simply the base RTT. In practice, the minimum RTT seen on a path is a good approximation of its base RTT. Because transmission and propagation delays are constant, and processing delays for an individual flow tend to be stable, a period of low RTT indicates a period of little or no queuing delay.

The base RTT represents the portion of delay that is relatively insensitive to network load offered by the foreground flows. This means that we do not need to emulate these network delays on a detailed hop-by-hop basis: a fixed delay for each path is sufficient.

2.2 Capacity, Available Bandwidth, and Queuing

The bottleneck link controls the bandwidth available on the path, contributes queuing delay to the RTT, and causes packet loss when its queue fills. Thus, three properties of this link are closely intertwined: link capacity, available bandwidth, and queue size.

We make the common assumption that there is only one bottleneck link on a path in a given direction [9] at a given time, though we do not assume that the same link is the bottleneck in both directions.

2.2.1 Capacity and Available Bandwidth

Existing link emulators fundamentally emulate limited *capacity* on links. The link speed given to the emulator is used to determine the rate at which packets drain from the emulator's bandwidth queue, in the same way that a router's queue empties at a rate governed by the capacity of the outgoing link. The quantity that more directly affects distributed applications, however, is *available bandwidth*, which we consider to be the maximum rate sustainable by a foreground TCP flow. This is the rate at which the foreground flow's packets empty from the bottleneck queue. Assuming the existence of competing traffic, this rate is lower than the link's capacity.

It is not enough to emulate available bandwidth using a capacity mechanism. Suppose that we set the capacity of a link emulator using the available bandwidth measured on some Internet path: inside of the emulator, packets will drain more slowly than they do in the real world. This difference in rate can result in vastly different queuing delays, which is not only disastrous for latency-sensitive experiments, but as we will show, can cause inaccurate bandwidth in the emulator as well.

Let q_f and q_r be the sizes of the bottleneck queues in the forward and reverse directions, respectively, and let C_f and C_r be the capacities. The maximum time a packet may spend in a queue is $\frac{q}{C}$, giving us a maximum RTT that can be observed on the path:

$$RTT_{max} = RTT_{base} + \frac{q_f}{C_f} + \frac{q_r}{C_r} \qquad (1)$$

If we were to use ABW_f and ABW_r—the available bandwidth measured from some real Internet path—to set C_f and C_r, Equation 1 would yield much larger queuing delays within the emulator than seen on the real path (assuming the queues sizes on the path and in the emulator are the same).

For instance, consider a real path with $RTT_{base} =$ 50 ms, a bottleneck of symmetric capacity $C_f = C_r =$ 43 Mbps (a T-3 link) and available bandwidth $ABW_f = ABW_r = 4.3$ Mbps. For a small q_f and q_r of 64 KB (fillable by a single TCP flow), the RTT on the path is bounded at 74 ms, since the forward and reverse directions each contribute at most 12 ms of queuing delay. However, if we set $C_f = C_r = 4.3$ Mbps within an emulator (keeping queue sizes the same), each direction of the path can contribute up 120 ms of queuing delay. The total resulting RTT could reach as high as 290 ms.

This unrealistically high RTT can lead to two problems. First, it fails to accurately emulate the RTT of the real path, causing problems for latency-sensitive applications. Second, it can also affect the bandwidth available to TCP, a problem we discuss in more detail in Section 2.2.2.

One approach reducing the maximum queuing delay would be to simply reduce the q_f and q_r inside of the emulator. This may result in queues that are simply too small. In the example above, to reduce the queuing delay within the path emulator to the same level as the Internet path, we would we would have to reduce the queue size by a factor of 10 to 6.4 KB. A queue this small will cause packet loss if a stream sends a small burst of traffic, preventing TCP from achieving the requested available bandwidth. We also discuss minimum queue size in more detail in Section 2.2.2.

The solution to these queuing problems is to separate the notions of *capacity* and *available bandwidth* in our path emulation model: they are independent parameters to each path. When we wish to emulate a path with competing traffic at the bottleneck, we set $C > ABW$. To model links with no background traffic, we can still set $C = ABW$, as is done implicitly in a link emulator.

Of course, when $C > ABW$, we must fill the excess capacity to limit foreground flows to the desired ABW. A common solution to this problem has been to add a number of background TCP flows to the bottleneck. The problem with this technique is one of measurement. When the emulation is constructed using end-to-end observations of a real path, discovering the precise behavior or even the number of competing background flows is not possible from the edges of the network. Adding reactive background flows to our emulation would not mirror the reactivity on the real network, and would result in an inexact ABW in the emulator.

Since there is not enough information to replicate the background traffic at the bottleneck, we separately emulate its *rate* and its *reactivity*. We can precisely emulate a particular level of background traffic using non-responsive, constant-bit-rate traffic. This mechanism allows us to provide an independent mechanism for emulating reactivity, described in Section 2.3. The reactivity model can change the level of background traffic to emulate responsiveness while providing a precise available bandwidth to the application at every point in time.

2.2.2 Queue Size

Much work has been done in choosing appropriate values for queue sizes in real routers [1], but the set of constraints for emulation are somewhat different: we have a relatively small set of foreground flows and a specific target ABW that we wish to achieve. Although queue sizes can be provided directly as parameters to our model, we typically calculate them from other parameters. We do this for two reasons. First, it is difficult to measure the bottleneck queue size from the endpoints of the network due to interference from cross-traffic. Second, the bottleneck queue size affects applications only through additional latency or reduced bandwidth it might cause. Because our primary concern is emulating application-visible effects, we include a method for selecting a queue size that enables accurate emulation of those effects.

We look at queue sizes in two ways: in terms of *space* (their capacity in bytes or packets) and in terms of *time* (the maximum queuing delay they may induce). This leads to two constraints on queue size:

- The queue must be large enough in space that a TCP stream is able to get the full desired ABW; it should not drop bursts of packets.
- The queue must not be so large in time that the queuing delay from a full queue causes excessive RTTs, as seen in Equation 1.

Lower bound. Finding the lower bound is straightforward. Current best practices suggest that for a small number of flows, a good lower bound on queue size is the sum of the bandwidth-delay products of all flows traversing that link [1]. Here, a “small number” of flows is fewer than about 500. Because we are concerned only with flows of a foreground application, the number of flows *on a specific path* will typically be much smaller than this. For a TCP flow f, the window size w_f is roughly equal to its bandwidth-delay product, and is capped by w_{max}, the maximum window size allowed by the TCP implementation. Thus, for a given path in a given direction, we sum over the set of flows F, giving us a lower bound on q:

$$q \geq \sum_{f \in F} min(w_f, w_{max}) \tag{2}$$

This bound applies to the queues in both directions on the path, q_f and q_r. Intuitively, the queue must be large enough to hold at least one window's worth of packets for each flow traversing the queue.

Upper bound. The upper bound is more complex. The maximum RTT tolerable for a given flow on a given path, before it becomes window-limited, is given by (using the empirically derived TCP performance model demonstrated by Padhye et al. [16]):

$$RTT_{max} = \frac{w_{max}}{ABW} \tag{3}$$

where ABW is the available bandwidth we wish the flow to experience. If the RTT grows above this limit, the bandwidth-delay product exceeds the maximum window size w_{max}, and the flow's bandwidth will be limited by TCP itself, rather than the ABW we have set in the emulator. Since our goal is to accurately emulate the given ABW, this would result in an incorrect emulation.

It is important to note that a single flow along a path cannot cause itself to become window-limited, as it will either fill up the bottleneck queue before it reaches w_{max}, or stabilize on an average queue occupancy no larger than w_{max}. Two or more flows, however, can induce this behavior in each other by filling a queue to a greater depth than can be sustained by either one. Even flows crossing a bottleneck in opposite directions can cause excessive RTTs, as each flow's ACK packets must wait in a queue with the other flow's data packets. The value of w_{max} may be defined by several factors, including limitations of the TCP header and configuration options in the TCP stack, but is essentially known and fixed for a given experiment.

Flows may travel in both directions along a path, and while both will see the same RTT, they may have different RTT_{max} values if the ABW on the path is not symmetric. Without loss of generality, we define the "forward" direction of the path to be the one with the higher ABW. From Equation 3, flows in this direction have the smaller RTT_{max}, and since we do not want either flow to become window-limited, we use ABW_f to find the upper bound.

Because most (Reno-derived) TCP stacks tend to reach a steady state in which the bottleneck queue is full [16], bottleneck queues tend to be nearly full, on average. Thus, we can expect flows to experience RTTs near the maximum given by Equation 1 in steady-state operation. For our emulation of ABW to be accurate, then, Equation 1 (the maximum observable RTT) must be less than or equal to Equation 3 (the maximum tolerable RTT). If we set the two capacities to be equal and solve for the queue sizes, this gives us:

$$q_f + q_r \leq C \cdot \left(\frac{w_{max}}{ABW_f} - RTT_{base} \right) \tag{4}$$

Because all terms on the right side are either fixed or parameters of the path, we have a bound on the total queue size for the path. (It is not necessary for the forward and reverse capacities to be equal to solve the equation. We do so here for simplicity and clarity.)

Setting the Queue Size. To select sizes for the queues on a path, we must simply split the total upper bound in Equation 4 between the two directions, in such a way that neither violates Equation 2.

These two bounds have a very important property: it is not necessarily possible to satisfy both when $C = ABW$. When either bound is not met, the emulation will not provide the desired network characteristics. The capacity C acts as a scaling factor on the upper bound. By adjusting it while holding ABW constant, we can raise or lower the maximum allowable queue size, making it possible to satisfy both equations.

Figure 1 illustrates this principle by showing valid queue sizes as a function of capacity. As capacity changes, the upper bound increases while the lower bound remains constant. When capacity is at or near available bandwidth, the upper bound is *below* the lower bound, which means that no viable queue size can be selected. As capacity increases, these lines intersect and yield an expanding region of queue sizes that fulfill both constraints. This underscores the importance of emulating available bandwidth and capacity separately.

Asymmetry. Throughput artifacts due to violations of Equation 3 are exacerbated when traffic on the path is bidirectional and the available bandwidth is asymmetric. In this case, the flows in each direction can tolerate different maximum RTTs, with the flow in the forward (higher ABW) direction having the smaller upper bound.

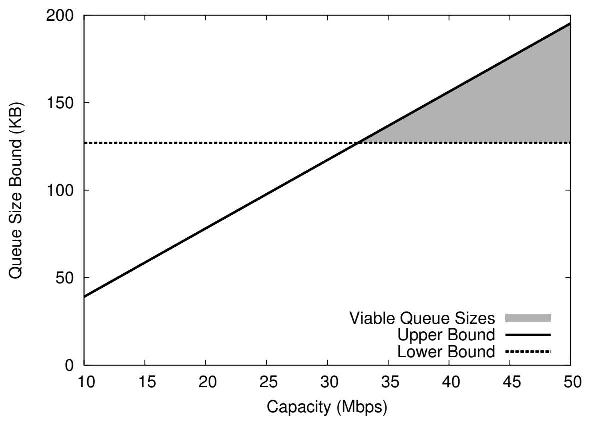


Figure 1: The relationship between capacity and the bounds on queue size for a path with $ABW_f = ABW_r = 10$ Mbps, $RTT_{base} = 20$ ms, and $w_{max} = 65$ KB. Low capacities prevent any viable queue size.

This means that it is disproportionately affected by high RTTs. Others have described this phenomenon [2], and we demonstrate it empirically in Section 4.1.

To determine how common paths with asymmetric ABW are in practice, we measured the available bandwidth on 7,939 paths between PlanetLab [17] nodes. Of those paths, 30% had greater ABW in one direction than the other by a ratio of at least 2:1, and 8% had a ratio of at least 10:1. Because links with asymmetric capacities (e.g., DSL and cable modems) are most common as last-mile links, and because PlanetLab has few nodes at such sites, it is highly likely that most of this asymmetry is a result of bottlenecks carrying asymmetric traffic. Our experiments in Section 4 shows that on a path with an available bandwidth asymmetry ratio as small as 1.5:1, a simple link emulation model that does not separate capacity and ABW, and does not set queue sizes carefully, can result in a 30% error in achieved throughput.

2.2.3 Putting It Together

Figure 2 shows an overview of our model as described thus far. We model the bottleneck of a path with a queue that drains at a fixed rate, and a constant bit-rate cross-traffic source. The rate at which the queue drains is the capacity, and the difference between the injection rate of the cross-traffic and the capacity is the available bandwidth. The remainder of the delay on the path is modeled by delaying packets for a constant amount of time governed by RTT_{base}. The two halves of the path are modeled independently to allow for asymmetric paths.

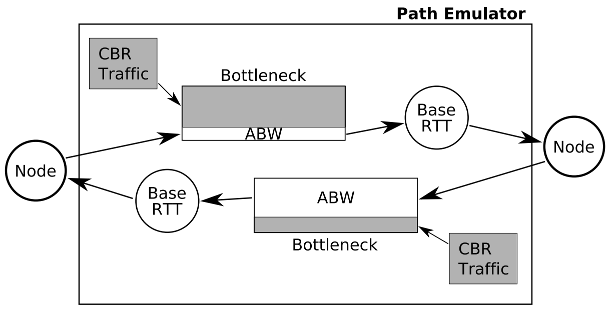


Figure 2: Modeling a single path, in both the forward and reverse directions.

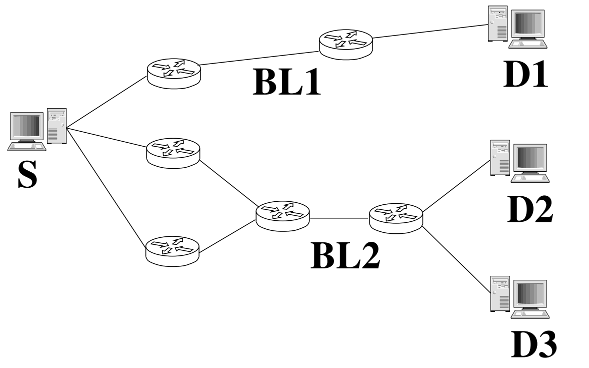


Figure 3: A router-level topology, showing two bottleneck links. One (*BL2*) is shared by two paths from source *S*: the paths to destinations *D2* and *D3*.

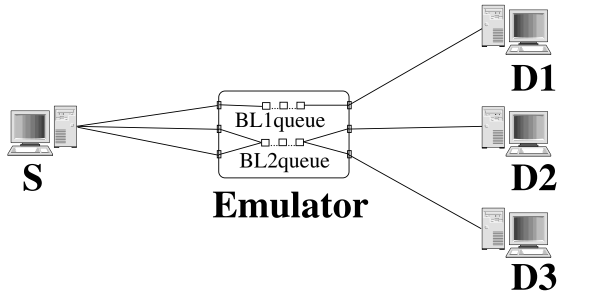


Figure 4: An abstracted view of Figure 3, with the bottleneck links represented as bottleneck queues.

2.3 Interactions Between Flows

In addition to emulating the behavior of the foreground flows' packets in the bottleneck queue, we must also emulate two important interactions: the interaction of multiple foreground flows on different paths that share bottlenecks, and the interaction of foreground flows with responsive background traffic.

Shared Bottlenecks. To properly emulate sets of paths, we must take into account bottlenecks that are shared by more than one path. Consider the simple case in Figure 3. If we do not model the bottleneck *BL2* (shared by the paths from source *S* to destinations *D2* and *D3*), we will allow multiple paths to independently use bandwidth that should be shared between them. Doing so could result in the application getting significantly more bandwidth within the emulator than it would on the real paths [21].

We do not, however, need to know the full router-level topology of a set of paths in order to know that they share bottlenecks. Existing techniques [12] can detect the existence of such bottlenecks from the edges of the network, by correlating the observed timings of simultaneous packet transmissions on the paths.

To model paths that share a bottleneck, we abstract shared bottlenecks in a simple manner: instead of giving each path an independent bandwidth queue, we allow multiple paths to share the same queue. Traffic leaving a node is placed into the appropriate queue based on which destinations, if any, share bottlenecks from that source.

This is illustrated in Figure 4: the two bottleneck links in the original topology are represented as bottleneck queues inside the path emulator. While paths sharing a bottleneck link share a bottleneck queue, each still has its own base RTT applied separately. Because base RTT represents links in the path other than the bottleneck link, links with a shared bottleneck do not necessarily have the same RTT. With this model, it is also possible for a path to pass through a different shared bottleneck in each direction.

Reactivity of Background Traffic. Flows traversing real Internet paths interact with cross-traffic, and this cross-traffic typically has some reactivity to the foreground flows. Thus, ABW on a path is not constant, even under the assumption that the set of background flows does not change. Simply setting a static ABW for a path can miss important effects: if more than one flow is sent along the path, the aggregate ABW available to all foreground flows may be greater, as the background traffic backs off further in reaction to the increased load. This is particularly important when the bottleneck is shared between two or more paths; the load on the bottleneck is the sum of the load on all paths that pass through it.

While it is possible to create reactivity in the emulation by sending real, reactive cross-traffic (such as competing TCP flows) across the bottlenecks, doing so in a way that faithfully reproduces conditions on an observed link is problematic. The number, size, and RTT of these background flows all affect their reactivity, and such detail is not easily observed from endpoints. We turn to our guiding principle of abstraction, and model the *reactivity* of the background traffic to our foreground flows, rather than the details of the background traffic itself.

We look at *ABW* as a function of offered load: $ABW_d(L_d)$ gives the aggregate bandwidth available in direction d (forward or reverse) of a given path, as a

Name	Type	Description
C_f, C_r	fixed	Capacity of the bottleneck in the forward and reverse directions. Fixed to value sufficient to make satisfaction of queue bounds possible for most experiments.
$ABW_f(\|F_f\|)$, $ABW_r(\|F_r\|)$	measured	Table giving available bandwidth for the forward and reverse directions, as a function of the number of flows traversing the path in that direction.
RTT_{base}	measured	Base RTT of the path, split evenly between the two directions.
$S_{p \in P}$	measured	A subset of paths p from the set of all paths P that share a common bottleneck. Multiple instances of this parameter may be given.
q_f, q_r	derived	The queue size for each bottleneck is derived from the measured values of ABW, RTT_{base}, and the fixed capacity. If ABW is adjusted based on the reactivity table, queue size is as well.

Table 1: The parameters to our path emulation. All parameters except $S_{p \in P}$ are given on a per-path basis.

function of the offered load L_d in that direction on that path. A set of such functions, one for each direction on each path, is supplied as a parameter to the emulation. Note that this offered load—and with it the available bandwidth—will likely vary over time during the emulated experiment. The *ABW* function can be created analytically based on a model or it can be measured directly from a real path, by offering loads at different levels and observing the resulting throughput. The emulation can then provide—with high accuracy—exactly the desired ABW. Once we have used the reactivity functions to determine the aggregate bandwidth available on a path, we can set both the capacity and queue sizes as described in Section 2.2.2.

Because an *ABW* function is a parameter of a particular path, when multiple paths share a bottleneck, we must combine their functions. There are multiple ways that the *ABW* functions may be combined. Ideally, we would like to account for every possible combination of flows using every possible set of paths that share the bottleneck. The combinatorial explosion this creates, however, quickly makes this infeasible for even a modest number of paths. Instead, the simple strategy that we currently employ is to take the mean of the *ABW* values for each individual path sharing the bottleneck, weighted by the number of flows on each path. We are exploring the possibility that more complicated approaches may yield more realistic results.

3 Implementing a Path Emulator

Although the model we have discussed is applicable to both simulation and emulation, we chose to do our initial implementation in an emulator. Our prototype path emulator is implemented as a set of enhancements to the Dummynet [22] link emulator. We constructed our prototype within the Emulab network testbed [32], but it is not fundamentally linked to that platform.

3.1 Basis: The Dummynet Link Emulator

Dummynet is a popular link emulator implemented in the FreeBSD kernel. It intercepts packets coming through an incoming network interface and places them in its internal objects—called *pipes*—to emulate the effects of delay, limited bandwidth, and probabilistic random loss. Each pipe has one or more queues associated with it. Given the capacity or the delay of a pipe, Dummynet schedules packets to be emptied from the corresponding queues and places them on the outgoing interface.

Dummynet can be configured to send a packet through multiple pipes on its path from an incoming interface to an outgoing interface. One pipe may enforce the base delay of the link, and a subsequent pipe may model the capacity of the link being emulated. Dummynet uses the IPFW packet filter to direct packets into pipes, and can therefore use many different criteria to map packets to pipes.

In network emulation testbeds, "shaping nodes" are interposed on emulated links, each acting as a transparent bridge between the endpoints. In Emulab, the shaping nodes' Dummynet is configured with one or more pipes to handle traffic in each direction on the emulated link, allowing for asymmetric link characteristics. Shaping nodes can also be used in LAN topologies by placing a shaping node between each node and switch implementing the LAN. Thus traffic between any two nodes passes through two shaping nodes: one between the source and the LAN, and one between the LAN and the destination.

3.2 Enhancements for Path Emulation

To turn Dummynet into a path emulator, we made a number of enhancements to it. The parameters to the resulting path emulator are summarized in Table 1.

Capacity and Available Bandwidth. Dummynet implements bandwidth shaping in terms of a bandwidth pipe, which contains a bandwidth queue that is drained at a specified rate, modeling some capacity C. To separate the emulation of capacity from available bandwidth, we modified Dummynet to insert "placeholder" packets into the bandwidth queues at regular, configurable intervals. These placeholder packets are neither received from nor sent to an actual network interface; their purpose is simply to adjust the rate at which foreground flows' packets move through the queue. The placeholders are sent at a constant bit rate of $C - ABW$, setting the bandwidth available to the experimenter's foreground flows. ABW can be set as a function of offered load, using the mechanism described below.

Base Delay. We leave Dummynet's mechanism for emulating the constant base delay unchanged. Packets pass through "delay" queues, where they remain for a fixed amount of time.

Queue Size. We use Equation 4 to set the queue size for the bandwidth queues in each direction of each path, dividing the number of bytes equally between the forward and reverse directions. Because the model assumes that packets are dropped almost exclusively by the bottleneck router, modeled by the bandwidth queues, the size of the delay queues is effectively infinite.

Background Traffic Reactivity. We implement the ABW_f and ABW_r functions as a set of tables that are parameters to the emulator. Each path is associated with a distinct table in each direction. We measure the offered load on a path by counting the number of foreground flows traversing that path. We do this for two reasons. First, it makes the measurement problem more tractable, allowing us to measure a relatively small, discrete set of possible offered loads on the real path. Second, our goal is to recreate inside the emulator the behavior that one would see by sending the same flows on the real network. The complex feedback system created by the interaction of foreground flows with background flows is captured most simply by measuring entire flows, as it is strongly related to TCP dynamics. It does have a downside, however, in that it makes the assumption that the foreground flows will be full-speed TCP flows. During an execution of the emulator, a traffic monitor counts the number of active foreground flows on each path, and informs the emulator which table entry to use to set the aggregate ABW for the path. This target ABW is achieved inside the emulator by adjusting the rate at which placeholder packets enter the bandwidth queue. Our implementation also readjusts bottleneck queue sizes in reaction to these changes in available bandwidth.

Shared Bottlenecks. We implement shared bottlenecks by allowing a bandwidth pipe to shape traffic to more than one destination simultaneously. For each endpoint host in the topology, the emulator takes as a parameter a set of "equivalence classes": sets of destination hosts that share a common bottleneck, and thus a common bandwidth pipe. Packets are directed into the proper bandwidth pipe using IPFW rules. Our current implementation only supports bottlenecks that share a common source. We are in the process of extending our prototype to implement other kinds of bottlenecks, such as those that share a destination.

3.3 Gathering Data from the Real World

To create and run experiments with the path emulator, we need a source of input data for the parameters shown in Table 1. Although it is possible to synthesize values for these parameters, we concentrate here on gathering them from end-to-end measurements of the Internet.

We developed a system for gathering data for these parameters using hosts in PlanetLab [17], which gives us a large number of end-site vantage points around the world. Each node in the emulation is paired with a PlanetLab node; measurements taken from the PlanetLab node are used to configure the paths to and from the emulated node.

To gather values for RTT_{base}, we use simple `ping` packets, sent frequently over long periods of time [10]. The smallest RTT seen for a path is presumed to be an event in which the probe packet encountered no significant queuing delay, and thus representative of the base RTT.

To detect shared bottlenecks from a source to a set of destinations, we make use of a wavelet-based congestion detection tool [12]. This tool sends UDP probes from a source node to all destination nodes of interest and records the variations in one-way delays experienced by the probe packets. Random noise introduced in the delays by non-bottleneck links is removed using a wavelet-based noise-removal technique. The paths are then grouped into different clusters, with all the paths from the source to the set of destinations going through the same shared bottleneck appearing in a single cluster. The shared bottlenecks found by this procedure are passed to the emulator as the $S_{p \in P}$ sets.

Our goal is that a TCP flow through the emulator should achieve the same throughput as a TCP flow sent along the real path. So, we use a definition of ABW that differs slightly from the standard one—we equate the available bandwidth on a path to the throughput achieved by a TCP flow. We also need to measure how this ABW changes in response to differing levels of foreground traffic. While we cannot observe the background traffic on the bottleneck directly, we *can* observe how different levels of foreground traffic result in different amounts of bandwidth available to that foreground traffic. Although packet-pair and packet-train [9, 13, 20] measurement tools are efficient, they do not elicit reactions from background traffic. For this reason, we use the following methodology to concurrently estimate the ABW and reactivity of background traffic on a particular path.

To measure the reactivity of Internet cross-traffic to the foreground flows, we run a series of tests using `iperf` between each pair of PlanetLab nodes, with the number of concurrent flows ranging from one to ten. We use the values obtained from these tests between all paths of interest to build the reactivity tables for the path emulator. However, running such a test takes time: only one test can be active on each path at a time, and `iperf` must run long enough to reach a steady state. Thus, our measurements necessarily represent a large number of snapshots taken at different times, rather than a consistent snapshot taken at a single time. The cross-traffic on the bottleneck may vary significantly during this time frame. So, the reactivity numbers are an approximation of the behavior of cross-traffic at the bottleneck link. This is a general problem with measurements that must perturb the environment to differing levels. The time required to gather these measurements is also the main factor limiting the scale of our emulations.

Another problem that arises is the proper ABW value for shared bottlenecks. Because paths that share a bottleneck do not necessarily have the same RTTs, they may evoke different levels of response from reactive background traffic. It is not feasible to measure every possible combination of flows on different paths through the same shared bottleneck. Thus, we use the approximation discussed in Section 2.3 to set ABW for shared bottlenecks.

Our current implementation does not measure the bottleneck link capacities C_f and C_r on PlanetLab paths, due to the difficulty of obtaining accurate packet timings on heavily loaded PlanetLab nodes [25]. We set the capacity of all bottleneck links to 100 Mbps. In practice, we find that for $C \gg ABW$, the exact value of C makes little difference on the emulation, and thus we typically set it to a fixed value. We demonstrate this in Section 4.3.

4 Evaluation

The goal of our evaluation is to show that our path emulator accurately reproduces measurements taken from Internet paths. We demonstrate, using micro-benchmarks and a real application, that our path emulator meets this goal under conditions in which approximating the path using a single link emulator fails to do so. In the experiments described below, we concentrate on accurately reproducing TCP throughput and observed RTT.

All of our experiments were run in Emulab on PCs with 3 GHz Pentium IV processors and 2 GB of RAM. The nodes running application traffic used the Fedora Core Linux distribution with a 2.6.12 kernel, with its default BIC-TCP implementation. The link emulator was Dummynet running in the FreeBSD 5.4 kernel, and our path emulator is a set of modifications to it. All measurements of Internet paths were taken using PlanetLab hosts.

4.1 Effect on TCP Throughput

We begin by running a micro-benchmark, `iperf`, a bulk-transfer tool that simply tries to achieve as much throughput as possible using a single TCP flow.

We performed a series of experiments to compare the behavior of `iperf` when run on real Internet paths, an unmodified Dummynet link emulator, and our path emulator. We used a range of ABW and RTT values, some taken from measurements on PlanetLab and some synthetic. The ABW values from PlanetLab were measured using `iperf`, and thus the emulators' accuracy can be judged by how closely `iperf`'s performance in the emulated environment matches the ABW parameter. In the link emulator, we set the capacity to the desired ABW (as there is only one bandwidth parameter), and in the path emulator, we set capacity to 100 Mbps. The link emulator uses Dummynet's default queue size of 73 KB, and the path emulator's queue size was set using Equation 4. Reactivity tables and shared bottlenecks were not used for these experiments. We started two TCP flows simultaneously on the emulated path, one in each direction, and report the mean of five 60-second runs.

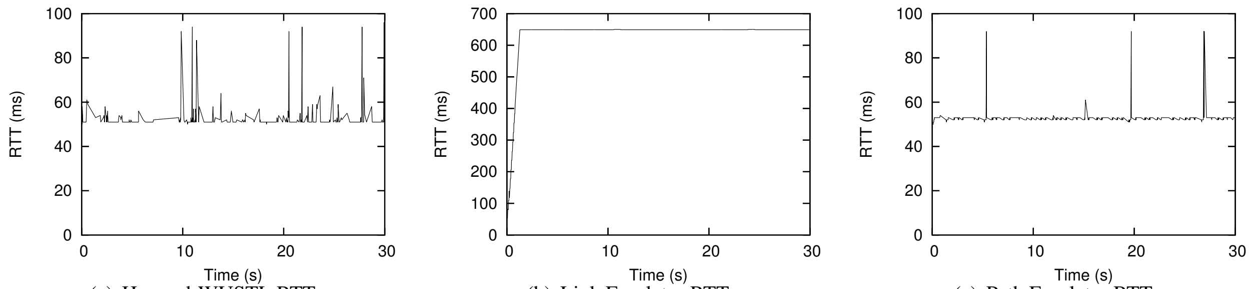
The results of these experiments are shown in Table 2. It is clear from the percent errors that the path emulation achieves higher accuracy than the link emulator in many scenarios. While both achieve within 10% of the specified throughput in the first test (a low-bandwidth, symmetric path), as path asymmetry and bandwidth-delay product increase, the effects discussed in Section 2.2 cause errors in the link emulator. While our path emulator remains within approximately 10% of the target ABW, the link emulator diverges by as much as 66%. The forward direction, with its higher throughput, tends to suffer disproportionately higher error rates. Because the measured values come from real Internet paths, they do not represent unusual or extreme conditions.

The first two rows of synthetic results demonstrate that, even in cases of symmetric bandwidth, the failure to differentiate between capacity and available bandwidth hurts the link emulator's accuracy. The third demonstrates divergence under highly asymmetric conditions.

To evaluate the importance of selecting proper queue sizes, we reran two earlier experiments in our path emulator, this time setting the queue sizes greater than the upper limits allowed by Equation 4. These results are shown in the bottommost section of Table 2 (labeled "Bad Queue Size"). The RTT for each flow grows until the flows reach their maximum window sizes, preventing them from utilizing the full ABW of the emulated path and resulting in large errors.

Test Type	Configured ABW (Kbps) Forward	Configured ABW (Kbps) Reverse	Configured Base Delay (ms)	Emulator	Queue size (KB)	Achieved Tput (Kbps) Forward	Achieved Tput (Kbps) Reverse	ABW Error (%) Forward	ABW Error (%) Reverse
Measured	2,251	2,202	64	link	73	2,070	2,043	8.0	7.2
				path	957	2,202	2,163	2.1	1.8
	4,061	2,838	29	link	73	2,774	2,599	31.7	8.4
				path	957	3,822	2,706	5.8	4.6
	6,436	2,579	12	link	73	3,176	2,358	50.6	8.5
				path	844	6,169	2,448	4.1	5.0
	25,892	17,207	4	link	73	20,608	15,058	20.4	12.5
				path	197	23,237	15,644	10.2	9.1
Synthetic	8,000	8,000	45	link	73	6,228	6,207	22.0	22.4
				path	237	7,493	7,420	6.3	7.2
	12,000	12,000	30	link	73	9,419	9,398	21.5	21.6
				path	158	11,220	11,208	6.5	6.6
	10,000	3,000	30	link	73	3,349	2,705	66.5	9.8
				path	265	9,150	2,690	8.5	10.3
Bad Queue Size	25,892	17,207	4	link	—	—	—	—	—
				path	390	21,012	15,916	18.8	7.5
	10,000	3,000	30	link	—	—	—	—	—
				path	488	7,641	2,768	23.6	7.7

Table 2: Throughput achieved by simultaneous TCP flows along both directions of a number of paths, using a link emulator and using our path emulator.



(a) Harvard-WUSTL RTT (b) Link Emulator RTT (c) Path Emulator RTT

Figure 5: RTT over the lifetime of a 30-second TCP flow. Note that the range of the Y-axis in the center graph is seven times larger than the other two graphs.

4.2 Effect on Round-Trip Time

In addition to TCP throughput, our path emulator also has a significant effect on the RTT observed by a flow, producing RTTs much more similar to those on real paths than those seen in a simple link emulator. To evaluate this difference, we measured the path between the PlanetLab nodes at Harvard and those at Washington University in St. Louis (WUSTL). The ABW was 409 Kbps from Harvard to WUSTL, and 4,530 Kbps from WUSTL to Harvard. The base RTT was 50 ms. To isolate the effects of distinguishing ABW and capacity from other differences between the emulators, we set the queue size in both to the same value (our Linux kernel's maximum window size of 32 KB), and exercised only one direction of the path.

Figure 5 shows the round-trip times seen during a 30-second `iperf` run from Harvard to WUSTL, and the round-trip times seen under both link and path emulation. Both emulators achieved the target bandwidth, but dramatically differ in the round-trip times and packet-loss characteristics of the flows. Figure 5(b) and Figure 5(c) show the round-trip times observed on the link and path emulators respectively. As TCP tends to keeps the bottleneck queue full, it quickly plateaus at the length of the queue in time. Because the link emulator's queue drains at the rate of *ABW*, rather than the much larger rate of *C*, packets spend much longer in the queue in the link emulator. The average RTT for the link emulator was 629 ms, an order of magnitude higher than the average RTT of 53.1 ms observed on the actual path (Figure 5(a)). Because the path emulator separates capacity and ABW, it

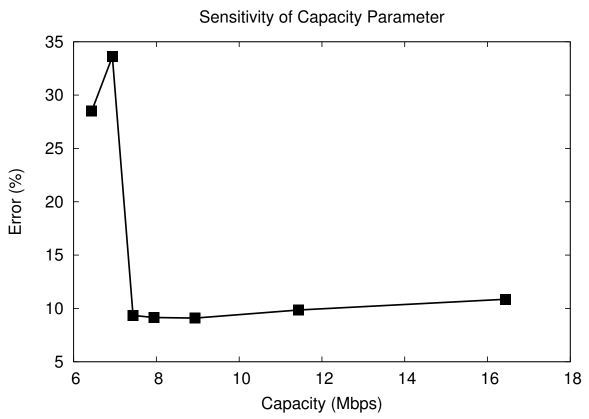


Figure 6: Experiments on an emulated path with 6.5 Mbps available bandwidth in the forward direction. A constant queue size is maintained while capacity is varied.

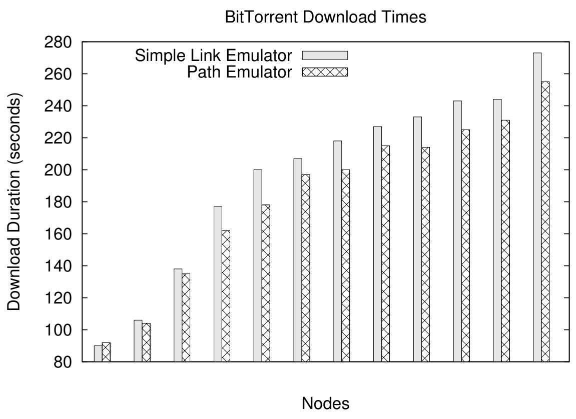


Figure 7: Time taken by participants in a BitTorrent swarm to download a file. Download times are shown for each node for both path and simple link emulation.

gives an average RTT of 53.2 ms, which is within 1% of the value on the real path. The standard deviation inside of the path emulator is 3.0 ms, somewhat lower than the 5.1 ms seen on the real path.

To get comparable RTTs from the link emulator, its queue would have to be much smaller, around 2.5 KB, which is not large enough to hold two full-size TCP packets. We reran this experiment in the link emulator using this smaller queue size, and a unidirectional TCP flow was able to achieve close to the target 409 Kbps throughput. However, when we ran bidirectional flows, the flow along the reverse direction was only able to achieve a throughput of around 200 Kbps, despite the fact that the ABW in that direction was set to 4,530 Kbps (the value measured on the real path). This demonstrates that adjusting queue size by itself is not sufficient to fix excessive RTTs, as it can cause significant errors in ABW emulation.

4.3 Sensitivity Analysis of Capacity

As we saw in Figure 1, once the capacity has grown sufficiently large, it is possible to satisfy both the upper and lower bounds on queue size. Our next experiment tests how sensitive the emulator is to capacity values larger than this intersection point.

We ran several trials with a fixed available bandwidth (6.5 Mbps) but varying levels of capacity. All other parameters were left constant. Figure 6 shows the relative error in achievable throughput as we vary the capacity. While error peaks when capacity is very near available bandwidth, outside of that range, changing the capacity has very little effect on the emulation. This justifies the decision in our implementation to use a fixed, large capacity, rather than measuring it for each path.

4.4 BitTorrent Application Results

We demonstrated in Section 4.1 that using path parameters in a simple link emulator causes artifacts in many situations. We now show that these artifacts cause inaccuracies when running real applications and are not just revealed using measurement traffic. Though this experiment uses multiple paths, to isolate the effects of capacity and queue size, it does not model shared bottlenecks or reactivity.

Figure 7 shows the download times of a group of BitTorrent clients using simple link emulation and path emulation with the same parameters, which were gathered from PlanetLab paths. Each pair of bars shows the time taken to download a fixed file on one of the twelve nodes. The simple link emulator limits available bandwidth inaccurately under some circumstances, which increases the download duration on many of the nodes. As seen in the figure, each node downloads an average of 6% slower in the link emulator than it does when under path emulator. The largest difference is 12%. This shows that the artifacts we observe with micro-benchmarks also affect the behavior of real applications.

4.5 Network Reactivity

Our next experiment examines the fidelity of our reactivity model. We ran reactivity tests on a set of thirty paths between PlanetLab nodes. For each path, we measured aggregate available bandwidth with a varying number of foreground `iperf` flows, ranging from one to eight. We used this data as input to our emulator, in the form of reactivity tables, then repeated the experiments inside of the emulator. In this experiment, the paths are tested independently at different times, so no shared bottlenecks are exercised.

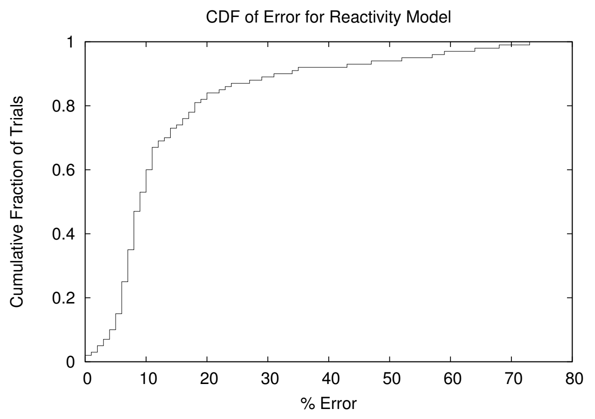


Figure 8: A CDF showing percentage of error over paths with multiple foreground flows.

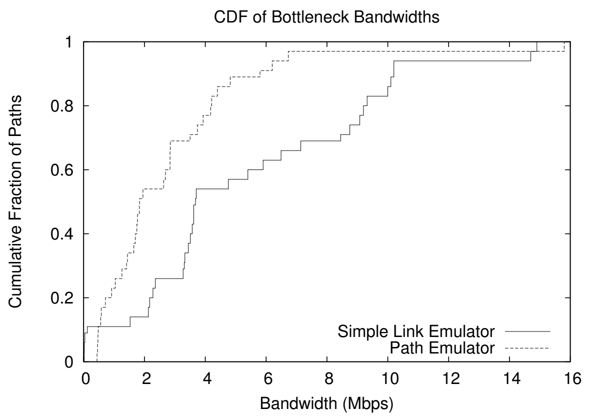


Figure 9: A CDF showing bandwidth achieved at shared bottlenecks.

By comparing the throughputs achieved inside of the emulator to those obtained on the real path, we can test the accuracy of our reactivity model. Figure 8 shows the results of this experiment. For each trial (a specific number of foreground flows over a specific path), we computed the error as the percentage difference between the aggregate bandwidth measured on PlanetLab and that recreated inside the emulator. Our emulator was quite accurate; 80% of paths were emulated to within 20% of the target bandwidth.

There are some outliers, however, with significant error. These point to limitations of our implementation, which currently sets capacities to 100 Mbps and has a 1 MB limit on the bottleneck queue size. Some paths in this experiment had very high ABW: as high as 78 Mbps in aggregate for eight foreground flows. As we saw in Figure 6, when ABW is close to capacity, significant errors can result. With high bandwidths and multiple flows, the lower bound on queue sizes (Equation 2) also becomes quite large, producing two sources of error. First, if this bound becomes larger than our 1 MB implementation limit, we are unable to provide sufficient queue space for all flows to achieve full throughput. Second, our limits on capacity limit the amount we can adjust the upper bound on queue size, Equation 4, meaning that we may end up in a situation where it is not possible to satisfy both the upper and lower bounds.

It would be possible to raise these limits in our implementation by improving bandwidth shaping efficiency and allowing larger queue sizes. The underlying issues are fundamental ones, however, and would reappear at higher bandwidths: our emulator requires that capacity be significantly larger than the available bandwidth to be emulated, and providing emulation for large numbers of flows with high ABW requires large queues.

4.6 Shared Bottlenecks

Finally, we examine the effects of shared bottlenecks on bandwidth. We again measured the paths between a set of PlanetLab nodes, finding their bandwidth, reactivity, and shared bottlenecks After characterizing the paths in the real world, we configured two emulations. The first is a simple link emulation, approximating each path with an independent link emulator. The second uses our full path emulator, including its modeling of shared bottlenecks and reactivity. In order to stress and measure the system, we simultaneously ran an instance of `iperf` in both directions between every pair of nodes. This causes competition on the shared bottlenecks and also ensures that every path is being exercised in both directions at the same time.

Figure 9 shows a CDF of the bandwidth achieved at the bottlenecks in both the link emulator and our path emulator, demonstrating that failure to model shared bottlenecks results in higher bandwidth. To isolate the effects of shared bottlenecks and reactivity, only flows passing through those bottlenecks are shown. In the link emulator, each flow receives the full bandwidth measured for the path. In the path emulator, flows passing through shared bottlenecks are forced to compete for this bandwidth, and as a result, each receives less of it. Modeling of reactivity plays an important role here: in the path emulator, each shared bottleneck is being exercised by multiple flows, and thus the aggregate bandwidth available is affected by the response of the cross-traffic. The few cases in which the path emulator achieves higher bandwidth than the link emulator are caused by highly asymmetric paths, where the effects demonstrated in Section 4.1 dominate.

5 Related Work

There is a large body of work on measuring the Internet and characterizing its paths. The focus of our work is not to create novel measurement techniques, but to create accurate emulations based on existing techniques. Our contribution lies in the identification of principles that can be used to accurately emulate paths, given these measurements.

Our emulator builds on the Emulab [32] and Dummynet [22] link emulators to reproduce measured end-to-end path characteristics. ModelNet [27] also emulates router-level topologies on a link-by-link basis. Capacity and delay are set for each link on the path. To create shared bottlenecks with a certain degree of reactivity, it is up to the experimenter to carefully craft a router topology and introduce cross-traffic on a particular link of the path. ModelNet includes tools for simplifying router-level topologies, but does not abstract them as heavily as we do in this work. NIST Net [7], a Linux-based network emulator, is an alternative to Dummynet. However, it is also a link emulator and does not distinguish between capacity and available bandwidth. Our model abstracts the important characteristics of the path, thereby simplifying their specification and faithfully reproducing those network conditions without the need for a detailed router-level topology.

Appenzeller et al. [1] show that the queuing buffer requirements for a router can be reduced provided that a large number of TCP flows are passing through the router and they are desynchronized. They also provide reasoning as to why setting the queue sizes to the bandwidth-delay product works for a reasonably small number of TCP flows. We use the bandwidth delay product as the lower limit on the queue sizes of the paths being modeled. We are also concerned about low capacity links (asymmetric or otherwise) causing large queuing delays that adversely affect the throughput of TCP. Our model separates capacity from available bandwidth and determines queue sizes such that the TCP flows on the path do not become window-size limited.

Researchers have investigated the effects of capacity and available bandwidth asymmetry on TCP performance [2, 3, 11, 14]. They proposed modifications to either the bottleneck router forwarding mechanism, or the end node TCP stack. We do not seek to minimize the queue sizes at the router, but rather to calculate the right queue size for a path to enable the foreground TCP flows to fully utilize the ABW during emulation. We modify neither router forwarding nor the TCP stack and our model is independent of the TCP implementation used on the end nodes.

Harpoon [24], Swing [28], and Tmix [31] are frameworks that characterize the traffic passing through a link and then generate statistically similar traffic for emulating that link or providing realistic workloads. Our work, in contrast, does not seek to characterize or recreate background traffic in great detail. We characterize cross-traffic at a much higher level, solely in terms of its reactivity to foreground flows. We are able to do this characterization with end-to-end measurements, and do not need to directly observe the packets comprising the cross-traffic.

6 Conclusion and Future Work

We have presented and evaluated a new path emulator that can accurately recreate the observed end-to-end conditions of Internet paths. The path model within our emulator is based on four principles that combine to enable accurate emulation over a wide range of conditions. We have compared our approach to two alternatives that make use of simple link emulation. Unlike router-level emulation of paths, our approach is suitable for reconstructing real paths solely from measurements taken from the edges of a network. As we have shown, using a single link emulator to approximate a measured multi-hop path can fail to produce accurate results. Our path model corrects these problems, enabling recreations of real paths in the repeatable, controlled environment of an emulator.

Much of our future work will concentrate on improving the reactivity portion of our model. Our method of measuring reactivity is currently the most intensive part of our data gathering: it uses the most bandwidth, and takes the most time. Improving it will allow our system to run at larger scale. Viewing ABW as a function of the number of full-speed foreground TCP flows limits us both to TCP and to applications that are able to fill their network paths. In future refinements of our design, we hope to characterize ABW in terms of lower-level metrics that are not intrinsically linked to TCP's congestion control behavior. Finally, our averaging of ABW values for paths that share a bottleneck could use more study and validation.

Another future direction will be the expansion of our work to the simulation domain. Simulators handle links and paths in much the same way as do emulators, and the model we describe in Section 2 can be directly applied to them as well.

Acknowledgments

We thank our colleagues Eric Eide and Mike Hibler for their comments and proofreading assistance. Eric provided significant typesetting help as well. We also thank the anonymous NSDI reviewers, whose comments helped us to improve this paper greatly. This material is based upon work supported by the National Science Foundation under Grant Nos. 0205702, 0335296, 0338785, and 0709427.

References

[1] G. Appenzeller, I. Keslassy, and N. McKeown. Sizing router buffers. In *Proc. SIGCOMM*, pages 281–292, Portland, OR, Aug.–Sept. 2004.

[2] H. Balakrishnan, V. Padmanabhan, G. Fairhurst, and M. Sooriyabandara. TCP performance implications of network path asymmetry. Internet RFC 3449, Dec. 2002.

[3] H. Balakrishnan, V. N. Padmanabhan, and R. H. Katz. The effects of asymmetry on TCP performance. In *Proc. MobiCom*, pages 77–89, Budapest, Hungary, 1997.

[4] A. Bharambe, J. Pang, and S. Seshan. Colyseus: A distributed architecture for online multiplayer games. In *Proc. NSDI*, pages 155–168, San Jose, CA, May 2006.

[5] J. M. Blanquer, A. Batchelli, K. Schauser, and R. Wolski. Quorum: Flexible quality of service for Internet services. In *Proc. NSDI*, pages 159–174, Boston, MA, May 2005.

[6] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM*, pages 24–35, London, England, Aug. 1994.

[7] M. Carson and D. Santay. NIST Net: a Linux-based network emulation tool. *Comput. Commun. Rev.*, 33(3):111–126, July 2003.

[8] P. B. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proc. SIGCOMM*, pages 147–158, Pisa, Italy, Sept. 2006.

[9] M. Jain and C. Dovrolis. End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. *IEEE/ACM Trans. Networking*, 11(4):537–549, Aug. 2003.

[10] D. Johnson, D. Gebhardt, and J. Lepreau. Towards a high quality path-oriented network measurement and storage system. In *Proc. PAM*, pages 102–111, Cleveland, OH, Apr. 2008.

[11] L. Kalampoukas, A. Varma, and K. K. Ramakrishnan. Improving TCP throughput over two-way asymmetric links: Analysis and solutions. In *Proc. SIGMETRICS*, pages 78–89, Madison, WI, June 1998.

[12] M. S. Kim et al. A wavelet–based approach to detect shared congestion. In *Proc. SIGCOMM*, pages 293–306, Portland, OR, Aug.–Sept. 2004.

[13] K. Lai and M. Baker. Nettimer: a tool for measuring bottleneck link, bandwidth. In *Proc. USITS*, San Francisco, CA, Mar. 2001.

[14] T. V. Lakshman, U. Madhow, and B. Suter. Window-based error recovery and flow control with a slow acknowledgement channel: A study of TCP/IP performance. In *Proc. INFOCOM*, pages 1199–1209, Kobe, Japan, Apr. 1997.

[15] R. Morris. TCP behavior with many flows. In *Proc. ICNP*, pages 205–211, Washington, DC, Oct. 1997.

[16] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Reno Performance: A Simple Model and Its Empirical Validation. *IEEE/ACM Trans. Networking*, 8(2):133–145, Apr. 2000.

[17] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. HotNets–I*, Princeton, NJ, Oct. 2002.

[18] H. Pucha, D. G. Andersen, and M. Kaminsky. Exploiting similarity for multi-source downloads using file handprints. In *Proc. NSDI*, pages 15–28, Cambridge, MA, Apr. 2007.

[19] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proc. SIGCOMM*, pages 367–378, Portland, OR, Aug.–Sept. 2004.

[20] V. Ribeiro et al. pathChirp: Efficient available bandwidth estimation for network paths. In *Proc. PAM*, San Diego, CA, 2003.

[21] R. Ricci et al. The Flexlab approach to realistic evaluation of networked systems. In *Proc. NSDI*, pages 201–214, Cambridge, MA, Apr. 2007.

[22] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *Comput. Commun. Rev.*, 27(1):31–41, Jan. 1997.

[23] A. Shieh, A. C. Myers, and E. G. Sirer. Trickles: a stateless network stack for improved scalability, resilience, and flexibility. In *Proc. NSDI*, pages 175–188, Boston, MA, May 2005.

[24] J. Sommer and P. Barford. Self-configuring network traffic generation. In *Proc. IMC*, pages 68–81, Taormina, Italy, Oct. 2004.

[25] N. Spring, L. Peterson, A. Bavier, and V. Pai. Using PlanetLab for network research: myths, realities, and best practices. *Oper. Syst. Rev.*, 40(1):17–24, 2006.

[26] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. NSDI*, pages 253–266, San Jose, CA, May 2006.

[27] A. Vahdat et al. Scalability and accuracy in a large-scale network emulator. In *Proc. OSDI*, pages 271–284, Boston, MA, Dec. 2002.

[28] K. V. Vishwanath and A. Vahdat. Realistic and responsive network traffic generation. In *Proc. SIGCOMM*, pages 111–122, Pisa, Italy, Sept. 2006.

[29] K. V. Vishwanath and A. Vahdat. Evaluating distributed systems: Does background traffic matter? In *Proc. USENIX*, pages 227–240, Boston, MA, June 2008.

[30] M. Walfish et al. DDoS defense by offense. In *Proc. SIGCOMM*, pages 303–314, Pisa, Italy, Sept. 2006.

[31] M. C. Weigle et al. Tmix: a tool for generating realistic TCP application workloads in ns-2. *Comput. Commun. Rev.*, 36(3):65–76, July 2006.

[32] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Boston, MA, Dec. 2002.

MODIST: Transparent Model Checking of Unmodified Distributed Systems

Junfeng Yang[○*], Tisheng Chen[‡], Ming Wu[‡], Zhilei Xu[‡], Xuezheng Liu[‡]
Haoxiang Lin[‡], Mao Yang[‡], Fan Long[†], Lintao Zhang[‡*], Lidong Zhou[‡*]
[○] Columbia University, [‡] Microsoft Research Asia
[*] Microsoft Research Silicon Valley, [†] Tsinghua University

Abstract

MODIST is the first model checker designed for *transparently* checking *unmodified* distributed systems running on *unmodified* operating systems. It achieves this transparency via a novel architecture: a thin interposition layer exposes all actions in a distributed system and a centralized, OS-independent model checking engine explores these actions systematically. We made MODIST practical through three techniques: an execution engine to simulate consistent, deterministic executions and failures; a *virtual clock* mechanism to avoid false positives and false negatives; and a state exploration framework to incorporate heuristics for efficient error detection.

We implemented MODIST on Windows and applied it to three well-tested distributed systems: Berkeley DB, a widely used open source database; MPS, a deployed Paxos implementation; and PACIFICA, a primary-backup replication protocol implementation. MODIST found 35 bugs in total. Most importantly, it found *protocol-level* bugs (i.e., flaws in the core distributed protocols) in every system checked: 10 in total, including 2 in Berkeley DB, 2 in MPS, and 6 in PACIFICA.

1 Introduction

Despite their growing popularity and importance, distributed systems remain difficult to get right. These systems have to cope with a practically infinite number of network conditions and failures, resulting in complex protocols and even more complex implementations. This complexity often leads to corner-case errors that are difficult to test, and, once detected in the field, impossible to reproduce.

Model checking has been shown effective at detecting subtle bugs in real distributed system implementations [19, 27]. These tools systematically enumerate the possible execution paths of a distributed system by starting from an initial state and repeatedly performing all possible actions to this state and its successors. This *state-space exploration* makes rare actions such as network failures appear as often as common ones, thereby quickly driving the target system (i.e., the system we check) into corner cases where subtle bugs surface.

To make model checking effective, it is crucial to expose the actions a distributed system can perform and do so at an appropriate level. Previous model checkers for distributed systems tended to place this burden on users, who have to either write (or rewrite) their systems in a restricted language that explicitly annotates event handlers [19], or heavily modify their system to shoehorn it into a model checker [27].

This paper presents MODIST, a system that checks *unmodified* distributed systems running on *unmodified* operating systems. It simulates a variety of network conditions and failures such as message reordering, network partitions, and machine crashes. The effort required to start checking a distributed system is simply to provide a simple configuration file specifying how to start the distributed system. MODIST spawns this system in the native environment the system runs within, infers what actions the system can do by *transparently* interposing between the application and the operating system (OS), and systematically explores these actions with a centralized, OS-independent model checking engine. We have carefully engineered MODIST to ensure the executions MODIST explores and the failures it injects are consistent and deterministic: inconsistency creates false positives that are painful to diagnose; non-determinism makes it hard to reproduce detected errors.

Real distributed systems tend to rely on timeouts for failure detection (e.g., leases [14]); many of these timeouts hide in branch statements (e.g., "`if(now > t + timeout)`"). To find bugs in the rarely tested timeout handling code, MODIST provides a *virtual clock* mechanism to explore timeouts systematically using a novel static symbolic analysis technique. Compared to the state-of-the-art symbolic analysis techniques [3, 4, 13, 31], our method reduces analysis complexity using the following two insights: (1) programmers use time values in *simple* ways (e.g., arithmetic operations) and (2) programmers check timeouts *soon* after they query the current time (e.g., by calling `gettimeofday()`).

We implemented MODIST on Windows. We applied it to three well-tested distributed systems: Berkeley DB, a widely used open-source database; MPS, a Paxos implementation that has managed production data centers with more than 100K machines for over two years; and PACIFICA, a primary-backup replication protocol implementation. MODIST found 35 bugs in total. In particular, it found *protocol-level* bugs (i.e., flaws in the core protocols) in every system checked: 10 in total, including 2 in Berkeley DB, 2 in MPS, and 6 in PACIFICA. We measured the speed of MODIST and found that (1) MODIST incurs reasonable overhead (up to 56.5%) as a checking tool and (2) it can *speed up* a checked execution (up to 216 times faster) using its virtual clock.

MODIST provides a customizable framework for incorporating various state-space exploration strategies. Using this framework, we implemented dynamic partial order reduction (DPOR) [9], random exploration, depth-first exploration, and their variations. Among these, DPOR is a strategy well-known in the model checking community for avoiding redundancy in exploration. To evaluate these strategies, we measured their protocol-level coverage (i.e., unique protocol states explored). The results show that, while DPOR achieves good coverage for a small bounded state space, it scales poorly as the state space grows; a more balanced variation of DPOR, with a set of randomly selected paths as starting points, achieves the best coverage.

This paper is organized as follows. We present an overview of MODIST (§2), then describe its implementation (§3) and evaluation (§4). Next we discuss related work (§5) and conclude (§6).

2 Overview

A typical distributed system that MODIST checks has multiple processes,[*] each running multiple threads. These processes communicate with each other by sending and receiving messages through socket connections. MODIST can re-order messages and inject failures to simulate an *asynchronous and unreliable* network. The processes may write data to disk, and MODIST will generate different possible crash scenarios by permuting these disk writes.

The remainder of this section gives an overview of MODIST, covering its architecture (§2.1), its checking process(§ 2.2), the checks it enables (§ 2.3), and its user interface (§2.4).

2.1 Architecture

Figure 1 illustrates the architecture of MODIST applied to a 4-node distributed system. The master node runs multiple threads (the curved lines in the figure) and might send or receive messages (the solid boxes). For each process in the target system, MODIST inserts an interposition frontend between the process and its native operating system to intercept and control non-deterministic decisions involving thread and network operations.

MODIST further employs a backend that runs in a different address space and communicates with the frontends via RPC. This design minimizes MODIST's perturbation of the target system, allowing us to build a generic backend that runs on a POSIX-compliant operating system, and makes it possible to build the frontends for MODIST on different operating systems. The backend consists of five components: a dependency tracker, a failure simulator, a virtual clock manager, a model checking engine, and a global assertion checker.

[*]In this paper we use node and process interchangeably.

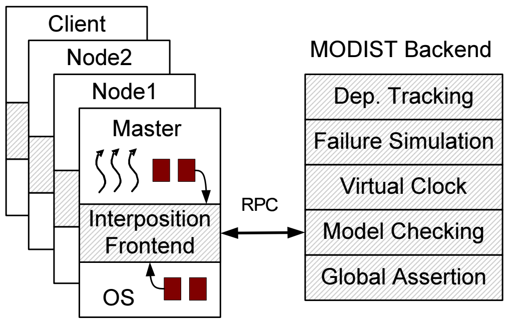


Figure 1: MODIST *architecture.* All MODIST components are shaded. The target system consists of one master, two replication nodes, and one client. MODIST's frontend interposes between each process in the target system and the operating system to intercept and control non-deterministic actions, such as message interleaving and thread interleaving. MODIST's backend runs in a separate address space to schedule these actions.

Interposition. MODIST's interposition frontend is a thin layer that exposes what actions a distributed system can do and lets MODIST's backend deterministically schedule them. Specifically, it does so in two steps: (1) when the target system is about to execute an action, the frontend pauses it and reports it to the backend; and (2) upon the backend's command, the frontend either resumes or fails the paused action, turning the target system into a "puppet" of the backend.

We place the interposition layer at the OS-application boundary to avoid modifying either the target system or the underlying operating system. In addition, despite variations in OS-application interfaces, they provide similar functions, allowing us to build a generic backend.

Since the interposition layer runs inside the target system, we explicitly design it to be simple and mostly stateless, and leave the logic and the state in the backend, thereby reducing the perturbation of the target system.

Dependency Tracking. MODIST's dependency tracker oversees how actions interfere with each other. It uses these dependencies to compute the set of *enabled* actions, i.e., the actions, if executed, that will not block in the OS. For example, a `recv()` is enabled if there is a message to receive, and disabled otherwise. The model checking engine (described below) only schedules enabled actions, because scheduling a disabled action will deadlock the target system (analogous to a cooperative thread scheduler scheduling a blocked thread).

Failure Simulation. MODIST's failure simulator or-

```
# command           working dir  inject failure?
  master.exe         ./master/         1
  node.exe           ./node1/          1
  node.exe           ./node2/          1
  client.exe test1   ./client/         0
```

Figure 2: A configuration file that spawns the distributed system in Figure 1. We used this file to check PACIFICA.

chestrates many rare events that may occur in a distributed system, including message reordering, message loss, network partition, and machine crashes; these events can expose bugs in the often-untested failure handling code. The failure simulator lets MODIST inject these failures as needed, consistently to avoid false positives, and deterministically to let users reliably reproduce errors (cf. §2.2).

Virtual Clock. MODIST's virtual clock manager has two main functions: (1) to discover timers in the target system and fire them as requested by MODIST's model checking engine to trigger more bugs, and (2) to ensure that all processes in the target system observe a consistent clock to avoid false positives. Since the clock is virtual, MODIST can "fast forward" the clock as needed, often making a checked execution faster than a real one.

Model Checking. MODIST's model checking engine acts as an "omnipresent" scheduler of the target system. It systematically explores a distributed system's executions by enumerating the actions, failures, and timers exposed by the other MODIST components. It uses a set of search heuristics and state-space reduction techniques to improve the efficiency of its exploration. We elaborate the model checking process in next section and the search strategies in §3.6.

Global Assertion. MODIST's global assertion mechanism lets users check distributed properties on consistent global snapshots; these properties cannot be checked by observing only the local states at each individual node. Its implementation leverages our previous work [25].

2.2 Checking Process

With all MODIST's components in place, we now describe MODIST's checking process. To begin checking a distributed system, the user only needs to prepare a simple configuration file that specifies how to start the target system. Figure 2 shows a configuration file for the 4-node replication system shown in Figure 1; it is a real configuration that we used to check PACIFICA. Each line in the configuration tells MODIST how to start a process in the target system. A typical configuration consists of 2 to 10 processes. The "inject failure" flag is useful when users do not want to check failures for a process. For example, `client.exe` is an internal test program that does not handle any failures, so we turned off failure checking for this process.

```
init_state = checkpoint(create_init_state());
q.enqueue(init_state, init_state.actions);

while(!q.empty()) {
  <state, action> = q.dequeue();
  try {
    next_state = checkpoint(action(restore(state)));
    global_assert(next_state); //check user-provided global assertions
    if (next_state has never been seen before)
      q.enqueue(next_state, next_state.actions);
  } catch (Error e) {
    // save trace and report error
    ...
  }
}
```

Figure 3: Model checking pseudo-code.

With a configuration file, users can readily start checking their systems by running `modist <config>`. MODIST then instruments the executables referred to in the configuration file to interpose between the application and the operating system, and starts its model checking loop to explore the possible *states* and *actions* in the target system: a *state* is an instantaneous snapshot of the target system, while an *action* can be to resume a paused WinAPI function via the interposition layer, to inject a failure via the failure simulator, or to fire a timer via the virtual clock manager.

Figure 3 shows the pseudo-code of MODIST's model checking loop. MODIST first spawns the processes specified in the configuration to create an *initial state*, and adds all $\langle initial\ state, action\rangle$ pairs to a *state queue*, where *action* is an action that the target system can do in the initial state. Next, MODIST takes a $\langle state, action\rangle$ pair off the state queue, restores the system to *state*, and performs *action*. If the action generates an error, MODIST will save a trace and report the error. Otherwise, MODIST invokes the user-provided global assertions on the resultant global state. MODIST further adds new state/action pairs to the state queue based on one of MODIST's search strategies (cf. §3.6 for details.) Then, it takes off another $\langle state, action\rangle$ pair and repeats.

To implement the above process, MODIST needs to checkpoint and restore states. It uses a *stateless* approach [12]: it checkpoints a state by remembering the actions that created the state and restores it by redoing all the actions. Compared to a stateful approach that checkpoints a state by saving all the relevant memory bits, a stateless approach requires little modifications to the target system, as previous work has shown [12, 19, 28, 39].

2.3 Checks

The checks that MoDist performs include generic checks that require no user intervention as well as user-written system-specific checks.

Currently, MoDist detects two classes of generic errors. The first is "fail-stop" errors, which manifest themselves when the target system unexpectedly crashes in the absence of an injected crash from MoDist. These crashes can be segmentation faults due to memory errors or program aborts because MoDist has brought the target system into an erroneous state. MoDist detects these unexpected crashes by catching the corresponding signals. The second is "divergence" errors [12], which manifest themselves when the target system deadlocks or goes into an infinite loop. MoDist catches these errors using timeouts. When MoDist schedules one of the actions of the target system, it waits for a user-specified timeout interval (10 seconds by default) until the target system gets back to it; otherwise, MoDist will flag a divergence error.

Because MoDist checks the target system by executing it, MoDist can easily check the effects of real executions and find errors. Thus, we can always combine MoDist with other dynamic error detection tools (e.g., Purify [16] and Valgrind [29]) to check more generic properties; we leave these checks for future work.

In addition to generic checks, MoDist can perform system-specific checks via user-provided assertions, including local assertions (via the `assert()` statements) inserted into the target system and global assertions that run in the centralized model checking engine. Given these assertions, MoDist will amplify them by driving the target code into many possible states where these assertions may fail. In general, the more assertions users add, the more effective MoDist will be.

2.4 Advanced User Interface

As with most other automatic error detection tools, the more system-specific knowledge MoDist has, the more effective it will be. For users who want to check their system more thoroughly, MoDist provides the following methods for incorporating domain knowledge.

Users can add more program assertions in the code for a more thorough check. In addition to these local assertions, users can enrich the set of checks by specifying *global assertions* in MoDist. These assertions check distributed properties on any consistent global snapshot.

Users can make MoDist more effective by reducing their system's state space. A simple trick is to bound the number of failures MoDist injects per execution. Our previous work [38, 39] showed that tricky bugs are often caused by a small number of failures at critical moments. Obviously, without bounds on the number of failures, a distributed system may keep failing without making any progress. In addition, developers tend to find bugs triggered by convoluted failures uninteresting [38].

Users can provide hints to let MoDist focus on the states (among an infinite number of states) that users consider most interesting. Users can do so in two ways: (1) extend one of MoDist's search algorithms through the well-defined state queue interface, and (2) construct a test case to test some unusual parts of the state space.

3 Implementation

We implemented MoDist on Windows by intercepting calls to WinAPI [36], the Windows Application Programming Interface. We chose WinAPI because it is the predominant programming interface used by almost all Windows applications and libraries, including the default POSIX implementation on Windows. While we built MoDist on Windows, we expect that porting to other operating systems, such as Linux, BSD, and Solaris, should be easy because WinAPI is more complicated than the POSIX API provided by most other operating systems. For example, WinAPI has several times as many functions as POSIX. Moreover, many WinAPI functions operate in both synchronous and asynchronous mode, and the completion notifications of asynchronous IO (AIO) may be delivered through several mechanisms, such as events, *select*, or IO completion ports [36].

When we implemented MoDist we tried to adhere to the following two goals:

1. *Consistent and deterministic execution.* The executions MoDist explores and the failures it injects should be consistent and deterministic to avoid difficult-to-diagnose false positives and non-deterministic errors.
2. *Tailor for distributed systems.* We explicitly designed MoDist to check distributed systems. Having this goal in mind, we customized our implementation for distributed systems and avoided being overly general.

These goals were reflected at many places in our implementation. In the rest of this section, we describe MoDist's implementation in details, highlighting the decisions entailed by these goals.

3.1 Interposition

MoDist's interposition layer transparently intercepts the WinAPI functions in the target system and allows MoDist's backend to control it deterministically. There are two main issues regarding interposition. First, *interposition complexity*: since the interposition layer runs inside the address space of the target system, it should be as simple as possible to avoid perturbing the target system, or introducing inconsistent or non-deterministic executions. Second, *IO abstraction*: as previously mentioned, WinAPI is a wide interface with rich semantics; Windows networking IO is particularly complex. To avoid

Category	# of functions	# of LOC
Network	28	1816
Time	7	161
File System	9	640
Mem	5	126
Thread	33	1433
Shared		1290
Total	82	5466

Table 1: *Interposition complexity.* This table shows the lines of code for WinAPI wrappers, broken down by categories. The "Shared" row refers to the code shared among all API categories. Most wrappers are fairly small (67 lines on average).

excessive complexity in MODIST's backend, the interposition layer should abstract out the semantics irrelevant to checking and abstract the WinAPI networking interface to a simpler form.

Interposition complexity. To reduce the interposition complexity, we implemented the interposition layer using the binary instrumentation toolkit from our previous work [25]. This toolkit takes a list of annotated WinAPI functions we want to hook and automatically generates much of the wrapper code for interposition. Under the hood, it intercepts calls to dynamically linked libraries by overwriting the function addresses in relocation tables (*import tables* in Windows terminology).

Since we check distributed systems, we only need to intercept WinAPIs relevant to these systems. Table 1 shows the categories of WinAPIs we currently hook: (1) networking APIs, such as `WSARecv()` (receiving a message), for exploring network conditions; (2) time APIs, such as `GetSystemTime()`, for discovering timers; (3) file system APIs, such as `WriteFile()` and `FlushFileBuffers()`, for injecting disk failures and simulating crashes, (4) memory APIs, such as `malloc()`, for injecting memory failures; and (5) thread APIs, such as `CreateThread()` and `SetEvent()`, for scheduling threads.

Most WinAPI wrappers are simple: they notify MODIST's backend about the WinAPI calls using an RPC call, wait for the reply from the backend, and, upon receiving the reply, they either call the underlying WinAPIs or inject failures. Table 1 shows the total lines of code in all manually-written wrappers. Each wrapper on average consists of only 67 lines of code.

IO abstraction. Controlling the Windows networking IO interface is complex for three reasons: (1) there are many networking functions; (2) these functions heavily use AIO, whose executions are hidden inside the kernel and not exposed to MODIST; and (3) these functions may produce non-deterministic results due to failures in the network. We addressed these issues using three methods: (1) abstracting similar network operations into one generic operation to narrow the networking IO interface, (2) exposing AIO to MODIST by running it synchronously in a *proxy thread*, and (3) carefully placing error injection points to avoid non-determinism.

To demonstrate our methods, we show in Figure 4 the wrapper for `WSARecv()`, a WinAPI function to synchronously or asynchronously receive data from a socket. For simplicity, we omit error-handling code and assume AIO completion is delivered using events only (events are similar to binary semaphores.)

Our wrapper first checks whether the network connection represented by the socket argument `s` is already broken by MODIST (line 5–8). If so, it simply returns an error to avoid inconsistently returning success on a broken socket. It then handles AIO (line 9–24) by creating a generic network IO structure `net_io` (line 10–14), hijacking the application's IO completion event (line 16–18), spawning a proxy thread (line 21), and issuing the AIO to the OS (line 23). The proxy thread will invoke function `mc::net_io::run()` (line 29–55). This function first notifies MODIST about the IO (line 34). Upon MODIST's reply, it either injects a failure (line 36–40), or waits for the OS to complete the IO (line 40–51). Function `run()` then reports the IO result to MODIST, which in this example is the length of the data received (47–50). Finally, it calls the wrapper to `SetEvent()` to wake up any real threads in the target system that are waiting for the IO to complete.

This wrapper example demonstrates the abstraction we use between MODIST's interposition frontend and the backend. A network IO is split into an `io_issue` and an `io_result` RPC. The first RPC, `io_issue`, expresses the IO intent of the target system to MODIST before it proceeds to a potentially blocking IO, letting MODIST avoid scheduling a disabled (i.e., blocked) IO. Its second purpose is to serve as a failure injection point. The second RPC, `io_result`, lets MODIST update the control information it tracks.

These RPC methods take the message sizes and the network connections as arguments, but not the specific message buffers or sockets, which may change across different executions. This approach ensures that MODIST's backend sees the same RPC calls when it replays the actions to recreate the same state as when it initially created the state. If MODIST detects a non-deterministic replay (e.g., a `WSARecv()` receives fewer bytes than expected), it will retry the IO by default.

There are two additional nice features about our IO abstraction: (1) it allows wrapper code sharing and therefore reduces the interposition complexity (Table 1,

```
1 : // the OS uses lpOverlap to deliver IO completion
2 : int mc_WSARecv(SOCKET s, LPWSABUF buf, DWORD nbuf,
3 :           . . ., LPWSAOVERLAPPED lpOverlap, . . .) {
4 :   // check if MODIST has broken this connection
5 :   if(mc_socket_is_broken(s)) {
6 :     ::WSASetLastError(WSAENETRESET);
7 :     return SOCKET_ERROR;
8 :   }
9 :   if(overlap) { // Asynchronous mode
10:     mc::net_io *io = . . .;
11:     io−>orig_lpOverlap = lpOverlap;
12:     io−>op = mc::RECV_MESSAGE; // set IO type
13:     io−>connection = . . .; // Identify connection using
14:           // source <ip, port> and destination <ip, port>
15:
16:     // Hijack application's IO completion notification event
17:     io−>orig_event = lpOverlap−>hEvent;
18:     lpOverlap−>hEvent = io−>proxy_event;
19:
20:     // Create a proxy thread and run mc::net_io::run
21:     io−>start_proxy_thread();
22:     // Issue asynchronous receive to the OS
23:     return ::WSARecv(s,buf,nbuf,. . .,io−>proxy_lpOverlap,. . .);
24:   }
25:   // Synchronous mode
26:   . . .
27: }
28: // mc::net_io code is shared among all networking IO
29: void mc::net_io::run() {// called by proxy thread
30:   mc::rpc_client *rpc = mc::current_thread_rpc_client();
31:
32:   // This RPC blocks this thead. It returns only when MODIST
33:   // wants to (1) inject a failure, or (2) complete the IO
34:   int ret = rpc−>io_issue(this−>op, this−>connection);
35:
36:   if(ret == mc::FAILURE) {
37:     // MODIST wants to inject a failure
38:     this−>orig_lpOverlap−>Internal // Fake an IO failure
39:       = STATUS_CONNECTION_RESET;
40:       . . . // Ask the OS to cancel the IO
41:   } else { // MODIST wants to complete this IO
42:     // Wait for the OS to actually complete the IO, because the
43:     // data to receive may still be in the real network.
44:     // This wait will not block forever, since MODIST's
45:     // dependency tracker knows there are bytes to receive
46:     ::WaitForSingleObject(this−>proxy_event, INFINITE);
47:
48:     // Report the bytes actually sent or received, so MODIST's
49:     // dep. tracker knows how many bytes are in the network.
50:     int msg_size = this−>orig_lpOverlap−>InternalHigh;
51:     rpc−>io_result(this−>op, this−>connection, msg_size);
52:   }
53:   // deliver IO notification to application. mc_SetEvent is
54:   // a wrapper to WinAPI SetEvent;
55:   mc_SetEvent(this−>orig_event);
56: }
```

Figure 4: Simplified `WSARecv()` wrapper.

"Shared" row), (2) it abstracts away the OS-specific features and enables the backend to be OS-agnostic.

3.2 Dependency Tracking

MODIST's dependency tracker monitors how actions might affect each other. The notion of dependency is from [12]: two actions are dependent if one can enable or disable the other or if executing them in a different order leads to a different state. MODIST uses these dependencies to avoid false deadlocks (described below), to simulate failures (§3.3), and to reduce state space (§3.6).

To avoid false deadlocks, MODIST needs to compute the set of enabled actions that will not block in the OS. For determinism, MODIST schedules one action at a time and pauses all other actions (cf. §2.2). If MODIST incorrectly schedules a disabled action (such as a blocking `WSARecv()`), it will deadlock the target system because the scheduled action is blocked in the OS while all other actions are paused by MODIST.

Since the dependency tracker tries to infer whether the OS scheduler would block a thread in a WinAPI call (recall that the interposition layer exposes AIOs as threads), it unsurprisingly resembles an OS scheduler and replicates a small amount of the control data in the OS and the network. To illustrate how it works, consider the `WSARecv()` wrapper in Figure 4. The dependency tracker will track precisely how many bytes are sent and received for each network connection using the `io_result` RPC (line 50). If a thread tries to receive a message (line 34) when none is available, the dependency tracker will mark this thread as disabled and place it on the wait queue of the connection. Later, when a `WSASend()` occurs at the other end of the connection, the dependency tracker will remove this thread from the wait queue and mark it as enabled. When MODIST schedules this thread by replying to its RPC `io_issue`, the thread will not block at line 45 because there is data to receive. In addition to network control data, the dependency tracker also tracks threads, locks, and semaphores.

3.3 Failure Simulation

When requested by the model checking engine, MODIST's failure simulator injects five categories of failures: API failures (e.g., `WriteFile()` returns "disk error"), message reordering,[*] message loss, network partitions, and machine crashes. Simulating API failures is the easiest: MODIST simply tells the interposition layer to return an error code. Reordering messages is also easy since the model checking engine already explores different orders of actions. To simulate different crash scenarios, we used techniques from our previous work [38, 39] to permute the disk writes that a system issues.

Simulating network failures is more complicated due to the consistency and determinism requirement. We first tried a naïve approach: simply closing sockets to simulate connection failures. This approach did not work well because we frequently experienced inconsistent failures:

[*]Message reordering is not a failure, but since it is often caused by abnormal network delay, for convenience we consider it as a failure.

the "macro" failures we want to inject (e.g., network partition) map to not one but a set of "micro" failures we can inject through the interposition layer (e.g., a failed `WSARecv()`). For example, to break a TCP connection, we must carefully fail all pending asynchronous IOs associated with the connection at both endpoints. Otherwise, the target system may see an inconsistent connection status and crash, thus generating a false positive.

We also frequently experienced non-deterministic failures because the OS detects failures using non-deterministic timeouts. Consider the following actions:

1. Process P_1 calls WSASend(P_2, message).
2. Process P_2 calls asynchronous WSARecv(P_1).
3. MoDist breaks the connection between P_1 and P_2.

P_2 may or may not receive the message, depending on when P_2's OS times out the broken connection.

Our current approach ensures that failure simulation is consistent and deterministic as follows. We know the exact set of real or proxy threads that are paused by MoDist in `rpc->io_issue()` (Figure 4, line 34). To simulate a network failure, we inject failures to all these threads, and we do so immediately to avoid any non-deterministic kernel timeouts. Note that doing so in the example above will not cause us to miss the scenario where P_2 receives the message before the connection breaks; MoDist will simply explore this scenario in a different execution where it completes P_2's asynchronous `WSARecv()` first (by replying to P_2's `io_issue()` RPC), and then breaks the connection between P_1 and P_2.

3.4 Virtual Clock

MoDist's virtual clock manager injects timeouts when requested by the model checking engine and provides a consistent view of the clock to the target system. A side benefit of virtual clock is that, the target system may run faster because the virtual clock manager can fast forward time. For example, when the target system calls `sleep(1000)`, the virtual clock manager can add 1000 to its current virtual clock and let the target system wake up immediately.

Discovering Timeouts. To detect bugs in rarely tested timeout handling code, we want to discover as many timers as possible. This task is made difficult because system code extensively uses *implicit timers* where the code first gets the current time (e.g., by calling `gettimeofday()`), then checks if a timeout occurs (e.g., using an `if`-statement). Figure 5 shows a real example in Berkeley DB.

Since implicit timers do not use OS APIs to check timeouts, they are difficult to discover by a model checker. Previous work [19, 27, 38] requires users to manually annotate implicit timers.

```
// db-4.7.25.NC/repmgr/repmgr_sel.c
int __repmgr_compute_timeout(ENV *env, timespec * timeout)
{
  db_timespec now, t;
  ...  // Set t to the first due time.
  if (have_timeout) {
    __os_gettime(env, &now, 1); // Query current time.
    if (now >= t) // Timeout check, immediately follows the query.
      *timeout = 0; // Timeout occurs.
    else
      *timeout = t − now; // No timeout.
  }
  ...
}
```

Figure 5: An implicit timer in Berkeley DB (after macro expansion and minor editing).

To discover implicit timers automatically, we developed a *static* symbolic analysis technique. It is based on the following two observations:

1. Programmers use time in *simple* ways. For example, they explicitly label time values (e.g., `db_timespec` in Figure 5), they do simple arithmetic on time values, and they generally do not cast time values to pointers and other unusual types. This observation implies that simple static analysis is sufficient to track how a time value flows.
2. Programmers check timeouts *soon* after they query the current time. The intuition is that programmers want the current time to be "fresh" when they check timeouts. This observation implies that our analysis only needs to track a short flow of a time value (e.g., within three function calls) and may stop when the flow becomes long.

We analyzed how time values are used in Berkeley DB version 4.7.25. We found that Berkeley DB mostly uses "+," "-," and occasionally "*" and "/" (for conversions, e.g., from seconds to milliseconds). In 12 out of 13 implicit timers, the time query and time check are within a few lines.

Our analysis resembles symbolic execution [3, 4, 13, 31]. It has three steps: (1) *statically* analyze the code of the target system and find all system calls that return time values; (2) track how the time values flow to variables; and (3) upon a branch statement involving a tracked time value, use a simple constraint solver to generate symbolic values to make both branches true. To show the idea, we use a source code instrumentation example. In Figure 5, our analysis can track how time flows from "`__os_gettime`" to "`if (now >= t)`," and replace the "`__os_gettime`" line in Figure 5 with

```
mc::rpc_client *rpc = mc::current_thread_rpc_client()
now = rpc->gettime(/*timer=*/t);
```

This RPC call tells the virtual clock manager that a timer fires at `t`; the virtual clock manager can then return a time value smaller than `t` for one execution, and greater than `t` for another execution, to explore both possible execution paths. We implemented our analysis using the Phoenix compiler framework [30].

Since our analysis is static, it avoids the runtime overhead of instrumenting each load and store for tracking symbolic values and thus is much simpler than dynamic symbolic execution tools [3, 4, 13, 31], which often take iterations to become stable [3, 4]. Note our analysis is unsound, as with other symbolic analysis tools, in that it may miss some timers and thus miss bugs. However, it will not introduce false positives because the virtual clock manager ensures the consistency of time.

Ensuring Consistent Clock. A consistent clock is crucial to avoid false positives. For example, the safety of the lease mechanism [14] requires that the lessee timeouts before the lessor; reversing the order may trigger "bugs" that never occur in practice. We actually encountered a painful false positive due to a violation of this safety requirement when checking PACIFICA.

To maintain consistent time, the virtual clock manager sorts all timers in the target system from earliest to last based on when these timers will fire. When the model checking engine decides to fire a timer, it will systematically choose one of several timers that fall in the range of $[T, T+E]$, where T is the earliest timer and E is a configurable clock error allowed by the target system. This mechanism lets MODIST explore interesting timer behaviors while not deviating too much from real timer-triggered executions.

3.5 Global Assertion

We have implemented global assertions leveraging our previous work D^3S [25]. D^3S enables transparent predicate checking of a running distributed system. It provides a simple programming interface for developers to specify global assertions, interposes both user-level functions and OS system calls in the target system to expose its runtime state as state tuples, and collects such tuples as globally consistent snapshots for evaluating assertions. To use D^3S, developers need to specify the functions being interposed, the state tuples being retrieved from function parameters, and a sequential program that takes a complete state snapshot as input to evaluate the predicate. D^3S compiles such assertions into a state exposing module, which is injected into all processes of the target system, and a checking module, which contains the evaluation programs and outputs checking results for every constructed snapshot.

MODIST incorporates D^3S to enable global assertions, with two noticeable modifications. First, we simplify D^3S by letting each node transmit state tuples synchronously to MODIST's checking process, which verifies assertions *immediately*. Previously, because nodes may transmit state tuples concurrently, D^3S must, before checking assertions, buffer each received tuple until all tuples causally dependent before that tuple have been received. Since MODIST runs one action at a time, it no longer needs to buffer tuples. Second, while D^3S uses a Lamport clock [23] to totally order state tuples into snapshots, MODIST uses a *vector clock* [26] to check more global snapshots.

3.6 State Space Exploration

MODIST maintains a queue of the state/action pairs to be explored. Due to the complexity of a distributed system, it is often infeasible for MODIST to exhaust the state space. Thus, it is key to decide which state/action pairs to add to the queue and the order in which they are explored.

MODIST tags each action with a vector clock and implements a customizable modular framework for exploring the state space so different reduction techniques and heuristics can be incorporated. This is largely inspired by our observation that the effectiveness of various strategies and heuristics is often application-dependent.

The basic state exploration process is simple: MODIST takes the first state/action pair $\langle s, a\rangle$ from the queue, steers the system execution to state s if that is not the current state, applies the action a, reaches a new state s', and examines the new resulting state for errors. It then calls a customizable function `explore`, which takes the entire *path* from the initial state to s and then s', where each state is tagged with its vector clock and each state transition is tagged with the action corresponding to the transition. For s', all enabled actions are provided to the function. The function then produces a list of state/action pairs and indicates whether the list should be added to the front of the queue or the back. MODIST then inserts the list into the queue and repeats the steps.

MODIST has a natural bias towards exploring $\langle s, a\rangle$ pairs where s is the state MODIST is in. This default strategy will save the cost of replaying the trace to reach the state in the selected state/action pair.

Now we show how various state exploration strategies and heuristics can be implemented in the MODIST framework.

Random. Random exploration with a bounded maximum path length explores a random path up to a bounded path length and then starts from the initial state for another random path. The `explore` function works as follows: if the current path has not exceeded the bound, the function will randomly pick an enabled action a' at the new state s' and has $\langle s', a'\rangle$ inserted to the end of the queue (note that the queue is empty). If the current path has reached the bound, the function will randomly choose an enabled action a_0 in the initial state s_0, and has $\langle s_0, a_0 \rangle$ inserted to the end of the queue.

DFS and BFS. For Depth First Search (DFS) and Breadth First Search (BFS), the `explore` function simply inserts $\langle s', a' \rangle$ for every enabled action a' in state s'. For DFS, the new list is inserted at the front of the queue, while for BFS at the back. Clearly, DFS is more attractive since MoDist does not have to replay traces often to recreate states.

DPOR. For dynamic partial order reduction (DPOR), the `explore` function works as follows. Let a be the last action causing the transition from s to s'. The function looks at every state s_p before s on the path and the action a_p taken at that state. If a is enabled at s_p (i.e., if s and s_p are concurrent judged by the vector clocks) and a does not commute with a_p (i.e., the different orders of the two actions could lead to different executions), we record $\langle s_p, a \rangle$ in the list of pairs to explore. Once all states are examined, the function returns the list and has MoDist insert the list in the queue.

By specifying how the list is inserted, the function could choose to use DFS or BFS on top of DPOR. Also, by ordering the pairs in the list differently, MoDist will be instructed to explore the newly added branches in different orders (e.g., top-down or bottom-up). The default is DFS again to avoid the cost of recreating states. We further introduce *Bounded DPOR* to refer to the variation of DPOR with bounds on DFS for a more balanced state-space exploration.

The `explore` function can be constructed to favor certain actions (e.g., crash events) over others, to bound the exploration in various ways (e.g., the path length and the number of certain actions on the path), and to focus on a subset of possible actions.

4 Evaluation

We have applied MoDist to three distributed systems: (1) Berkeley DB, a widely used open-source database (a version with replication); (2) MPS, a closed source Paxos [22] implementation built by a Microsoft product team and has been deployed in commercial data centers for more than two years; and (3) PacificA, a mature implementation of a primary-backup replication protocol we developed. We picked Berkeley DB and MPS because of their wide deployment and importance and PacificA because it provides an interesting case study where the developers apply model checking to their own systems.

Table 2 summarizes the errors we found, all of which are previously unknown bugs. We found a total of 35 errors, 10 of which are *protocol-level bugs* that occur only under rare interleavings of messages and crashes; these bugs reflect flaws in the underlying communication protocols of the systems. *Implementation bugs* are those that can be caused by injecting API failures. All MPS and PacificA bugs were confirmed by the developers. Three out of seven Berkeley DB bugs, including one protocol-level bug, were confirmed by Berkeley DB developers; we are having the rest confirmed. These unconfirmed bugs are likely real bugs because we can reproduce them without MoDist by manually tweaking the executions and killing processes according to the traces from MoDist.

System	KLOC	Protocol	Impl.	Total
Berkeley DB	172.1	2	5	7
MPS	53.5	2	11	13
PacificA	12	6	9	15
Total	237.6	10	25	35

Table 2: *Summary of errors found.* The **KLOC** (thousand lines of code) column shows the sizes of the systems we checked. We separate protocol-level bugs (**Protocol**) and implementation-level bugs (**Impl.**), in addition to reporting the total (**Total**). 31 of the 35 bugs have been confirmed by the developers.

While other tools (e.g., a static analyzer) can also find implementation bugs, MoDist has the advantage of not generating false positives. In addition, it can expose the effects of these bugs, helping prioritize fixing.

In the rest of this section, we describe our error detection methodology, the bugs we found, MoDist's coverage results and runtime overhead, and the lessons we have learned.

4.1 Experimental Methodology

Test driver. Model checking is most effective at checking complicated interactions between a small number of objects. Thus, in all tests we run, we use several processes servicing a bounded number of requests. Since the systems we check came with test cases, we simply use them with minor modifications.

Global assertions. By default, MoDist checks fail-stop errors. To check the correctness properties of a distributed system, MoDist supports user supplied global assertions (§2). For the replication systems we checked, we added two types of assertions. The first type was global predicates for the safety properties. For example, all replicas agree on the same sequence of commands. The second type of predicates check for liveness. True liveness conditions cannot be checked by execution monitoring so we instead approximate them by checking for progress in the system: we expect the target system to make progress in the absence of failures. In the end, we did not find any bug that violated the safety properties in any of the systems, probably reflecting the relative maturity of these systems. However, we did find bugs that violated liveness global assertions in every system.

Search strategy. MODIST has a set of built-in search strategies; no single strategy works the best. We have combined these strategies in our experiments for discovering bugs effectively. For example, we can first perform random executions (Random) on the system and inject the API failures randomly to get the shallow implementation bugs. We can then use the DPOR strategy with randomly chosen initial paths to explore message orders systematically. We can further add crash and recovery events on top of the message interleaving, starting from a single crash and gradually increasing the number of crashes, to exercise the system's handling of crash and recovery. We can run these experiments concurrently and fine-tune the strategies.

Terminology. Distributed systems use different terminologies to describe the roles the nodes play in the systems. In this paper, we will use *primary* and *secondary* to distinguish the replicas in the systems. They are called *master* and *client* respectively in Berkeley DB documents. In the Paxos literature, a primary is also called a *leader*.

4.2 Berkeley DB: a Replicated Database

Berkeley DB is a widely used open source transactional storage engine. Its latest version supports replication for applications that must be highly available. In a Berkeley DB replication group, the primary supports both reads and writes while secondaries support reads only. New replicas can join the replication group at any time.

We checked the latest Berkeley DB production release: 4.7.25.NC. We use `ex_rep_mgr`, an example application that comes with Berkeley DB as the test driver. This application manages its data using the Berkeley DB Replication Manager. Our test setup has 3 to 5 processes. They first run an election. Once the election completes, the elected primary inserts data into the replicated database, reads it back, and verifies that it matches the data inserted.

Results and Discussions. We found seven bugs in Berkeley DB: four were triggered by injecting API failures, one was a dangling pointer error triggered by the primary waiting for multiple ACK messages simultaneously from the secondaries, and the remaining two were protocol-level bugs, which we describe below.

The first protocol-level bug causes a replica to crash due to an "unexpected" message. The timing diagram of this bug is depicted in Figure 6. Replica *C* is the original primary. Suppose a new election is launched, resulting in replica *A* becoming the new primary. Replica *A* will broadcast a `REP_NEWMASTER` message, which means "I am the new primary." After replica *B* receives this message, it tries to synchronize with the new primary and sends *A* a `REP_UPDATA_REQ` message to get the up-to-date data. Meanwhile, *C*

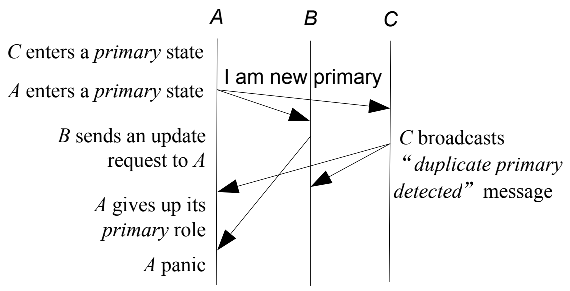

A
B
C
C enters a primary state
I am new primary
A enters a primary state
B sends an update request to A
C broadcasts "duplicate primary detected" message
A gives up its primary role
A panic

Figure 6: Timing Diagram of Message Exchanges in a Berkeley DB Replication Bug.

processes `REP_NEWMASTER` by first broadcasting a `REP_DUPMASTER` message, which means "duplicate primary detected," and then degrading itself to a secondary. Broadcasting a `REP_DUPMASTER` message is necessary to ensure that all other replicas know that *C* is not primary anymore. When *A* processes `REP_DUPMASTER`, it has to give up its primary role because it cannot make sure that it is the latest primary. Soon *A* receives the delayed but not-outdated `REP_UPDATA_REQ` message from *B*. Replica *A* panics at once, because such message should only be received by primary. Such panics occur whenever a delayed `REP_UPDATA_REQ` message arrives at a recently degraded primary.

The second protocol level bug is more severe: it causes permanent failures in leader election due to a primary crash when all secondaries believe they cannot be primaries. Suppose replica *A* is the original primary and is synchronizing data with secondaries *B* and *C*. Normally synchronization works as follows. *A* sends a `REP_PAGE` message with the modified database page to *B* and *C*. Upon receipt of this message, *B* and *C* transit to log recovery state by setting the `REP_F_RECOVER_LOG` flag. *A* then sends a `REP_LOG` message with the updated log records. However, if *A* crashes before it sends `REP_LOG`, *B* and *C* will never be able to elect a new primary because, in Berkeley DB's replication protocol, a replica in log recovery is not allowed to be a primary.

4.3 MPS: Replicated State Machine Library

MPS is a practical implementation of a replicated state machine library. The library has been used for over two years in production clusters of more than 100K machines for maintaining important system metadata consistently and reliably. It consists of 8.5K lines of C++ code for the communication protocol, and 45K for utilities such as networking and storage.

At the core of MPS is a distributed Paxos protocol for consensus [22]. The protocol is executed on a set of machines called *replicas*. The goal of the protocol is to have replicas agree on a sequence of deterministic commands

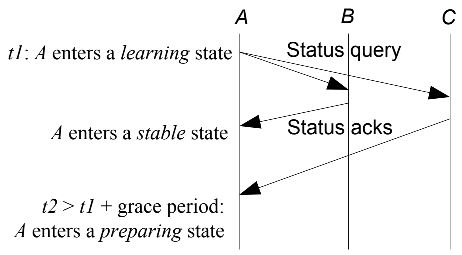


Figure 7: The Timing Diagram of Message Exchange in MPS Bug 1.

and execute the commands in the same sequence order. Because all replicas start with the same initial state and execute the same sequence of commands, consistency among replicas is guaranteed.

The MPS consensus protocol is leader (primary) based. While the protocol ensures safety despite the existence of multiple primaries, a single primary is needed for the protocol to make progress. A replica can act as a primary using a certain *ballot number*. A primary accepts requests from clients and proposes those requests as *decrees*, where decree numbers indicate the positions of the requests in the sequence of commands that is going to be executed by the replicated state machine. A decree is considered *committed* when the primary gets acknowledgment from a quorum (often a majority) of replicas indicating that they have accepted and persistently stored the decree.

If a replica receives a message that indicates that a decree unknown to the replica is committed, then the replica enters a *learning* phase, in which it learns the missing decrees from other replicas.

When an existing primary is considered to have failed, a new primary can be elected. The new primary will use a higher ballot number and carry out a *prepare* phase to learn the decrees that could have been committed and ensure no conflicting decrees are proposed. For each replica, a proposal with a higher ballot number overwrites any previous proposal with lower ballot numbers.

Our test setup consists of 3 replicas, proposing a small number of decrees.

Results and Discussions. We found 13 bugs in MPS, 11 are implementation bugs that crash replicas, and the other two bugs are protocol-level bugs.

The first protocol-level bug reveals a scenario that leads to state transitions that are not expected by the developers (as demonstrated by the assertion that rules out the transition). MPS has a simple set of states and state transitions. A replica is normally in a *stable* state. When it gets indication that its state is falling behind (i.e., missing decrees), it enters a *learning* state. In the learning state, it fetches the decrees from a quorum of replicas. Once it brings its state up to date with what it receives from a quorum of replicas, it checks whether it should become a primary: if the primary lease expires, then the replica will compete to be a primary by entering a *preparing* state; otherwise, it will return to a stable state. There is an assertion in the code (and also in the design document for MPS) that the state transition from stable to preparing is impossible.

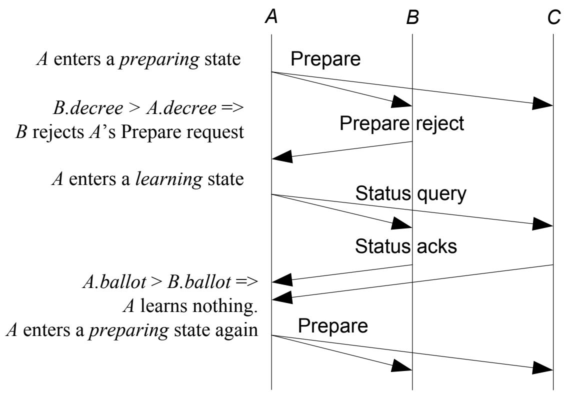


Figure 8: The Timing Diagram of Message Exchange in MPS Bug 2.

Figure 7 shows the MoDist-generated scenario that triggers the assertion failure. The following is a list of steps that lead to the violation. Consider the case where the system consists of three replicas *A*, *B*, and *C*, where any two of them form a quorum. Replica *A* enters the learning state because it realizes that it does not have the information related to some decree numbers. This could be due to the receipt of a message that indicates that the last committed decree number is at least *k*, while *A* knows only up to some decree number less than *k*. *A* then sends a status query to *B* and *C*. *A* receives the response from *B* and learns all the missing decrees. Since *A* and *B* form a quorum, *A* enters the stable state. *C* was the primary. *C*'s response to *A* status query was delayed, and the primary lease becomes expired on *A*. At some later point, *C*'s response arrives. The implementation will handle that message as if *A* were in the learning state. After *A* is done, it notices that the primary lease has expired and transitions into the preparing state, causing the unexpected state transition. As a result, *A* crashes and reboots.

The second protocol-level bug is a violation of a global liveness assertion. It is triggered during primary election under the following scenario: replica *A* has accepted a decree with ballot number 2 and decree number 1, while replica *B* only has ballot number 1, but accepted a decree of decree number 2.

The following series of events lead to this problematic scenario: *B* is a primary with ballot number 1, it proposes a decree with decree number 1 and the decree is accepted by all replicas including *A* and *B*. It then proposes another decree with decree number 2, which is accepted only on *B*. *B* fails before *A* gets the proposal. *A* then becomes a primary with ballot number 2, learns the decree with decree number 1, re-proposes it with a ballot number 2.

Figure 8 shows the timing diagram continuing from this scenario. *B* comes back, receives the prepare request from *A*, and sends a rejection to *A* because *B* thinks *A* is not up-to-date given that *B* has a higher decree number. After getting the rejection, *A* enters a learning state. In the learning state, even if *B* returns the decree with decree number 2, *A* will reject it because it has a lower ballot number. *A* will consider itself up-to-date and enter the preparing state again with a yet higher ballot number. This continues as *A* keeps increasing its ballot number, but unable to have new decrees committed, triggering a liveness violation.

The problem in this scenario is due to the inconsistency of the views on what constitutes a newer state between the *preparing phase* and the *learning phase*: one view uses a higher ballot number, while the other uses a higher decree number. The inconsistency is exposed when one has a higher decree number, but a lower ballot number than the other.

4.4 PACIFICA: a Primary-Backup Replication Protocol

PACIFICA [24] is a large-scale storage system for semi-structured data. It implements a Primary-Backup protocol for data replication. We used MODIST to check an implementation of PACIFICA's replication protocol. This implementation consists of 5K lines of C++ code for the communication protocol and 7K for utilities.

PACIFICA uses a variety of familiar components including two-phase commit for consistent replica updates, perfect failure detection, replica group reconfiguration to handle node failures, and replica reconciliation for nodes rejoining a replica group.

Our test setup for PACIFICA has 4 processes: 1 master that maintains global metadata, 2 replica nodes that implement the replication protocol, and 1 client that updates the system and drives the checking process. Figure 2 shows the configuration file.

Results and Discussions. We found 15 bugs in PACIFICA: 9 are implementation bugs that cause crashes and 6 are protocol-level bugs. We managed to find more protocol-level bugs in PACIFICA than in other systems for two reasons: (1) since we built the system, we could quickly fix the bugs MODIST found then re-run MODIST to go after other bugs; and (2) we could check more global assertions for PACIFICA.

The most interesting bug we found in PACIFICA prevents PACIFICA from making progress. It is triggered by a node crash followed by a replication group reconfiguration. A primary replica keeps a list of prepared updates (i.e., updates that have been prepared on all replicas, but not yet committed); a secondary replica does not have this data structure. When a primary crashes, a secondary will try to take over and become the new primary. If the crash happens in the middle of a commit operation that leaves some commands prepared but not yet committed, the new primary will try to re-commit all prepared updates by sending the "prepare" messages to the remaining secondary replicas. Unfortunately, PACIFICA did not put these newly prepared updates into the prepared update list. This prevents all the following updates from getting committed because of a hole in the prepared update list.

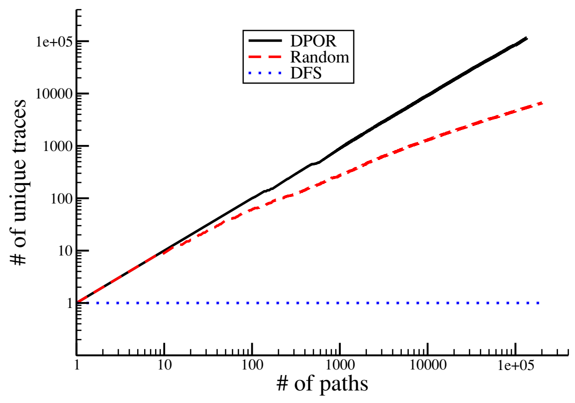


Figure 9: Partial order state coverage of different exploration strategies.

4.5 State Coverage

To evaluate the state-space exploration strategies described in §3.6, we measured *state coverage*: the number of unique states a strategy could explore after running a fixed number of execution paths. We examined the coverage of two types of states:

1. *Partial order traces [12].* Since two paths with the same partial order are equivalent, the number of different partial order traces provides an upper bound on the number of unique behaviors a strategy can explore.
2. *Protocol states*. These states capture the more important protocol behaviors of a distributed system.

We did two experiments, both on MPS: one with a small partial order state space and the other with a nearly unbounded state space. These two state spaces give an idea of how sensitive the strategies are to state-space sizes. No crash was injected during the evaluation.

In the first experiment, we made the state space small using a configuration of two nodes, each receiving up to two messages. Figure 9 shows the number of unique partial order traces with respect to the number of paths explored. (Note that both axes are in log scale.) DPOR

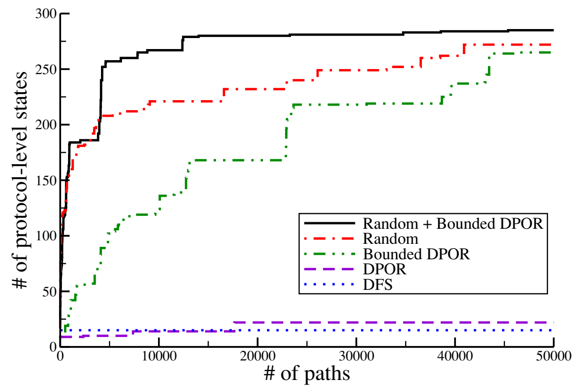


Figure 10: Protocol state coverage of different exploration strategies.

shows a clear advantage: it exhausted all 115,425 traces after 134,627 paths (the small redundancy was due to an approximation in our DPOR implementation.) The Random strategy explored 6,614 unique traces or 5.7% of the entire state space after 200,000 paths. DFS is the worst: all the 200,000 paths were partial order equivalent and corresponded to only one partial order trace.

In the second experiment, we used a nearly unbounded partial order state space with three MPS nodes sending and receiving an unbounded number of messages. We bounded the maximum decree (two decrees) and the maximum path length (40,000 actions) to make the execution paths finite. Since the state space was large, it was unlikely that Random ever explored a partial order trace twice. As a result, DPOR behaved the same as Random. (This result is not shown.)

While partial order state coverage provides an upper bound on the unique behaviors a strategy explores, different partial order traces may still be redundant and map to the same protocol state. Thus, we further measured the protocol state coverage of different exploration strategies. We defined the *protocol state* of MPS as a tuple $\langle state, ballot, decree \rangle$, where the *state* could be `initializing`, `learning`, `stable primary`, or `stable secondary`.[*]

Figure 10 shows the protocol states covered by the first 50,000 paths explored in each strategy, using the MPS configuration from the second experiment. DFS had the worst coverage: it found no new states after exploring the first path. The reason is, when the state space is large, DFS tends to explore a large number of paths that differ only at the final few steps; these paths are often partial-order equivalent. DPOR performed almost equally badly: it found less than 30 protocol states. This result is not surprising for two reasons: (1) different partial order traces might correspond to the same protocol state and (2) DPOR is DFS-based, thus suffers the same problem as DFS when the state space is large.

[*]We also measured the coverage of *global* protocol states, which consist of protocol states of each node in a consistent global snapshot. The results were similar and not shown.

In Bounded DPOR, protocol-level redundancy is partially conquered by the bounds on backtracks. As shown in Figure 10, the protocol-level state coverage of Bounded DPOR was larger than that of DPOR by an order of magnitude, in the first 50,000 paths.

Surprisingly, the Random strategy yielded better coverage than DFS, DPOR, and even Bounded DPOR. The reason is that Random is more balanced: it explores actions anywhere along a path uniformly, therefore it has a better chance to jump to a new path early on and explores a different area of the state space.

These results prompted us to develop a hybrid *Random + Bounded DPOR* search strategy that works as follows. It starts with a random path and explores the state space with Bounded DPOR. We further bound the total number of backtracks so that the Bounded DPOR exploration ends. Then, a new round of Bounded DPOR exploration starts with a new random path. *Random + Bounded DPOR* inherits both the balance of Random and the thoroughness of DPOR to cover the corner cases. Both the round number of DPOR explorations and the bound of the total number of backtracks are customizable, reflecting a bias towards Random or towards DPOR. As shown in Figure 10, the *Random + Bounded DPOR* strategy with a round number 100 performed the best.

4.6 Performance

In our performance measurements, we focused on three metrics: (1) MODIST's path exploration speed; (2) the speedup due to the virtual clock fast-forward; and (3) the runtime overhead MODIST adds to the target system, including interposition, RPC, and backend scheduling.

We set up our experiments as follows. We ran MODIST with two different search strategies: RANDOM and DPOR. For each search strategy, we let MODIST explore 1K execution paths and recorded the running times. We repeated this experiment 50 times and took the average. We used Berkeley DB and MPS as our benchmarks, using identical configurations as those used for error detection. We ran our experiments on a 64-bit Windows Server 2003 machine with dual Intel Xeon 5130 CPU and 4GB memory. We measured all time values using `QueryPerformanceCounter()`, a high-resolution performance counter.

It appears that we should measure MODIST's overhead by comparing a system's executions with MODIST to those without. However, due to nondeterminism, we cannot compare these two directly: the executions without MODIST may run different program paths than those with MODIST. Moreover, repeated executions of the same testcase without MODIST may differ; we did ob-

System	Strategy	Real (s)	Sleep (s)	Speedup	Overhead (absolute and relative)
Berkeley DB	RANDOM	$1,717 \pm 14$	$38,204 \pm 193$	25.7 ± 0.2	$302 \pm 1s$ $(17.7 \pm 0.1\%)$
Berkeley DB	DPOR	$1,658 \pm 24$	$36,402 \pm 5,137$	22.1 ± 3.2	$301 \pm 17s$ $(18.2 \pm 0.9\%)$
MPS	RANDOM	$1,661 \pm 20$	$240,568 \pm 1,405$	216 ± 2	$825 \pm 11s$ $(49.9 \pm 0.2\%)$
MPS	DPOR	$1,853 \pm 116$	$295,435 \pm 45,659$	159 ± 19	$1,048 \pm 108s$ $(56.5 \pm 2.6\%)$

Table 3: MODIST's performance. All numbers are of the form $average \pm standard\,deviation$.

serve a large variance in MPS's execution times and final protocol states. Thus, we evaluated MODIST's overhead by running a system with MODIST and measuring the time spent in MODIST's components.

Table 3 shows the performance results. The **Real** column shows the time it took for MODIST to explore 1K paths of Berkeley DB and MPS with RANDOM and DPOR strategies; the exploration speed is roughly two seconds per path and does not change much for the two different search strategies. The **Sleep** column shows the time MODIST saved using its virtual clock when the target systems were asleep; we would have spent this amount of extra time had we run the same executions without MODIST. As shown in the table, the real execution time is much smaller that the sleep time, translated into significant speedups (Column **Speedup**, computed as **Sleep/Real**). The **Overhead** column in this table shows the time spent in MODIST's interposition, RPC, and backend scheduling. For Berkeley DB, MODIST accounts for about 18% of the real execution time. For MPS, MODIST accounts for a higher percentage of execution time (up to 56.5%) because the MPS testcase we used is almost the worst case for MODIST: it only exercises the underlying communication protocol and does no real message processing. Nonetheless, we believe such overhead is reasonable for an error detection tool.

4.7 Lessons

This section discusses the lessons we learned.

Real distributed protocols are buggy. We found many protocol-level bugs and we found them in every system we target, suggesting that real distributed protocols are buggy. Amusingly, these protocols are based on theoretically sound protocols; the bugs are introduced when developers filled in the unspecified parts in the protocols in practice.

Controlling all non-determinism is hard. Systematic checking requires control of non-determinism in the target system. This task is very hard given the non-determinism in the OS and network, the wide API interface, the many possible failures and their combinations, and MODIST's goal of reducing intrusiveness to the target system. We have had bitter experiences debugging non-deterministic errors in Berkeley DB, which uses process id, memory address, and time to generate random numbers, and in MPS, which randomly interferes with the default Windows firewall. Among all, making the Windows socket APIs deterministic was the most difficult; the interface shown in §3 went through several iterations. Our own experiences show that controlling all non-determinism is much harder than merely capturing it as in replay-debugging tools.

Avoid false positives at all cost. False positives may take several days to diagnose. Thus, we want to avoid them, even at the risk of *missing errors.*

Leverage domain knowledge. In a sense, this entire paper boils down to leveraging the domain knowledge of distributed systems to better model-check them. The core idea of model checking is simple: explore all possible executions; a much more difficult task is to implement this idea effectively in an application domain.

When in doubt, reboot. When we checked MPS, we were surprised by how robust it was. MPS uses a defensive programming technique that works particularly well in the context of distributed replication protocols. MPS extensively uses local assertions, reboots when any assertion fails, and relies on the replication protocol to recover from these eager reboots. This recovery mechanism makes MPS robust against a wide range of failures. Of course, rebooting is not without penalty: if a primary reboots, there could be noticeable performance degradation, and the system also becomes less fault tolerant.

5 Related Work

5.1 Model Checking

Model checkers have previously been used to find errors in both the design and the implementation of software [1, 6, 12, 17–19, 27, 28, 34, 38, 39]. Traditional model checkers require users to write an abstract model of the target system, which often incurs large up-front cost when checking large systems. In contrast, MODIST is an implementation-level model checker that checks code directly, thus avoids this cost. Below we compare MODIST to implementation-level model checkers.

Model checkers for distributed system. MODIST is most related to model checkers that check real distributed system implementations. CMC [27] is a stateful model checker that checks C code directly. It has been used to check network protocol implementations [27] and file systems [38]. However, to check a system, CMC requires invasive modifications to run the system inside CMC's address space [39]. MaceMC [19] uses bounded depth first search combined with random walk to find safety and liveness bugs in a number of network protocol implementations written in a domain-specific language. Compared to these two checkers, MODIST directly checks live, unmodified distributed systems running in their native execution environments, thus avoids the invasive modifications required by CMC, and the language restrictions [20] enforced by MaceMC.

CrystalBall [37] detects and avoids errors in deployed distributed systems using an efficient global state collection and exploration technique. While CrystalBall is based on MaceMC and thus checks only systems written in the Mace language [20], its core technique may be portable to MODIST's model checking framework to improve the reliability of general distributed systems.

Other software model checkers. We compare MODIST to other closely related implementation-level model checkers. Our transparent checking approach is motivated by our previous work EXPLODE [39]. However, EXPLODE focuses on storage systems and does not check distributed systems.

To our best knowledge, VeriSoft [12] is the first implementation-level model checker. It systematically explores the interleavings of concurrent C programs, and uses partial order reduction to soundly reduce the number of states it explores. It has been used to check industrial-strength programs [5].

Chess [28] is a stateless model checker for exploring the interleavings of multi-threaded programs. To avoid perturbing the target system, it also interposes on WinAPIs. In addition, Chess uses a *context-bounding* heuristic and a *starvation-free* scheduler to make its checking more efficient. It has been applied to several industry-scale systems and found many bugs.

ISP [35] is an implementation-level model checker for MPI programs. It controls a MPI program by intercepting calls to MPI methods and reduces the state-space it explores using new partial order reduction algorithms.

All three systems focus on checking interleavings of concurrent programs, thus do not address issues on checking real distributed systems, such as providing a transparent, distributed checking architecture and enabling consistent and deterministic failure simulation

5.2 Replay-based debugging

A number of systems [11, 21, 32], including our previous work [15, 25], use deterministic replay to debug distributed system. These approaches attack a different problem: when a bug occurs, how to capture its manifestation so that developers can reproduce the bug. Combined with fault injection, these tools can be used to detect bugs. Like these systems, MODIST also provides reproducibility of errors. Unlike these systems, MODIST aims to *proactively* drive the target system into corner-cases for errors in the testing phase before the system is deployed. MODIST uses the instrumentation library in our previous work [25] to interpose on WinAPIs.

5.3 Other error detection techniques

We view testing as complementary to our approach. Testing is usually less comprehensive than our approach, but works "out of the box." Thus, there is no reason not to use both testing and MODIST together.

There has been much recent work on static bug finding (e.g., [1, 2, 7, 8, 10, 33]). Roughly speaking, because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by the code (e.g., two replicas are consistent). The protocol-level errors we found would be difficult to find statically. We view static analysis as complementary: easy enough to apply such that there is no reason not to use them together with MODIST.

Recently, symbolic execution [3, 4, 13, 31] has been used to detect errors in real systems. This technique is good at detecting bugs caused by tricky input values, whereas our approach is good at detecting bugs caused by the non-deterministic events in the environment.

6 Conclusions

MODIST represents an important step in achieving the ideal of model checking unmodified distributed system in a transparent and effective way. Its effectiveness has been demonstrated by the subtle bugs it uncovered in well-tested production and deployed systems.

Our experience shows that it requires a combination of art, science, and engineering. It is an art because various heuristics must be developed for finding delicate bugs effectively, taking into account the peculiarity of complex distributed systems; it is a science because a systematic, modular approach with a carefully designed architecture is a key enabler; it involves heavy engineering effort to interpose between the application and the OS, to model and control low-level system behavior, and to handle system-level non-determinism.

Acknowledgement

We thank Huayang Guo and Liying Tang for their help in evaluating MODIST, Jian Tang for his contributions to an earlier version of the system, and our colleagues at Microsoft Research Silicon Valley and the System Research Group at Microsoft Research Asia for their comments and support. We especially thank Stephen A. Edwards for extensive edits and Stelios Sidiroglou-Douskos for detailed comments. We are also grateful to the anonymous reviewers for their valuable feedback and to our shepherd Steven Hand for his guidance.

References

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the Eighth International SPIN Workshop on Model Checking of Software (SPIN '01)*, pages 103–122, May 2001.

[2] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[3] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the 13th ACM conference on Computer and communications security (CCS '06)*, pages 322–335, Oct.–Nov. 2006.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 209–224, Dec. 2008.

[5] S. Chandra, P. Godefroid, and C. Palm. Software model checking in practice: an industrial case study. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 431–441, May 2002.

[6] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 439–448, June 2000.

[7] M. Das, S. Lerner, and M. Seigle. Esp: path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 57–68, June 2002.

[8] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI '00)*, Sept. 2000.

[9] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd Annual Symposium on Principles of Programming Languages (POPL '05)*, pages 110–121, Jan. 2005.

[10] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI '02)*, pages 234–245, June 2002.

[11] D. Geels, G. Altekarz, P. Maniatis, T. Roscoey, and I. Stoicaz. Friday: Global comprehension for distributed replay. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, Apr. 2007.

[12] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th Annual Symposium on Principles of Programming Languages (POPL '97)*, pages 174–186, Jan. 1997.

[13] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI '05)*, pages 213–223, June 2005.

[14] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 202–210, Dec. 1989.

[15] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang. R2: An application-level kernel for record and replay. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 193–208, Dec. 2008.

[16] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '92)*, pages 125–138, Dec. 1992.

[17] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5): 279–295, 1997.

[18] G. J. Holzmann. From code to models. In *Proceedings of the Second International Conference on Applications of Concurrency to System Design (ACSD '01)*, June 2001.

[19] C. Killian, J. W. Anderson, R. Jhala, , and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the Fourth Symposium on Networked Systems Design and Implementation (NSDI '07)*, pages 243–256, April 2007.

[20] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 179–188, June 2007.

[21] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed Java applications. In *Proceedings of the 14th International Symposium on Parallel and Distributed Processing (IPDPS '00)*, pages 219–228, May 2000.

[22] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[23] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Comm. ACM*, 21(7):558–565, 1978.

[24] W. Lin, M. Yang, L. Zhang, and L. Zhou. Pacifica: Replication in log-based distributed storage systems. Technical report.

[25] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. In *Proceedings of the Fifth Symposium on Networked Systems Design and Implementation (NSDI '08)*, pages 423–437, Apr. 2008.

[26] F. Mattern. Virtual time and global states of distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226. 1988.

[27] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI '02)*, pages 75–88, Dec. 2002.

[28] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI '08)*, pages 267–280, Dec. 2008.

[29] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI '07)*, pages 89–100, June 2007.

[30] Phoenix. `http://research.microsoft.com/Phoenix/`.

[31] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05)*, pages 263–272, Sept. 2005.

[32] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the USENIX Annual Technical Conference (USENIX '04)*, pages 29–44, June 2004.

[33] The Coverity Software Analysis Toolset. `http://coverity.com`.

[34] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[35] A. Vo, S. Vakkalanka, M. DeLisi, G. Gopalakrishnan, R. M. Kirby, , and R. Thakur. Formal verification of practical mpi programs. In *Principles and Practices of Parallel Programming (PPoPP)*, pages 261–260, Feb. 2009.

[36] Windows API. `http://msdn.microsoft.com/`.

[37] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. CrystalBall: Predicting and preventing inconsistencies in deployed distributed systems. In *Proceedings of the Sixth Symposium on Networked Systems Design and Implementation (NSDI '09)*, Apr. 2009.

[38] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04)*, pages 273–288, Dec. 2004.

[39] J. Yang, C. Sar, and D. Engler. Explode: a lightweight, general system for finding serious storage system errors. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation (OSDI '06)*, pages 131–146, Nov. 2006.

CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems

Maysam Yabandeh, Nikola Knežević, Dejan Kostić and Viktor Kuncak
School of Computer and Communication Sciences, EPFL, Switzerland
email: `firstname.lastname@epfl.ch`

Abstract

We propose a new approach for developing and deploying distributed systems, in which nodes predict distributed consequences of their actions, and use this information to detect and avoid errors. Each node continuously runs a state exploration algorithm on a recent consistent snapshot of its neighborhood and predicts possible future violations of specified safety properties. We describe a new state exploration algorithm, consequence prediction, which explores causally related chains of events that lead to property violation.

This paper describes the design and implementation of this approach, termed CrystalBall. We evaluate CrystalBall on RandTree, BulletPrime, Paxos, and Chord distributed system implementations. We identified new bugs in mature Mace implementations of three systems. Furthermore, we show that if the bug is not corrected during system development, CrystalBall is effective in steering the execution away from inconsistent states at runtime.

1 Introduction

Complex distributed protocols and algorithms are used in enterprise storage systems, distributed databases, large-scale planetary systems, and sensor networks. Errors in these protocols translate to denial of service to some clients, potential loss of data, and monetary losses. The Internet itself is a large-scale distributed system, and there are recent proposals [19] to improve its routing reliability by further treating routing as a distributed consensus problem [26]. Design and implementation problems in these protocols have the potential to deny vital network connectivity to a large fraction of users.

Unfortunately, it is notoriously difficult to develop reliable high-performance distributed systems that run over asynchronous networks. Even if a distributed system is based on a well-understood distributed algorithm, its implementation can contain errors arising from complexities of realistic distributed environments or simply coding errors [27]. Many of these errors can only manifest after the system has been running for a long time, has developed a complex topology, and has experienced a particular sequence of low-probability events such as node resets. Consequently, it is difficult to detect such errors using testing and model checking, and many of such errors remain unfixed after the system is deployed.

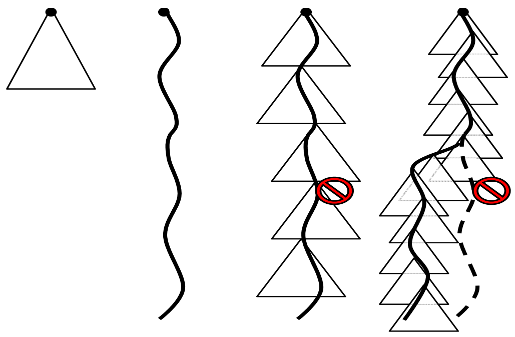


Figure 1: Execution path coverage by a) classic model checking, b) replay-based or live predicate checking, c) CrystalBall in deep online debugging mode, and d) CrystalBall in execution steering mode. A triangle represents the state space searched by the model checker; a full line denotes an execution path of the system; a dashed line denotes an avoided execution path that would lead to an inconsistency.

We propose to leverage increases in computing power and bandwidth to make it easier to find errors in distributed systems, and to increase the resilience of the deployed systems with respect to any remaining errors. In our approach, distributed system nodes predict consequences of their actions while the system is running. Each node runs a state exploration algorithm on a consistent snapshot of its neighborhood and predicts which actions can lead to violations of user-specified consistency properties. As Figure 1 illustrates, the ability to detect future inconsistencies allows us to address the problem of reliability in distributed systems on two fronts: debugging and resilience.

- Our technique enables deep online debugging because it explores more states than live runs alone or model checking from the initial state. For each state that a running system experiences, our technique checks many additional states that the system did not go through, but that it could reach in similar executions. This approach combines benefits of distributed debugging and model checking.
- Our technique aids resilience because a node can modify its behavior to avoid a predicted inconsistency. We call this approach *execution steering*. Execution steering enables nodes to resolve nondeterminism in ways that aim to minimize future inconsistencies.

To make this approach feasible, we need a fast state exploration algorithm. We describe a new algorithm, termed *consequence prediction*, which is efficient enough to detect future violations of safety properties in a running system. Using this approach we identified bugs in Mace implementations of a random overlay tree, and the Chord distributed hash table. These implementations were previously tested as well as model-checked by exhaustive state exploration starting from the initial system state. Our approach therefore enables the developer to uncover and correct bugs that were not detected using previous techniques. Moreover, we show that, if a bug is not detected during system development, our approach is effective in steering the execution away from erroneous states, without significantly degrading the performance of the distributed service.

1.1 Contributions

We summarize the contributions of this paper as follows:

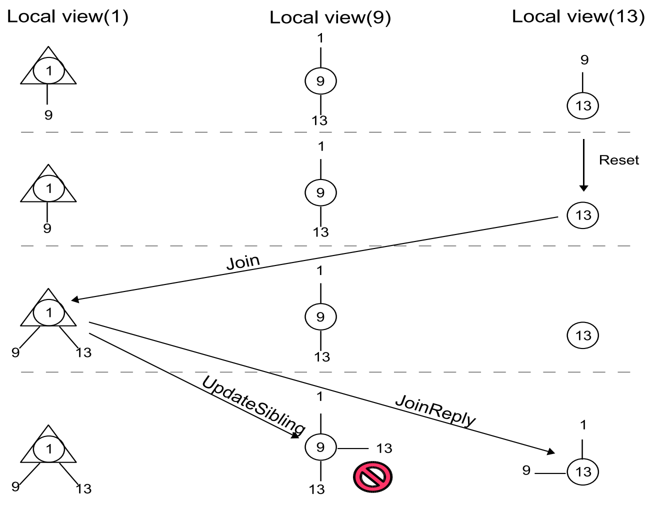
- We introduce the concept of continuously executing a state space exploration algorithm in parallel with a deployed distributed system, and introduce an algorithm that produces useful results even under tight time constraints arising from runtime deployment;
- We describe a mechanism for feeding a consistent snapshot of the neighborhood of a node in a large-scale distributed system into a running model checker; the mechanism enables reliable consequence prediction within limited time and bandwidth constraints;
- We present execution steering, a technique that enables the system to steer execution away from possible inconsistencies;
- We describe CrystalBall, the implementation of our approach on top of the Mace framework [21]. We evaluate CrystalBall on RandTree, Bullet′, Paxos, and Chord distributed system implementations. CrystalBall detected several previously unknown bugs that can cause system nodes to reach inconsistent states. Moreover, if the developer is not in a position to fix these bugs, CrystalBall's execution steering predicts them in a deployed system and steers execution away from them, all with an acceptable impact on the overall system performance.

1.2 Example

We next describe an example of an inconsistency exhibited by a distributed system, then show how CrystalBall predicts and avoids it. The inconsistency appears in the Mace [21] implementation of the RandTree overlay. RandTree implements a random, degree-constrained overlay tree designed to be resilient to node failures and network partitions. Trees built by an earlier version of this protocol serve as a control tree for a number of large-scale distributed services such as Bullet [23] and RanSub [24]. In general, trees are used in a variety of multicast scenarios [3, 7] and data collection/monitoring environments [17]. Inconsistencies in these environments translate to denial of service to users, data loss, inconsistent measurements, and suboptimal control decisions. The RandTree implementation was previously manually debugged both in local- and wide-area settings over a period of three years, as well as debugged using an existing model checking approach [22], but, to our knowledge, this inconsistency has not been discovered before (see Section 4 for some of the additional bugs that CrystalBall discovered).

RandTree Topology. Nodes in a RandTree overlay form a directed tree of bounded degree. Each node maintains a list of its children and the address of the root. The node with the numerically smallest IP address acts as the root of the tree. Each non-root node contains the address of its parent. Children of the root maintain a sibling list. Note that, for a given node, its parent, children, and siblings are all distinct nodes. The seemingly simple task of maintaining a consistent tree topology is complicated by the requirement for groups of nodes to agree on their roles (root, parent, child, sibling) across asynchronous networks, in the face of node failures, and machine slowdowns.

Joining the Overlay. A node n_j joins the overlay by issuing a Join request to one of the designated nodes. If the node receiving the join request is not the root, it forwards the request to the root. If the root already has the maximal number of children, it asks one of its children to incorporate the node into the overlay. Once the



Safety property: children and siblings are disjoint lists

Figure 2: An inconsistency in a run of RandTree

request reaches a node n_p whose number of children is less than maximum allowed, node n_p inserts n_j as one of its children, and notifies n_j about a successful join using a JoinReply message (if n_p is the root, it also notifies its other children about their new sibling n_j using an UpdateSibling message).

Example System State. The first row of Figure 2 shows a state of the system that we encountered by running RandTree in the ModelNet cluster [43] starting from the initial state. We examine the local states of nodes n_1, n_9, and n_{13}. For each node n we display its neighborhood view as a small graph whose central node is n itself, marked with a circle. If a node is root and in a "joined" state, we mark it with a triangle in its own view.

The state in the first row of Figure 2 is formed by n_{13} joining as the only child of n_9 and then n_1 joining and assuming the role of the new root with n_9 as its only child (n_{13} remains as the only child of n_9). Although the final state shown in first row of Figure 2 is simple, it takes 13 steps of the distributed system (such as atomic handler executions, including application events) to reach this state from the initial state.

Scenario Exhibiting Inconsistency. Figure 2 describes a sequence of actions that leads to a state that violates the consistency of the tree. We use arrows to represent the sending and the receiving of some of the relevant messages. A dashed line separates distinct distributed system states (for simplicity we skip certain intermediate states and omit some messages).

The sequence begins by a silent reset of node n_{13} (such reset can be caused by, for example, a power failure). After the reset, n_{13} attempts to join the overlay again. The root n_1 accepts the join request and adds n_{13} as its child. Up to this point node n_9 received no information on actions that followed the reset of n_{13}, so n_9 maintains n_{13} as its own child. When n_1 accepts n_{13} as a child, it sends an UpdateSibling message to n_9. At this point, n_9 simply inserts n_{13} into the set of its sibling. As a result, n_{13} *appears both in the list of children and in the list of siblings of* n_9, which is inconsistent with the notion of a tree.

Challenges in Finding Inconsistencies. We would clearly like to avoid inconsistencies such as the one appearing in Figure 2. Once we have realized the presence of such inconsistency, we can, for example, modify the handler for the UpdateSibling message to remove the new sibling from the children list. Previously, researchers had successfully used explicit-state model checking to identify inconsistencies in distributed systems [22] and reported a number of safety and liveness bugs in Mace implementations. However, due to an exponential explosion of possible states, current techniques capable of model checking distributed system implementations take a prohibitively long time to identify inconsistencies, even for seemingly short sequences such as the ones needed to generate states in Figure 2. For example, when we applied the Mace Model Checker's [22] exhaustive search to the safety properties of RandTree starting from the initial state, it failed to identify the inconsistency in Figure 2 even after running for 17 hours (on a 3.4-GHz Pentium-4 Xeon that we used for all our experiments in Section 4). The reason for this long running time is the large number of states reachable from the initial state up to the depth at which the bug occurs, all of which are examined by an exhaustive search.

1.3 CrystalBall Overview

Instead of running the model checker from the initial state, we propose to execute a model checker concurrently with the running distributed system, and continuously feed current system states into the model checker. When, in our example, the system reaches the state at the beginning of Figure 2, the model checker will predict the state at the end of Figure 2 as a possible future inconsistency. In summary, instead of trying to predict all possible inconsistencies starting from the initial state (which for complex protocols means never exploring states beyond the initialization phase), our model checker predicts inconsistencies that can occur in a system that has been running for a significant amount of time in a realistic environment.

As Figure 1 suggests, compared to the standard model checking approach, this approach identifies inconsistencies that can occur within much longer system executions. Compared to simply running the system for a long time, our approach has two advantages.

1. Our approach systematically covers a large number

of executions that contain low-probability events, such as node resets that ultimately triggered the inconsistency in Figure 2. It can take a very long time for a running system to encounter such a scenario, which makes testing for possible bugs difficult. Our technique therefore improves system debugging by providing a new technique that combines some of the advantages of testing and static analysis.

2. Our approach identifies inconsistencies before they actually occur. This is possible because the model checker can simulate packet transmission in time shorter than propagation latency, and because it can simulate timer events in time shorter than than the actual time delays. This aspect of our approach opens an entirely new possibility: adapt the behavior of the running system on the fly and avoid an inconsistency. We call this technique *execution steering*. Because it does not rely on a history of past inconsistencies, execution steering is applicable even to inconsistencies that were previously never observed in past executions.

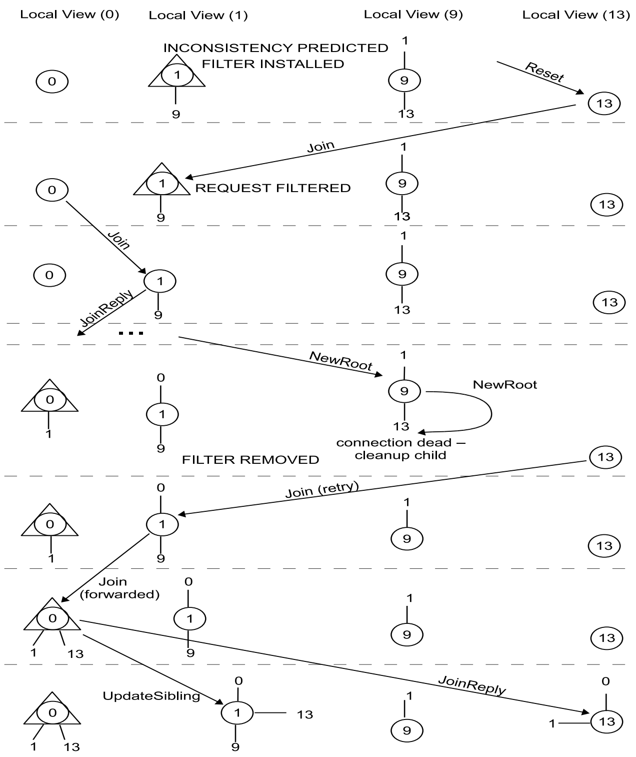


Figure 3: An Example execution sequence that avoids the inconsistency from Figure 2 thanks to execution steering.

Example of Execution Steering. In our example, a model checking algorithm running in n_1 detects the violation at the end of Figure 2. Given this knowledge, execution steering causes node n_1 not to respond to the join request of n_{13} and to break the TCP connection with it. Node n_{13} eventually succeeds joining the random tree (perhaps after some other nodes have joined first). The stale information about n_{13} in n_9 is removed once n_9 discovers that the stale communication channel with n_{13} is closed, which occurs the first time when n_9 attempts to communicate with n_{13}. Figure 3 presents one scenario illustrating this alternate execution sequence. Effectively, execution steering has exploited the non-determinism and robustness of the system to choose an alternative execution path that does not contain the inconsistency.

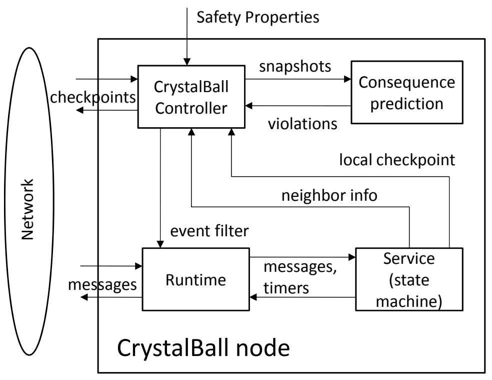


Figure 4: High-level overview of CrystalBall

2 CrystalBall Design

We next sketch the design of CrystalBall (see [44] for details). Figure 4 shows the high-level overview of a CrystalBall-enabled node. We concentrate on distributed systems implemented as state machines, as this is a widely-used approach [21, 25, 26, 37, 39].

The state machine interfaces with the outside world via the runtime module. The runtime receives the messages coming from the network, demultiplexes them, and invokes the appropriate state machine handlers. The runtime also accepts application level messages from the state machines and manages the appropriate network connections to deliver them to the target machines. This module also maintains the timers on behalf of all services that are running.

The CrystalBall controller contains a checkpoint manager that periodically collects consistent snapshots of a node's neighborhood. The controller feeds them to the model checker, along with a checkpoint of the local state. The model checker runs the consequence prediction algorithm which checks user- or developer-defined properties and reports any violation in the form of a sequence of events that leads to an erroneous state.

CrystalBall can operate in two modes. In the *deep online debugging mode* the controller only outputs the information about the property violation. In the *execution* *steering mode* the controller examines the report from the model checker, prepares an *event filter* that can avoid the erroneous condition, checks the filter's impact, and installs it into the runtime if it is deemed to be safe.

2.1 Consistent Neighborhood Snapshots

To check system properties, the model checker requires a snapshot of the system-wide state. Ideally, every node would have a consistent, up-to-date checkpoint of every other participant's state. Doing so would give every node high confidence in the reports produced by the model checker. However, given that the nodes could be spread over a high-latency wide-area network, this goal is unattainable. In addition, the sheer amount of bandwidth required to disseminate checkpoints might be excessive.

Given these fundamental limitations, we use a solution that aims for scalability: we apply model checking to a *subset* of all states in a distributed system. We leverage the fact that in scalable systems a node typically communicates with a small subset of other participants ("neighbors") and perform model checking only on this neighborhood. In some distributed hash table implementations, a node keeps track of $O(\log n)$ other nodes; in mesh-based content distribution systems nodes communicate with a constant number of peers; or this number does not explicitly grow with the size of the system. In a random overlay tree, a node is typically aware of the root, its parent, its children, and its siblings. We therefore arrange for a node to distribute its state checkpoints to its neighbors, and we refer to them as *snapshot neighborhood*. The *checkpoint manager* maintains checkpoints and snapshots. Other CrystalBall components can request an on-demand snapshot to be gathered by invoking an appropriate call on the checkpoint manager.

Discovering and Managing Snapshot Neighborhoods. To propagate checkpoints, the checkpoint manager needs to know the set of a node's neighbors. This set is dependent upon a particular distributed service. We use two techniques to provide this list. In the first scheme, we ask the developer to implement a method that will return the list of neighbors. The checkpoint manager then periodically queries the service and updates its snapshot neighborhood.

Since changing the service code might not always be possible, our second technique uses a heuristic to determine the snapshot neighborhood. Specifically, we periodically query the runtime to obtain the list of open connections (for TCP), and recent message recipients (for UDP). We then cluster connection endpoints according to the communication times, and selects a sufficiently large cluster of recent connections.

Enforcing Snapshot Consistency. To avoid false positives, we ensure that the neighborhood snapshot corresponds to a consistent view of a distributed system at some point of logical time. There has been a large body of work in this area, starting with the seminal paper by Chandy and Lamport [5]. We use one of the recent algorithms for obtaining consistent snapshots [29], in which the general idea is to collect a set of checkpoints that do not violate the happens-before relationship [25] established by messages sent by the distributed service.

Instead of gathering a global snapshot, a node periodically sends a checkpoint request to the members of its snapshot neighborhood. Even though nodes receive checkpoints only from a subset of nodes, all distributed service and checkpointing messages are instrumented to carry the checkpoint number (logical clock) and each neighborhood snapshot is a fragment of a globally consistent snapshot. In particular, a node that receives a message with a logical timestamp greater than its own logical clock takes a forced checkpoint. The node then uses the forced checkpoint to contribute to the consistent snapshot when asked for it.

Node failures are commonplace in distributed systems, and our algorithm has to deal with them. The checkpoint manager proclaims a node to be dead if it experiences a communication error (e.g., a broken TCP connection) with it while collecting a snapshot. An additional cause for an apparent node failure is a change of a node's snapshot neighborhood in the normal course of operation (e.g., when a node changes parents in the random tree). In this case, the node triggers a new snapshot gather operation.

Checkpoint Content. Although the total footprint of some services might be very large, this might not necessarily be reflected in checkpoint size. For example, the Bullet$'$ [23] file distribution application has non-negligible total footprint, but the actual file content transferred in Bullet$'$ does not play any role in consistency detection. In general, the checkpoint content is given by a serialization routine. The developer can choose to omit certain parts of the state from serialized content and reconstruct them if needed at de-serialization time. As a result, checkpoints are smaller, and the code compensates the lack of serialized state when a local state machine is being created from a remote node's checkpoint in the model checker. We use a set of well-known techniques for managing checkpoint storage (quotas) and controlling the bandwidth used by checkpoints (bandwidth limits, compression).

2.2 Consequence Prediction Algorithm

The key to enabling fast prediction of future inconsistencies in CrystalBall is our consequence prediction algorithm, presented in Figure 5. For readability, we present

```
1  proc findConseq(currentState : G, property : (G → boolean))
2    explored = emptySet(); errors = emptySet();
3    localExplored = emptySet();
4    frontier = emptyQueue();
5    frontier.addLast(currentState);
6    while (!STOP_CRITERION)
7      state = frontier.popFirst();
8      if (!property(state))
9        errors.add(state); // predicted inconsistency found
10     explored.add(hash(state));
11     foreach ((n,s) ∈ state.L) // node n in local state s
12       // process all network handlers
13       foreach (((s,m),(s',c)) ∈ $H_M$ where (n,m) ∈ state.I)
14         // node n handles message m according to st. machine
15         addNextState(state,n,s,s',{m},c);
16       // process local actions only for fresh local states
17       if (!localExplored.contains(hash(n,s)))
18         foreach (((s,a),(s',c)) ∈ $H_A$)
19           addNextState(state,n,s,s',{},c);
20       localExplored.add(hash(n,s));
21
22 proc addNextState(state,n,s,s',c0,c)
23   nextState.L = (state.L \ {(n,s)}) ∪ {(n,s')};
24   nextState.I = (state.I \ c0) ∪ c;
25   if (!explored.contains(hash(nextState)))
26     frontier.addLast(nextState);
```

Figure 5: Consequence Prediction Algorithm

the algorithm as a refinement of a generic state-space search. The notation is based on a high-level semantics of a distributed system, shown in Figure 6. (Our concrete model checker implementation uses an iterative deepening algorithm which combines memory efficiency of depth-first search, while favoring the states in the near future, as in breadth-first search.) The STOP_CRITERION in Figure 5 in our case is given by time constraints and external commands to restart the model checker upon the arrival of a new snapshot.

In Line 8 of Figure 5 the algorithm checks whether the explored state satisfies the desired safety properties. The developer can use a simple language [22] that involves loops, existential and comparison operators, state variables, and function invocations to specify the properties.

Exploring Independent Chains. We can divide the actions in a distributed system into *event chains*, where each chain starts with an application or scheduler event and continues by triggering network events. We call two chains *independent* if no event of the first chain changes state of a node involved in the second chain. Consequence Prediction avoids exploring the interleavings of independent chains. Therefore, the test in Line 17 of Figure 5 makes the algorithm re-explore the scheduler and application events of a node if and only if the previous events changed the local state of the node. For dependent chains, if a chain event changes local state of a node, Consequence Prediction therefore explores all other active chains which have been initiated from this node.

N – node identifiers
S – node states
M – message contents
$N \times M$ – (destination process, message)-pair
$C = 2^{N \times M}$ – set of messages with destination
A – local node actions (timers, application calls)

system state : $(L, I) \in G, \;\; G = 2^{N \times S} \times 2^{N \times M}$
local node states : $L \subseteq N \times S$ (function from N to S)
in-flight messages (network) : $I \subseteq N \times M$

behavior functions for each node :
message handler : $\;\; H_M \subseteq (S \times M) \times (S \times C)$
internal action handler : $H_A \subseteq (S \times A) \times (S \times C)$

transition function for distributed system :

node message handler execution :

$$\frac{((s_1, m), (s_2, c)) \in H_M}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I_0 \uplus \{(n, m)\}) \leadsto \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I_0 \cup c)\end{array}}$$

internal node action (timer, application calls) :

$$\frac{((s_1, a), (s_2, c)) \in H_A}{\begin{array}{ll}\text{before:} & (L_0 \uplus \{(n, s_1)\}, I) \leadsto \\ \text{after:} & (L_0 \uplus \{(n, s_2)\}, I \cup c)\end{array}}$$

Figure 6: A Simple Model of a Distributed System

Note that $hash(n, s)$ in Figure 5 implies that we have separate tables corresponding to each node for keeping hashed local states. If a state variable is not necessary to distinguish two separate states, the user can annotate the state variable that he or she does not want include in the hash function, improving the performance of Consequence Prediction. Instead of holding all encountered hashes, the hash table could be designed as a bounded cache to fit into the L2 cache or main memory, favoring access speed while admitting the possibility of re-exploring previously seen states.

Although simple, the idea of removing from the search actions of nodes with previously seen states eliminates many (uninteresting) interleavings from search and has a profound impact on the search depth that the model checker can reach with a limited time budget. This change was therefore key to enabling the use of the model checker at runtime. Knowing that consequence prediction avoids considering certain states, the question remains whether the remaining states are sufficient to make the search useful. Ultimately, the answer to this question comes from our evaluation (Section 4).

2.3 Execution Steering

CrystalBall's execution steering mode enables the system to avoid entering an erroneous state by steering its execution path away from predicted inconsistencies. If a protocol was designed with execution steering in mind, the runtime system could report a predicted inconsistency as a special programming language exception, and allow the service to react to the problem using a service-specific policy. However, to measure the impact on existing implementations, this paper focuses on generic runtime mechanisms that do not require the developer to insert exception-handling code.

Event Filters. Recall that a node in our framework operates as a state machine and processes messages, timer events, and application calls via handlers. Upon noticing that running a certain handler can lead to an erroneous state, CrystalBall installs an *event filter*, which temporarily blocks the invocation of the state machine handler for messages from the relevant sender.

The rationale is that a distributed system often contains a large amount of non-determinism that allows it to proceed even if certain transitions are disabled. For example, if the offending message is a Join request in a random tree, ignoring the message can prevent violating a local state property. The joining nodes can later retry the procedure with an alternative potential parent and successfully join the tree. Similarly, if handling a message causes an equivalent of a race condition manifested as an inconsistency, delaying message handling allows the system to proceed to the point where handling the message becomes safe again. Note that state machine handlers are atomic, so CrystalBall is unlikely to interfere with any existing recovery code.

Point of Intervention. In general, execution steering can intervene at several points in the execution path. Our current policy is to steer the execution as early as possible. For example, if the erroneous execution path involves a node issuing a Join request after resetting, the system's first interaction with that node occurs at the node which receives its join request. If this node discovers the erroneous path, it can install the event filter.

Non-Disruptiveness of Execution Steering. Ideally, execution steering would always prevent inconsistencies from occurring, without introducing new inconsistencies due to a change in behavior. In general, however, guaranteeing the absence of inconsistencies is as difficult as guaranteeing that the entire program is error-free. CrystalBall therefore makes execution steering safe in practice through two mechanisms:

1. **Sound Choice of Filters.** It is important that the chosen corrective action does not sacrifice the soundness of the state machine. A *sound filtering* is the one in which the observed sequence of events after filtering is a subset of possible sequence of events without filtering. The breaking of a TCP connection is common in a distributed system using TCP. Therefore, such distributed systems include failure-handling code that deals with broken TCP connections. This makes sending a TCP RST signal a good candidate for a sound event filter, and is the filter we choose to use in CrystalBall. In the case of communication over UDP, the filter simply drops the UDP packet, which could similarly happen in normal operation of the network.

2. **Exploration of Corrected Executions.** Before allowing the event filter to perform an execution steering action, CrystalBall runs the consequence prediction algorithm to check the effect of the event filter action on the system. If the consequence prediction algorithm does not suggest that the filter actions are safe, CrystalBall does not attempt execution steering and leaves the system to proceed as usual.

Rechecking Previously Discovered Violations. An event filter reflects possible future inconsistencies reachable from the current state, and leaving an event filter in place indefinitely could deny service to some distributed system participants. CrystalBall therefore removes the filters from the runtime after every model checking run. However, it is useful to quickly check whether the previously identified error path can still lead to an erroneous condition in a new model checking run. This is especially important given the asynchronous nature of the model checker relative to the system messages, which can prevent the model checker from running long enough to rediscover the problem. To prevent this from happening, the first step executed by the model checker is to replay the previously discovered error paths. If the problem reappears, CrystalBall immediately reinstalls the appropriate filter.

Immediate Safety Check. CrystalBall also supports *immediate safety check*, a mechanism that avoids inconsistencies that would be caused by executing the current handler. Such imminent inconsistencies can happen even in the presence of execution steering because 1) consequence prediction explores states given by only a subset of all distributed system nodes, and 2) the model checker runs asynchronously and may not always detect inconsistencies in time. The immediate safety check speculatively runs the handler, checks the consistency properties in the resulting state, and prevents actual handler execution if the resulting state is inconsistent.

We have found that exclusively using immediate safety check would not be sufficient for avoiding inconsistencies. The advantages of installing event filters are: i) performance benefits of avoiding the bug sooner, e.g., reducing unnecessary message transmission, ii) faster reaction to an error, which implies greater chance of avoiding a "point of no return" after which error avoidance is impossible, and iii) the node that is supposed to ultimately avoid the inconsistency by immediate safety check might not have all the checkpoints needed to notice the violation; this can result in false negatives (as shown in Figure 9).

Liveness Issues. It is possible that by applying an event filter would affect liveness properties of a distributed system. In our experience, due to a large amount of nondeterminism (e.g., the node is bootstrapped with a list of multiple nodes it can join), the system usually finds a way to make progress. We focus on enforcing safety properties, and we believe that occasionally sacrificing liveness is a valid approach. According to a negative result by Fischer, Lynch, and Paterson [12], it is impossible to have both in an asynchronous system anyway. (For example, the Paxos [26] protocol guarantees safety but not liveness.)

2.4 Scope of Applicability

CrystalBall does not aim to find all errors; it is rather designed to find and avoid important errors that can manifest in real runs of the system. Results in Section 4 demonstrate that CrystalBall works well in practice. Nonetheless, we next discuss the limitations of our approach and characterize the scenarios in which we believe CrystalBall to be effective.

Up-to-Date Snapshots. For Consequence Prediction to produce results relevant for execution steering and immediate safety check, it needs to receive sufficiently many node checkpoints sufficiently often. (Thanks to snapshot consistency, this is not a problem for deep online debugging.) We expect the stale snapshots to be less of an issue with *stable properties*, e.g., those describing a deadlock condition [5]. Since the node's own checkpoint might be stale (because of enforcing consistent neighborhood snapshots for checking multi-node properties), immediate safety check is perhaps more applicable to node-local properties.

Higher frequency of changes in state variables requires higher frequency of snapshot exchanges. High-frequency snapshot exchanges in principle lead to: 1) more frequent model checker restarts (given the difficulty in building incremental model checking algorithms), and 2) high bandwidth consumption. Among the examples for which our techniques is appropriate are overlays in which state changes are infrequent.

Consequence Prediction as a Heuristic. Consequence Prediction is a heuristic that explores a subset of the search space. This is an expected limitation of explicit-state model checking approaches applied to concrete implementations of large software systems. The key question in these approaches is directing the search towards most interesting states. Consequence Prediction uses information about the nature of the distributed system to guide the search; the experimental results in Section 4 show that it works well in practice, but we expect that further enhancements are possible.

3 Implementation Highlights

We built CrystalBall on top of the Mace [21] framework. Mace allows distributed systems to be specified succinctly and outputs high-performance C++ code. We implemented our consequence prediction within the Mace model checker, and run the model checker as a separate thread that communicates future inconsistencies to the runtime. Our current implementation of the immediate safety check executes the handler in a copy of the state machine's virtual memory (using fork()), and holds the transmission of messages until the successful completion of the consistency check. Upon encountering an inconsistency in the copy, the runtime does not execute the handler in the primary state machine. In case of applications with high messaging/state change rates in which the performance of immediate safety check is critical, we could obtain a state checkpoint [41] before running the handler and rollback to it in case of an encountered inconsistency. Another option would be to employ operating system-level speculation [32].

4 Evaluation

Our experimental evaluation addresses the following questions: **1)** Is CrystalBall effective in finding bugs in live runs? **2)** Can any of the bugs found by CrystalBall also be identified by the MaceMC model checker alone? **3)** Is execution steering capable of avoiding inconsistencies in deployed distributed systems? **4)** Are the CrystalBall-induced overheads within acceptable levels?

4.1 Experimental Setup

We conducted our live experiments using ModelNet [43]. ModelNet allows us to run live code in a cluster of machines, while application packets are subjected to packet delay, loss, and congestion typical of the Internet. Our cluster consists of 17 older machines with dual 3.4 GHz Pentium-4 Xeons with hyper-threading, 8 machines with dual 2.33 GHz dual-core Xeon 5140s, and 3 machines with 2.83 GHz Xeon X3360s (for Paxos experiments). Older machines have 2 GB of RAM, while the newer ones have 4 GB and 8 GB. These machines run GNU/Linux 2.6.17. One 3.4 GHz Pentium-4 machine running FreeBSD 4.9 served as the ModelNet packet forwarder for these experiments. All machines are interconnected with a full-rate 1-Gbps Ethernet switch.

We consider two deployment scenarios. For our large-scale experiments with deep online debugging, we multiplex 100 logical end hosts running the distributed service across the 20 Linux machines, with 2 participants running the model checker on 2 different machines. We run with 6 participants for small-scale debugging experiments, one per machine.

We use a 5,000-node INET [6] topology that we further annotate with bandwidth capacities for each link. The INET topology preserves the power law distribution of node degrees in the Internet. We keep the latencies generated by the topology generator; the average network RTT is 130ms. We randomly assign participants to act as clients connected to one-degree stub nodes in the topology. We set transit-transit links to be 100 Mbps, while we set access links to 5 Mbps/1 Mbps inbound-/outbound bandwidth. To emulate the effects of cross traffic, we instruct ModelNet to drop packets at random with a probability chosen uniformly at random between [0.001,0.005] separately for each link.

4.2 Deep Online Debugging Experience

We have used CrystalBall to find inconsistencies (violations of safety properties) in two mature implemented protocols in Mace, namely an overlay tree (RandTree) and a distributed hash table (Chord [42]). These implementation were not only manually debugged both in local- and wide-area settings, but were also model checked using MaceMC [22]. We have also used our tool to find inconsistencies in Bullet′, a file distribution system that was originally implemented in MACEDON [37], and then ported to Mace. We found 13 new subtle bugs in these three systems that caused violation of safety properties.

System	Bugs found	LOC Mace/C++
RandTree	7	309 / 2000
Chord	3	254 / 2200
Bullet′	3	2870 / 19628

Table 1: Summary of inconsistencies found for each system using CrystalBall. LOC stands for lines of code and reflects both the MACE code size and the generated C++ code size. The low LOC counts for Mace service implementations are a result of Mace's ability to express these services succinctly. This number does not include the line counts for libraries and low-level services that services use from the Mace framework.

Table 1 summarizes the inconsistencies that CrystalBall found in RandTree, Chord and Bullet′. Typical elapsed times (wall clock time) until finding an inconsistency in our runs have been from less than an hour up to a day. This time allowed the system being debugged to go through complex realistic scenarios.[1] CrystalBall identified inconsistencies by running consequence prediction from the current state of the system for up to several hundred seconds. To demonstrate their depth and complexity, we detail four out of 13 inconsistencies we found in the three services we examined.

[1]During this time, the model checker ran concurrently with a normally executing system. We therefore do not consider this time to be wasted by the model checker before deployment; rather, it is the time consumed by a running system.

4.2.1 Example RandTree Bugs Found

We next discuss bugs we identified in the RandTree overlay protocol presented in Section 1.2. We name bugs according to the consistency properties that they violate.

Children and Siblings Disjoint. The first safety property we considered is that the children and sibling lists should be disjoint. CrystalBall identified the scenario from Figure 2 in Section 1.2 that violates this property. The problem can be corrected by removing the stale information about children in the handler for the UpdateSibling message. CrystalBall also identified variations of this bug that requires changes in other handlers.

Recovery Timer Should Always Run. An important safety property for RandTree is that the recovery timer should always be scheduled. This timer periodically causes the nodes to send Probe messages to the peer list members with which it does not have direct connection. It is vital for the tree's consistency to keep nodes up-to-date about the global structure of the tree. The property was written by the authors of [22] but the authors did not report any violations of it. We believe that our approach discovered it in part because our experiments considered more complex join scenarios.

Scenario exhibiting inconsistency. CrystalBall found a violation of the property in a state where node A joins itself, and changes its state to "joined" but does not schedule any timers. Although this does not cause problems immediately, the inconsistency happens when another node B with smaller identifier tries to join, at which point A gives up the root position, selects B as the root, and adds B it to its peer list. At this point A has a non-empty peer list but no running timer.

Possible correction. Keep the timer scheduled even when a node has an empty peer list.

4.2.2 Example Chord Bug Found

We next describe a violation of a consistency property in Chord [42], a distributed hash table that provides key-based routing functionality. Chord and other related distributed hash tables form a backbone of a large number of proposed and deployed distributed systems [17, 35, 38].

Chord Topology. Each Chord node is assigned a Chord id (effectively, a key). Nodes arrange themselves in an overlay ring where each node keeps pointers to its predecessor and successor. Even in the face of asynchronous message delivery and node failures, Chord has to maintain a ring in which the nodes are ordered according to

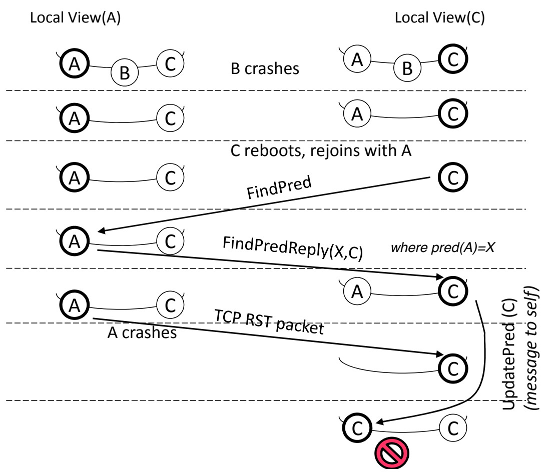


Figure 7: An inconsistency in a run of Chord. Node C has its predecessor pointing to itself while its successor list includes other nodes.

their ids, and each node has a set of "fingers" that enables it to reach exponentially larger distances on the ring.

Joining the System. To join the Chord ring, a node A first identifies its potential predecessor by querying with its id. This request is routed to the appropriate node P, which in turn replies to A. Upon receiving the reply, A inserts itself between P and P's successor, and sends the appropriate messages to its predecessor and successor nodes to update their pointers. A "stabilize" timer periodically updates these pointers.

Property: If Successor is Self, So Is Predecessor. If a predecessor of a node A equals A, then its successor must also be A (because then A is the only node in the ring). This is a safety property of Chord that had been extensively checked using MaceMC, presumably using both exhaustive search and random walks.

Scenario exhibiting inconsistency: CrystalBall found a state where node A has A as a predecessor but has another node B as its successor. This violation happens at depths that are beyond those reachable by exhaustive search from the initial state. Figure 7 shows the scenario. During live execution, several nodes join the ring and all have a consistent view of the ring. Three nodes A, B, and C are placed consecutively on the ring, i.e., A is predecessor of B and B is predecessor of C. Then B experiences a node reset and other nodes which have established TCP connection with B receive a TCP RST. Upon receiving this error, node A removes B from its internal data structures. As a consequence, Node A considers C as its immediate successor.

Starting from this state, consequence prediction detects the following scenario that leads to violation. C experiences a node reset, losing all its state. C then tries to rejoin the ring and sends a FindPred message to A. Because nodes A and C did not have an established TCP connection, A does not observe the reset of C. Node A replies to C by a FindPredReply message that shows A's successor to be C. Upon receiving this message, node C i) sets its predecessor to A; ii) stores the successor list included in the message as its successor list; and iii) sends an UpdatePred message to A's successor which, in this case, is C itself. After sending this message, C receives a transport error from A and removes A from all of its internal structures including the predecessor pointer. In other words, C's predecessor would be unset. Upon receiving the (loopback) message to itself, C observes that the predecessor is unset and then sets it to the sender of the UpdatePred message which is C. Consequently, C has its predecessor pointing to itself while its successor list includes other nodes.

Possible corrections. One possibility is for nodes to avoid sending UpdatePred messages to themselves (this appears to be a deliberate coding style in Mace Chord). If we wish to preserve such coding style, we can alternatively place a check after updating a node's predecessor: if the successor list includes nodes in addition to itself, avoid assigning the predecessor pointer to itself.

4.2.3 Example Bullet′ Bug Found

Next, we describe our experience of applying CrystalBall to the Bullet′ [23] file distribution system. The Bullet′ source sends the blocks of the file to a subset of nodes in the system; other nodes discover and retrieve these blocks by explicitly requesting them. Every node keeps a file map that describes blocks that it currently has. A node participates in the discovery protocol driven by RandTree, and peers with other nodes that have the most disjoint data to offer to it. These peering relationships form the overlay mesh.

Bullet′ is more complex than RandTree, Chord (and tree-based overlay multicast protocols) because of 1) the need for senders to keep their receivers up-to-date with file map information, 2) the block request logic at the receiver, and 3) the finely-tuned mechanisms for achieving high throughput under dynamic conditions. The starting point for our exploration was property 1):

Sender's File Map and Receivers View of it Should Be Identical. Every sender keeps a "shadow" file map for each receiver informing it which are the blocks it has not told the receiver about. Similarly, a receiver keeps a file map that describes the blocks available at the sender. Senders use the shadow file map to compute "diffs" on-demand for receivers containing information about blocks that are "new" relative to the last diff.

Senders and receivers communicate over non-blocking TCP sockets that are under control of MaceTcpTransport. This transport queues data on top of the TCP socket buffer, and refuses new data when its buffer is full.

Scenario exhibiting inconsistency: In a live run lasting less than three minutes, CrystalBall quickly identified a mismatch between a sender's file map and the receiver's view of it. The problem occurs when the diff cannot be accepted by the underlying transport. The code then clears the receiver's shadow file map, which means that the sender will never try again to inform the receiver about the blocks containing that diff. Interestingly enough, this bug existed in the original MACEDON implementation, but there was an attempt to fix it by the UCSD researchers working on Mace. The attempted fix consisted of retrying later on to send a diff to the receiver. Unfortunately, since the programmer left the code for clearing the shadow file map after a failed send, all subsequent diff computations will miss the affected blocks.

Possible corrections. Once the inconsistency is identified, the fix for the bug is easy and involves not clearing the sender's file map for the given receiver when a message cannot be queued in the underlying transport. The next successful enqueuing of the diff will then correctly include the block info.

4.3 Comparison with MaceMC

To establish the baseline for model checking performance and effectiveness, we installed our safety properties in the original version of MaceMC [22]. We then ran it for the three distributed services for which we identified safety violations. After 17 hours, exhaustive search did not identify any of the violations caught by CrystalBall, and reached the depth of only Some of the specific depths reached by the model checker are as follows 1) RandTree with 5 nodes: 12 levels, 2) RandTree with 100 nodes: 1 level, 3) Chord with 5 nodes: 14 levels, and Chord with 100 nodes: 2 levels. This illustrates the limitations of exhaustive search from the initial state.

In another experiment, we additionally employed random walk feature of MaceMC. Using this setup, MaceMC identified some of the bugs found by CrystalBall, but it still failed to identify 2 Randtree, 2 Chord, and 3 Bullet$'$ bugs found by CrystalBall. In Bullet$'$, MaceMC found no bugs despite the fact that the search lasted 32 hours. Moreover, even for the bugs found, the long list of events that lead to a violation (on the order of hundreds) made it difficult for the programmer to identify the error (we spent five hours tracing one of the violations involving 30 steps). Such a long event list is unsuitable for execution steering, because it describes a low probability way of reaching the final erroneous state. In contrast, CrystalBall identified violations that are close to live executions and therefore more likely to occur in the immediate future.

4.4 Execution Steering Experience

We next evaluate the capability of CrystalBall as a runtime mechanism for steering execution away from previously unknown bugs.

4.4.1 RandTree Execution Steering

To estimate the impact of execution steering on deployed systems, we instructed the CrystalBall controller to check for violations of RandTree safety properties (including the one described in Section 4.2.1). We ran a live churn scenario in which one participant (process in a cluster) per minute leaves and enters the system on average, with 25 tree nodes mapped onto 25 physical cluster machines. Every node was configured to run the model checker. The experiment ran for 1.4 hours and resulted in the following data points, which suggest that in practice the execution steering mechanism is not disruptive for the behavior of the system.

When CrystalBall is not active, the system goes through a total of 121 states that contain inconsistencies. When only the immediate safety check but not the consequence prediction is active, the immediate safety check engages 325 times, a number that is higher because blocking a problematic action causes further problematic actions to appear and be blocked successfully. Finally, we consider the run in which both execution steering and the immediate safety check (as a fallback) are active. Execution steering detects a future inconsistency 480 times, with 65 times concluding that changing the behavior is unhelpful and 415 times modifying the behavior of the system. The immediate safety check fallback engages 160 times. Through a combined action of execution steering and immediate safety check, CrystalBall avoided all inconsistencies, so there were no uncaught violations (false negatives) in this experiment.

To understand the impact of CrystalBall actions on the overall system behavior, we measured the time needed for nodes to join the tree. This allowed us to empirically address the concern that TCP reset and message blocking actions can in principle cause violations of liveness properties (in this case extending the time nodes need to join the tree). Our measurements indicated an average node join times between 0.8 and 0.9 seconds across different experiments, with variance exceeding any difference between the runs with and without CrystalBall. In summary, CrystalBall changed system actions 415 times (2.77% of the total of 14956 actions executed), avoided all specified inconsistencies, and did not degrade system performance.

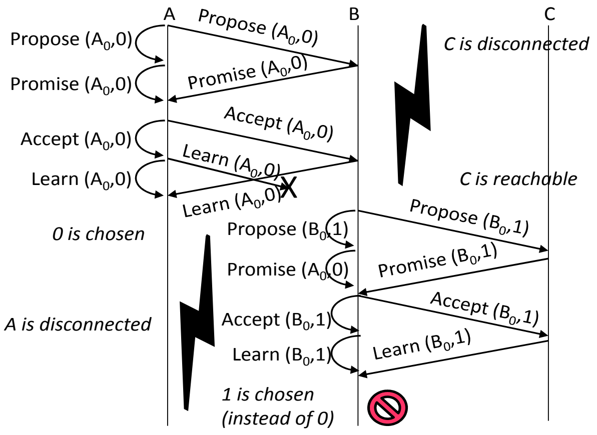


Figure 8: Scenario that exposes a previously reported Paxos violation of a safety property (two different values are chosen in the same instance).

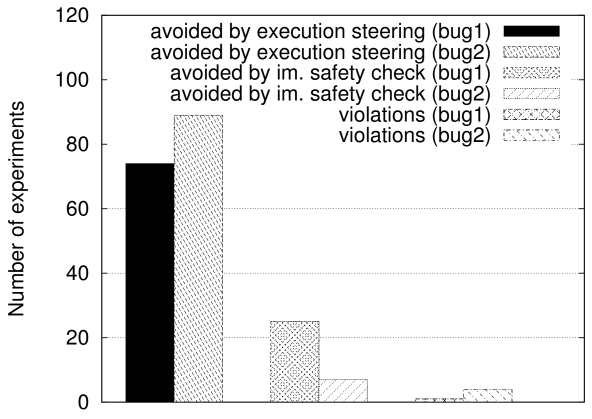


Figure 9: In 200 runs that expose Paxos safety violations due to two injected errors, CrystalBall successfully avoided the inconsistencies in all but 1 and 4 cases, respectively.

4.4.2 Paxos Execution Steering

Paxos [26] is a well known fault-tolerant protocol for achieving consensus in distributed systems. Recently, it has been successfully integrated in a number of deployed [4, 28] and proposed [19] distributed systems. In this section, we show how execution steering can be applied to Paxos to steer away from realistic bugs that have occurred in previously deployed systems [4, 28]. The Paxos protocol includes five steps:

1. A leader tries to take the leadership position by sending Prepare messages to acceptors, and it includes a unique round number in the message.
2. Upon receiving a Prepare message, each acceptor consults the last promised round number. If the message's round number is greater than that number, the acceptor responds with a Promise message that contains the last accepted value if there is any.
3. Once the leader receives a Promise message from the majority of acceptors, it broadcasts an Accept request to all acceptors. This message contains the value of the Promise message with the highest round number, or is any value if the responses reported no proposals.
4. Upon the receipt of the Accept request, each acceptor accepts it by broadcasting a Learn message containing the Accepted value to the learners, unless it had made a promise to another leader in the meanwhile.
5. By receiving Learn messages from the majority of the nodes, a learner considers the reported value as chosen.

The implementation we used was a baseline Mace Paxos implementation that includes a minimal set of features. In general, a physical node can implement one or more of the roles (leader, acceptor, learner) in the Paxos algorithm; each node plays all the roles in our experiments. The safety property we installed is the original Paxos safety property: at most one value can be chosen, across all nodes. The first bug we injected [28] is related to an implementation error in step 3, and we refer to it as *bug1*: once the leader receives the Promise message from the majority of nodes, it creates the Accept request by using the submitted value from the last Promise message instead of the Promise message with highest round number. Because the rate at which the violation (due to the injected error) occurs was low, we had to schedule some events to lead the live run toward the violation in a repeatable way. The setup we use comprises 3 nodes and two rounds, without any artificial packet delays. As illustrated in Figure 8, in the first round the communication between node C and the other nodes is broken. Also, a Learn packet is dropped from A to B. At the end of this round, A chooses the value proposed by itself (0). In the second round, the communication between A and other nodes is broken. At the end of this round, the value proposed by B (1) is chosen by B itself.

The second bug we injected (inspired by [4]) involves keeping a promise made by an Acceptor, even after crashes and reboots. As pointed in [4], it is often difficult to implement this aspect correctly, especially under various hardware failures. Hence, we inject an error in the way a promise is kept by not writing it to disk (we refer to it as *bug2*). To expose this bug we use a scenario similar to the one used for *bug1*, with the addition of a reset of node B.

To stress test CrystalBall's ability to avoid inconsistencies at runtime, we repeat the live scenarios in the cluster 200 times (100 times for each bug) while varying the time between rounds uniformly at random between 0 and 20 seconds. As we can see in Figure 9, CrystalBall's execution steering is successful in avoiding the inconsistency at runtime 74% and 89% of the time for *bug1* and *bug2*, respectively. In these cases, CrystalBall starts model checking after node C reconnects and receives checkpoints from other participants. After running the model checker for 3.3 seconds, C successfully predicts that the scenario in the second round would result in violation of the safety property, and it then installs the event filter. The avoidance by execution steering happens when C rejects the Propose message sent by B. Execution steering is more effective for *bug2* than for *bug1*, as the former involves resetting B. This in turn leaves more time for the model checker to rediscover the problem by: i) consequence prediction, or ii) replaying a previously identified erroneous scenario. Immediate safety check engages 25% and 7% of the time, respectively (in cases when model checking did not have enough time to uncover the inconsistency), and prevents the inconsistency from occurring later, by dropping the Learn message from C at node B. CrystalBall could not prevent the violation for only 1% and 4% of the runs, respectively. The cause for these false negatives was the incompleteness of the set of checkpoints.

4.5 Performance Impact of CrystalBall

Memory, CPU, and bandwidth consumption. Because consequence prediction runs in a separate process that is most likely mapped to a different CPU core on modern processors, we expect little impact on the service performance. In addition, since the model checker does not cache previously visited states (it only stores their hashes) the memory is unlikely to become a bottleneck between the model-checking CPU core and the rest of the system.

One concern with state exploration such as model-checking is the memory consumption. Figure 10 shows the consequence prediction memory footprint as a function of search depth for our RandTree experiments. As expected, the consumed memory increases exponentially with search depth. However, since the effective CrystalBall's search depth in is less than 7 or 8, the consumed memory by the search tree is less than 1MB and can thus easily fit in the L2 or L3 (most recently) cache of the state of the art processors. Having the entire search tree in-cache reduces the access rate to main memory and improves performance.

In the deep online debugging mode, the model checker was running for 950 seconds on average in the 100-node case, and 253 seconds in the 6-node case. When running in the execution steering mode (25 nodes), the model checker ran for an average of about 10 seconds. The checkpointing interval was 10 seconds.

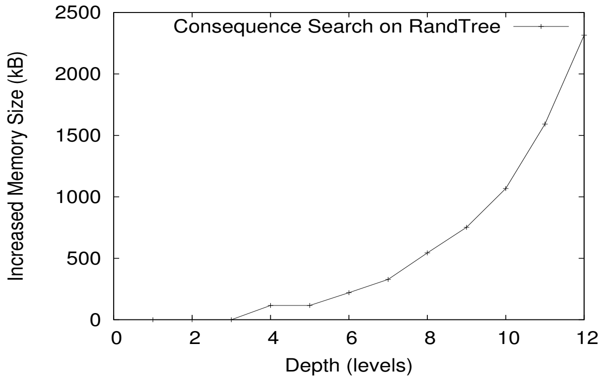


Figure 10: The memory consumed by consequence prediction (RandTree, depths 7 to 8) fits in an L2 CPU cache.

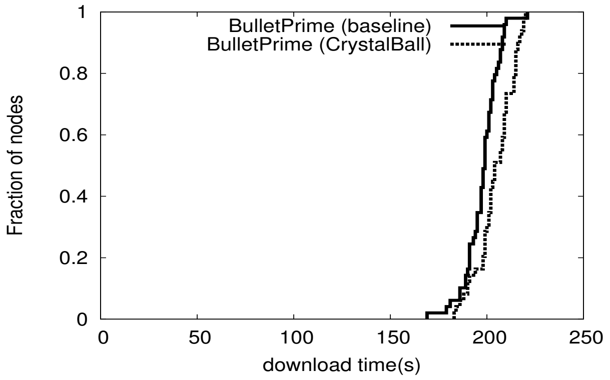


Figure 11: CrystalBall slows down Bullet′ by less than 10% for a 20 MB file download.

The average size of a RandTree node checkpoint is 176 bytes, while a Chord checkpoint requires 1028 bytes. Average per-node bandwidth consumed by checkpoints for RandTree and Chord (100-nodes) was 803 bps and 8224 bps, respectively. These figures show that overheads introduced by CrystalBall are low. Hence, we did not need to enforce any bandwidth limits in these cases.

Overhead from Checking Safety Properties. In practice we did not find the overhead of checking safety properties to be a problem because: i) the number of nodes in a snapshot is small, ii) the most complex of our properties have $O(n^2)$ complexity, where n is the number of nodes, and iii) the state variables fit into L2 cache.

Overall Impact. Finally, we demonstrate that having CrystalBall monitor a bandwidth-intensive application featuring a non-negligible amount of state such as Bullet′ does not significantly impact the application's performance. In this experiment, we instructed 49 Bullet′ instances to download a 20 MB file. Bullet′ is not a CPU intensive application, although computing the next block to request from a sender has to be done quickly. It is therefore interesting to note that in 34 cases during this experiment the Bullet′ code was competing with the model checker for the Xeon CPU with hyper-threading. Figure 11 shows that in this case using CrystalBall reduced performance by less than 5%. Compressed Bullet′ checkpoints were about 3 kB in size, and the bandwidth that was used for checkpoints was about 30 Kbps per node (3% of a node's outbound bandwidth of 1 Mbps). The reduction in performance is therefore primarily due to the bandwidth consumed by checkpoints.

5 Related Work

Debugging distributed systems is a notoriously difficult and tedious process. Developers typically start by using an ad-hoc logging technique, coupled with strenuous rounds of writing custom scripts to identify problems. Several categories of approaches have gone further than the naive method, and we explain them in more detail in the remainder of this section.

Collecting and Analyzing Logs. Several approaches (Magpie [2], Pip [34]) have successfully used extensive logging and off-line analysis to identify performance problems and correctness issues in distributed systems. Unlike these approaches, CrystalBall works on deployed systems, and performs an online analysis of the system state.

Deterministic Replay with Predicate Checking. Friday [14] goes one step further than logging to enable a gdb-like replay of distributed systems, including watch points and checking for global predicates. WiDS-checker [28] is a similar system that relies on a combination of logging/checkpointing to replay recorded runs and check for user predicate violations. WiDS-checker can also work as a simulator. In contrast to replay- and simulation-based systems, CrystalBall explores additional states and can steer execution away from erroneous states.

Online Predicate Checking. Singh *et al.* [40] have advocated debugging by online checking of distributed system state. Their approach involves launching queries across the distributed system that is described and deployed using the OverLog/P2 [40] declarative language/runtime combination. D^{3}S [27] enables developers to specify global predicates which are then automatically checked in a deployed distributed system. By using binary instrumentation, D3S can work with legacy systems. Specialized *checkers* perform predicate-checking topology on snapshots of the nodes' states. To make the snapshot collection scalable, the checker's *snapshot neighborhood* can be manually configured by the developer. This work has shown that it is feasible to collect snapshots at runtime and check them against a set of user-specified properties. CrystalBall advances the state-of-the-art in online debugging in two main directions: 1) it employs an efficient algorithm for model checking from a live state to search for bugs "deeper" and "wider" than in the live run, and it 2) enables execution steering to automatically prevent previously unidentified bugs from manifesting themselves in a deployed system.

Model Checking. Model checking techniques for finite state systems [16, 20] have proved successful in analysis of concurrent finite state systems, but require the developer to manually abstract the system into a finite-state model which is accepted as the input to the system. Early efforts on explicit-state model checking of C and C++ implementations [31, 30, 46] have primarily concentrated on a single-node view of the system.

MODIST [45] and MaceMC [22] represent the state-of-the-art in model checking distributed system implementations. MODIST [45] is capable of model checking unmodified distributed systems; it orchestrates state space exploration across a cluster of machines. MaceMC runs state machines for multiple nodes within the same process, and can determine safety and liveness violations spanning multiple nodes. MaceMC's exhaustive state exploration algorithm limits in practice the search depth and the number of nodes that can be checked. In contrast, CrystalBall's consequence prediction allows it to achieve significantly shorter running times for similar depths, thus enabling it to be deployed at runtime. In [22] the authors acknowledge the usefulness of prefix-based search, where the execution starts from a given supplied state. Our work addresses the question of obtaining prefixes for prefix-based search: we propose to directly feed into the model checker states as they are encountered in live system execution. Using CrystalBall we found bugs in code that was previously debugged in MaceMC and that we were not able to reproduce using MaceMC's search. In summary, CrystalBall differs from MODIST and MaceMC by being able to run state space exploration from live state. Further, CrystalBall supports execution steering that enables it to automatically prevent the system from entering an erroneous state.

Cartesian abstraction [1] is a technique for over-approximating state space that treats different state components independently. The independence idea is also present in our consequence prediction, but, unlike over-approximating analyses, bugs identified by consequence search are guaranteed to be real with respect to the model explored. The idea of disabling certain transitions in state-space exploration appears in partial-order reduction (POR) [15],[13]. Our initial investigation suggests that a POR algorithm takes considerably longer than the consequence prediction algorithm. The advantage of POR is its completeness, but completeness is of second-order importance in our case because no complete search can terminate in a reasonable amount of time for state spaces of distributed system implementations.

Runtime Mechanisms. In the context of operating systems, researchers have proposed mechanisms that safely re-execute code in a changed environment to avoid errors [33]. Such mechanisms become difficult to deploy in the context of distributed systems. Distributed transactions are a possible alternative to execution steering, but involve several rounds of communication and are inapplicable in environments such as wide-area networks. A more lightweight solution involves forming a FUSE [11] failure group among all nodes involved in a join process. Making such approaches feasible would require collecting snapshots of the system state, as in CrystalBall. Our execution steering approach reduces the amount of work for the developer because it does not require code modifications. Moreover, our experimental results show an acceptable computation and communication overhead.

In Vigilante [9] and Bouncer [8], end hosts cooperate to detect and inform each other about worms that exploit even previously unknown security holes. Hosts protect themselves by generating filters that block bad inputs. Relative to these systems, CrystalBall deals with distributed system properties, and predicts inconsistencies before they occur.

Researchers have explored modifying actions of concurrent programs to reduce data races [18] by inserting locks in an approach that does not employ running static analysis at runtime. Approaches that modify state of a program at runtime include [10, 36]; these approaches enforce program invariants or memory consistency without computing consequences of changes to the state.

6 Conclusions

We presented a new approach for improving the reliability of distributed systems, where nodes predict and avoid inconsistencies before they occur, even if they have not manifested in any previous run. We believe that our approach is the first to give running distributed system nodes access to such information about their future. To make our approach feasible, we designed and implemented consequence prediction, an algorithm for selectively exploring future states of the system, and developed a technique for obtaining consistent information about the neighborhood of distributed system nodes. Our experiments suggest that the resulting system, CrystalBall, is effective in finding bugs that are difficult to detect by other means, and can steer execution away from inconsistencies at runtime.

Acknowledgments

We thank our shepherd Arvind Krishnamurthy and the anonymous reviewers who provided excellent feedback. We also thank Barbara Jobstmann for useful discussions, James Anderson for his help with the Mace Paxos implementation, and Charles Killian for answering questions about MaceMC. Nikola Knežević was funded in part by a grant from the Hasler foundation (grant 2103).

References

[1] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. In *TACAS*, 2001.

[2] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[3] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh. Splitstream: High-bandwidth Content Distribution in Cooperative Environments. In *SOSP*, October 2003.

[4] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: an Engineering Perspective. In *PODC*, 2007.

[5] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[6] Hyunseok Chang, Ramesh Govindan, Sugih Jamin, Scott Shenker, and Walter Willinger. Towards Capturing Representative AS-Level Internet Topologies. In *SIGMETRICS*, June 2002.

[7] Yang Chu, S. G. Rao, S. Seshan, and Hui Zhang. A Case for End System Multicast. *Selected Areas in Communications, IEEE Journal on*, 20(8):1456–1471, Oct 2002.

[8] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. In *SOSP*, 2007.

[9] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *SOSP*, October 2005.

[10] Brian Demsky and Martin Rinard. Automatic Detection and Repair of Errors in Data Structures. In *OOPSLA*, 2003.

[11] John Dunagan, Nicholas J. A. Harvey, Michael B. Jones, Dejan Kostić, Marvin Theimer, and Alec Wolman. FUSE: Lightweight Guaranteed Distributed Failure Notification. In *OSDI*, 2004.

[12] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[13] Cormac Flanagan and Patrice Godefroid. Dynamic Partial-Order Reduction for Model Checking Software. In *POPL*, 2005.

[14] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *NSDI*, 2007.

[15] Patrice Godefroid and Pierre Wolper. A Partial Approach to Model Checking. *Inf. Comput.*, 110(2):305–326, 1994.

[16] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[17] Navendu Jain, Prince Mahajan, Dmitry Kit, Praveen Yalagandula, Mike Dahlin, and Yin Zhang. Network Imprecision: A New Consistency Metric for Scalable Monitoring. In *OSDI*, December 2008.

[18] Muhammad Umar Janjua and Alan Mycroft. Automatic Correction to Safety Violations. In *Thread Verification (TV06)*, 2006.

[19] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson, and Arun Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, San Francisco, April 2008.

[20] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *LICS*, 1990.

[21] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: Language Support for Building Distributed Systems. In *PLDI*, 2007.

[22] Charles E. Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[23] Dejan Kostić, Ryan Braud, Charles Killian, Erik Vandekieft, James W. Anderson, Alex C. Snoeren, and Amin Vahdat. Maintaining High Bandwidth under Dynamic Network Conditions. In *USENIX ATC*, 2005.

[24] Dejan Kostić, Adolfo Rodriguez, Jeannie Albrecht, Abhijeet Bhirud, and Amin Vahdat. Using Random Subsets to Build Scalable Network Services. In *USITS*, 2003.

[25] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Com. of the ACM*, 21(7):558–565, 1978.

[26] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[27] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M. Frans Kaashoek, and Zheng Zhang. D^3S: Debugging Deployed Distributed Systems. In *NSDI*, 2008.

[28] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[29] D. Manivannan and Mukesh Singhal. Asynchronous Recovery Without Using Vector Timestamps. *J. Parallel Distrib. Comput.*, 62(12):1695–1728, 2002.

[30] Madanlal Musuvathi and Dawson R. Engler. Model Checking Large Network Protocol Implementations. In *NSDI*, 2004.

[31] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, 2002.

[32] Edmund B. Nightingale, Peter M. Chen, and Jason Flinn. Speculative Execution in a Distributed File System. In *SOSP*, 2005.

[33] Feng Qin, Joseph Tucek, Jagadeesan Sundaresan, and Yuanyuan Zhou. Rx: Treating Bugs as Allergies—a Safe Method to Survive Software Failures. In *SOSP*, 2005.

[34] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul A. Shah, and Amin Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[35] Sean Rhea, Brighten Godfrey, Brad Karp, John Kubiatowicz, Sylvia Ratnasamy, Scott Shenker, Ion Stoica, and Harlan Yu. OpenDHT: A Public DHT Service and Its Uses. In *SIGCOMM*, 2005.

[36] Martin C. Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *OSDI*, 2004.

[37] Adolfo Rodriguez, Charles Killian, Sooraj Bhat, Dejan Kostić, and Amin Vahdat. MACEDON: Methodology for Automatically Creating, Evaluating, and Designing Overlay Networks. In *NSDI*, 2004.

[38] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. In *SOSP*, 2001.

[39] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: a Tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[40] Atul Singh, Petros Maniatis, Timothy Roscoe, and Peter Druschel. Using Queries for Distributed Monitoring and Forensics. *SIGOPS Oper. Syst. Rev.*, 40(4):389–402, 2006.

[41] Sudarshan M. Srinivasan, Srikanth K, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *USENIX ATC*, 2004.

[42] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.

[43] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI*, December 2002.

[44] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. CrystalBall: Predicting and Preventing Inconsistencies in Deployed Distributed Systems. Technical report, EPFL, `http://infoscience.epfl.ch/record/124918`, 2008.

[45] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *NSDI*, April 2009.

[46] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using Model Checking to Find Serious File System Errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, 2006.

Tolerating latency in replicated state machines through client speculation

Benjamin Wester* James Cowling† Edmund B. Nightingale◇
Peter M. Chen* Jason Flinn* Barbara Liskov†

*University of Michigan** *MIT CSAIL†* *Microsoft Research◇*

Abstract

Replicated state machines are an important and widely-studied methodology for tolerating a wide range of faults. Unfortunately, while replicas should be distributed geographically for maximum fault tolerance, current replicated state machine protocols tend to magnify the effects of high network latencies caused by geographic distribution. In this paper, we examine how to use speculative execution at the clients of a replicated service to reduce the impact of network and protocol latency. We first give design principles for using client speculation with replicated services, such as generating early replies and prioritizing throughput over latency. We then describe a mechanism that allows speculative clients to make new requests through replica-resolved speculation and predicated writes. We implement a detailed case study that applies this approach to a standard Byzantine fault tolerant protocol (PBFT) for replicated NFS and counter services. Client speculation trades in 18% maximum throughput to decrease the effective latency under light workloads, letting us speed up run time on single-client micro-benchmarks 1.08–19$\times$ when the client is co-located with the primary. On a macro-benchmark, reduced latency gives the client a speedup of up to 5$\times$.

1 Introduction

As more of society depends on services running on computers, tolerating faults in these services is increasingly important. Replicated state machines [34] provide a general methodology to tolerate a wide variety of faults, including hardware failures, software crashes, and malicious attacks. Numerous examples exist for how to build such replicated state machines, such as those based on agreement [8, 11, 22, 25] and those based on quorums [1, 11].

For replicated state machines to provide increased fault tolerance, the replicas should fail independently. Various aspects of failure independence can be achieved by using multiple computers, independently written software [2, 33], and separate administrative domains. Geographic distribution is one important way to achieve failure independence when confronted with failures such as power outages, natural disasters, and physical attacks.

Unfortunately, distributing the replicas geographically increases the network latency between replicas, and many protocols for replicated state machines are highly sensitive to latency. In particular, protocols that tolerate Byzantine faults must wait for multiple replicas to reply, so the effective latency of the service is limited by the latency of the slowest replica being waited for. Agreement-based protocols further magnify the effects of high network latency because they use multiple message rounds to reach agreement. Some implementations may also choose to delay requests and batch them together to improve throughput.

Our work uses speculative execution to allow clients of replicated services to be less sensitive to high latencies caused by network delays and protocol messages. We observe that faults are generally rare, and, in the absence of faults, the response from even a single replica is an excellent predictor of the final, collective response from the replicated state machine. Based on this observation, clients in our system can proceed after receiving the first response, thereby hiding considerable latency in the common case in which the first response is correct, especially if at least one replica is located nearby. When responses are completely predictable, clients can even continue before they receive any response.

To provide safety in the rare case in which the first response is incorrect, a client in our system may only continue executing *speculatively*, until enough responses are collected to confirm the prediction. By tracking all effects of the speculative execution and not externalizing speculative state, our system can undo the effects of the speculation if the first response is later shown to be incorrect.

Because client speculation hides much of the latency of the replicated service from the client, replicated servers in our system are freed to optimize their behavior to maximize their throughput and minimize load, such as by handling agreement in large batches.

We show how client speculation can help clients of a replicated service tolerate network and protocol latency by adding speculation to the Practical Byzantine Fault Tolerance (PBFT) protocol [8]. We demonstrate how performance improves for a counter service and an NFSv2 service on PBFT from decreased effective latency and increased concurrency in light workloads. Speculation improves the client throughput of the counter service 2–58$\times$ across two different network topologies. Speculation speeds up the run time of NFS micro-benchmarks 1.08–19$\times$ and up to 5$\times$ on a macro-benchmark when co-locating a replica with the client. When replicas are equidistant from each other, our benchmarks speed up by 1.06–6$\times$ and 2.2$\times$, respectively. The decrease in latency that client speculation provides does have a cost: under heavy workloads, maximum throughput is decreased by 18%.

We next describe our general approach to adding client speculation to a system with a replicated service.

2 Client speculation in replicated services

2.1 Speculative execution

Speculative execution is a general latency-hiding technique. Rather than wait for the result of a slow operation, a computer system may instead predict the outcome of that operation, checkpoint its state, and speculatively execute further operations using the predicted result. If the speculation is correct, the checkpoint is committed and discarded. If the speculation is incorrect, it is aborted, and the system rolls back its state to the checkpoint and re-executes further operations using the correct result.

In general, speculative execution is beneficial only if the time to checkpoint state is less than the time to perform the operation that generates the result. Further, the outcome of that operation must be predictable. Incorrect speculations waste resources since all work that depends on a mispredicted result is thrown away. This waste lowers throughput, especially when multiple entities are participating in a distributed system, since the system might have been able to service other entities in lieu of doing work for the incorrect speculation. Thus, the decision of whether or not to speculate on the result of an operation often boils down to determining which operations will be slow and which slow operations have predictable results.

2.2 Applicability to replicated services

Replicated services are an excellent candidate for client-based speculative execution. Clients of replicated state machine protocols that tolerate Byzantine faults must wait for multiple replicas to reply. That may mean waiting for multiple rounds of messages to be exchanged among replicas in an agreement-based protocol. If replicas are separated by geographic distances (as they should be in order to achieve failure independence), network latency introduces substantial delay between the time a client starts an operation and the time the client receives the reply that commits the operation. Thus, there is substantial time available to benefit from speculative execution, especially if one replica is located near the client.

Replicated services also provide an excellent predictor of an operation's result. Under the assumption that faults are rare, a client's request will generate identical replies from every replica, so the first reply that a client receives is an excellent predictor of the final, collective reply from the replicated state machine (which we refer to as the *consensus reply*). After receiving the first reply to any operation, a client can speculate *based on 1 reply* with high confidence. For example, when an NFS client tries to read an uncached file, it cannot predict what data will be returned, so it must wait for the first reply before it can continue with reasonable data.

The results of some remote operations can be predicted even before receiving any replies; for instance, an NFS client can predict with high likelihood of success that file system updates will succeed and that read operations will return the same (possibly stale) values in its cache [28]. For such operations, a client may speculate *based on 0 replies* since it can predict the result of a remote operation with high probability.

2.3 Protocol adjustments

Based on the above discussion, it becomes clear that some replicated state machine protocols will benefit more from speculative execution than others. For this reason, we propose several adjustments to protocols that increase the benefit of client-based speculation.

2.3.1 Generate early replies

Since the maximum latency that can be hidden by speculative execution, in the absence of 0-reply speculation, is the time between when the client receives the first reply from any replica and when the client receives enough replies to determine the consensus response, a protocol should be designed to get the first reply to the client as quickly as possible. The fastest reply is realized when the client sends its request to the closest replica, and that replica responds immediately. Thus, a protocol that supports client speculation should have one or more replicas immediately respond to a client with the replica's best guess for the final outcome of the operation, as long as that guess can accurately predict the consensus reply.

Assuming each replica stores the complete state of the service, the closest replica can always immediately perform and respond to a read-only request. However, that reply is not guaranteed to be correct in the presence of concurrent write operations. It could be wrong if the closest replica is behind in the serial order of operations and returns a stale value, or in quorum protocols where the replica state has diverged and is awaiting repair [1]. We describe optimizations in Section 3.2.2 that allow early responses from any replica in the system, along with techniques to minimize the likelihood of an incorrect speculative read response.

It is more difficult to allow any replica to immediately execute a modifying request in an agreement protocol. Backup replicas depend on the primary replica to decide a single ordering of requests. Without waiting for that ordering, a backup could guess at the order, speculatively executing requests as it receives them. However, it is unlikely that each replica will perceive the same request ordering under workloads with concurrent writers, especially with geographic distribution of replicas. Should the guessed order turn out wrong (beyond acceptable levels [23]), the replica must roll back its state and re-execute operations in the committed order, hurting throughput and likely causing its response to change.

For agreement protocols like PBFT, a more elegant solution is to have only the primary execute the request early and respond to the client. As we explain in Section 3.3, such predictions are correct unless the primary is faulty. This solution enables us to avoid speculation or complex state management on the replicas that would reduce throughput. Used in this way, the primary should be located near the most active clients in a system to reduce their latency.

2.3.2 Prioritize throughput over latency

There exist a myriad of replicated state machine protocols that offer varying trade-offs between throughput and latency [1, 8, 11, 22, 30, 32, 37]. Given client support for speculative execution, it is usually best to choose a protocol that improves throughput over one that improves latency. The reason is that speculation can do much to hide replica latency but little to improve replica throughput.

As discussed in the previous section, speculative execution can hide the latency that occurs between the receipt of an early reply from a replica and the receipt of the reply that ends the operation. Thus, as long as a speculative protocol provides for early replies from the closest or primary replica, reducing the latency of the overall operation does not ordinarily improve user-perceived latency.

Speculation can only improve throughput in the case where replicas are occasionally idle by allowing clients to issue more operations concurrently. If the replicas are fully loaded, speculation may even decrease throughput because of the additional work caused by mispredictions or the generation of early replies. Thus, it seems prudent to choose a protocol that has higher latency but higher potential throughput, perhaps through batching, and stable performance under write contention [8, 22], rather than protocols that optimize latency over throughput [1, 11].

An important corollary of this observation is that client speculation allows one to choose simpler protocols. With speculation, a complex protocol that is highly optimized to reduce latency may perform approximately the same as a simpler, higher latency protocol from the viewpoint of a user. A simpler protocol has many benefits, such as allowing a simpler implementation that is quicker to develop, is less prone to bugs, and may be more secure because of a smaller trusted computing base.

2.3.3 Avoid speculative state on replicas

To ensure correctness, speculative execution must avoid *output commits* that externalize speculative output (e.g., by displaying it to a user) since such output can not be undone once externalized. The definition of what constitutes external output, however, can change. For instance, sending a network message to another computer would be considered an output commit if that computer did not support speculation. However, if that computer could be trusted to undo, if necessary, any changes that causally depend on the receipt of the message, then the message would not be an output commit. One can think of the latter case as enlarging the *boundary of speculation* from just a single computer to encompass both the sender and receiver.

What should be the boundary of speculation for a replicated service? At least three options are possible: allow all replicas and clients of the service to share speculative state, allow replicas to share speculative state with individual clients but not to propagate one client's speculative state to other clients, and disallow replicas from storing speculative state.

Our design uses the third option, with the smallest boundary of speculation, for several reasons. First, the complexity of the system increases as more parts participate in a speculation. The system would need to use distributed commit and rollback [14] to involve replicas and other clients in the speculation, and the interaction between such a distributed commit and the normal replicated service commit would need to be examined carefully. Second, as the boundary of speculation grows larger, the cost of a misprediction is higher; all replicas and clients that see speculative state must roll back all actions that depend on that state when a prediction is wrong. Finally, it may be difficult to precisely track dependencies as they propagate through the data structures of a replica, and any false dependencies in a replica's state may force clients to trust each other in ways not required by the data they share in the replicated service.

For example, if the system takes the simple approach of tainting the entire replica state, then one client's misprediction would force the replica to roll back all later operations, causing unrelated clients to also roll back.

2.3.4 Use replica-resolved speculation

Even with this small boundary of speculation, we would still like to allow clients to issue new requests that depend on speculative state (which we call *speculative requests*). Speculative requests allow a client to continue submitting requests when it would otherwise be forced to block. These additional requests can be handled concurrently, increasing throughput when the replicas are not already fully saturated.

One complication here is that, to maintain correctness, if one of the prior operations on which the client is speculating fails, any dependent operations that the client issues must also abort. There is currently no mechanism for a replica to determine whether or not a client received a correct speculative response. Thus, the replica is unable to detect whether or not to execute subsequent dependent speculative requests.

To overcome this flaw, we propose *replica-resolved speculation through predicated writes*, in which replicas are given enough information to determine whether the speculations on which requests depend will commit or abort. With predicated writes, an operation that modifies state includes a list of the active speculations on which it depends, along with the predicted responses for those speculations. Replicas log each committed response they send to clients and compare each predicted response in a predicated write with the actual response sent. If all predicated responses match the saved versions, the speculative request is consistent with the replica's responses, and it can execute the new request. If the responses do not match, the replica knows that the client will abort this operation when rolling back a failed speculation, so it discards the operation. This approach assumes a protocol in which all non-faulty replicas send the same response to a request.

Note that few changes may need to be made to a protocol to handle speculative requests that modify data. An operation O that depends on a prior speculation O_s, with predicted response r, may simply be thought of as a single deterministic request to the replicated service of the predicated form: `if` $response(O_s) = r$, `then do` O. This predicate must be enforced on the replicas. However, as shown in Section 5, predicate checking may be performed by a shim layer between the replication protocol and the application without modifying the protocol itself.

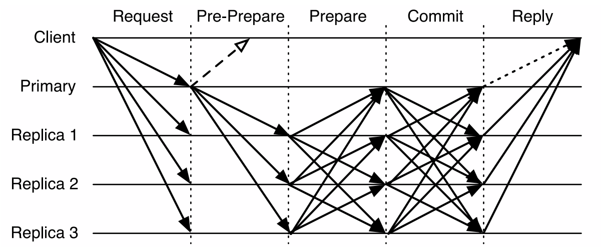


Figure 1: PBFT-CS Protocol Communication. The early response from the primary is shown with a dashed hollow arrow, which replaces its response from the Reply phase (dotted filled arrow) in PBFT.

3 Client speculation for PBFT

In this section, we apply our general strategy for supporting client speculative execution in replicated services to the Practical Byzantine Fault Tolerance (PBFT) protocol. We call the new protocol we develop PBFT-CS (CS denotes the additional support for client speculation).

3.1 PBFT overview

PBFT is a Byzantine fault tolerant state machine replication protocol that uses a primary replica to assign each client request a sequence number in the serial order of operations. The replicas run a three-phase agreement protocol to reach consensus on the ordering of each operation, after which they can execute the operation while ensuring consistent state at all non-faulty replicas. Optionally, the primary can choose and attach *non-deterministic data* to each request (for NFS, this contains the current time of day).

PBFT requires $3f + 1$ replicas to handle f concurrent faulty replicas, which is the theoretical minimum [5]. The protocol guarantees liveness and correctness with up to f failures, and runs a *view change* sub-protocol to move the primary to another replica in the case of a bad primary.

The communication pattern for PBFT is shown in Figure 1. The client normally receives a commit after five one-way message delays, although this may be shortened to four delays by overlapping the *commit* and *reply* phases using a *tentative execution* optimization [8]. To reduce the overhead of the agreement protocol, the primary may collect a number of client requests into a *batch* and run agreement once on the ordering of operations within this batch.

In our modified protocol, PBFT-CS, the primary responds immediately to client requests, as illustrated by the dashed line in Figure 1.

3.2 PBFT-CS base protocol

In both PBFT and PBFT-CS, the client sends each request to all replicas, which buffer the request for execution after agreement. Unlike the PBFT agreement protocol, the primary in PBFT-CS executes an operation immediately upon receiving a request and sends the early reply to the client as a speculative response. The primary then forms a pre-prepare message for the next batch of requests and continues execution of the agreement protocol. Other replicas are unmodified and reply to the client request once the operation has committed.

Since the primary determines the serial ordering of all requests, under normal circumstances the client will receive at least f committed responses from the replicas matching the primary's early response. This signifies that the speculation was correct because the request committed with the same value as the speculative response. If the client receives $f + 1$ matching responses that differ from the primary's response, the client rolls back the current speculation and resumes execution with the consensus response.

3.2.1 Predicated writes

A PBFT-CS client can issue subsequent requests immediately after predicting a response to an earlier request, rather than waiting for the earlier request to commit. To enable this without requiring replicas themselves to speculate and potentially roll back, PBFT-CS ensures that a request that modifies state does not commit if it depends on the value of any incorrect speculative responses. To meet this requirement, clients must track and propagate the dependencies between requests.

For example, consider a client that reads a value stored in a PBFT-CS database (`op1`), performs some computation on the data, then writes the result of the computation back to the database (`op2`). If the primary returns an incorrect speculative result for `op1`, the value to be written in `op2` will also be incorrect. When `op1` eventually commits with a different value, the client will fail its speculation and resume operation with the correct value. Although the client cannot undo the send of `op2`, dependency tracking prevents `op2` from writing its incorrect value to the database.

Each PBFT-CS client maintains a log of the digests d_T of each speculative response issued at logical timestamp T. When an operation commits, its corresponding digest is removed from the tail of the log. If an operation aborts, its digest is removed from the log, along with the digests of any dependent operations.

Clients append any required dependencies to each speculative request, of the form $\{c, \langle t_i, d_i \rangle, ...\}$ for client c and each digest d_i at timestamp t_i.

Replicas also store a log of digests for each client with the committed response for each operation. The replica executes a speculative request only if all digests in the request's dependency list match the entries in the replica's log. Otherwise, the replica executes a no-op in place of the operation.

It is infeasible for replicas to maintain an unbounded digest log for each client in a long-running system, so PBFT-CS truncates these logs periodically. Replicas must make a deterministic decision on when to truncate their logs to ensure that non-faulty replicas either all execute the operation or all abort it. This is achieved by truncating the logs at fixed deterministic intervals.

If a client issues a request containing a dependency that has since been discarded from the log, the replicas abort the operation, replacing it with a no-op. The client recognizes this scenario when receiving a consensus response that contains a special *retry* result. It retries execution once all its dependencies have committed. In practice an operation will not abort due to missing dependencies, provided that the log is sufficiently long to record all operations issued in the time between a replica executing an operation and a quorum of responses being received by the client.

3.2.2 Read-only optimization

Many state machine replication protocols provide a read-only optimization [1, 8, 11, 22] in which read requests can be handled by each replica without being run through the agreement protocol. This allows reads to complete in a single communication round, and it reduces the load on the primary.

In the standard optimization, a client issues optimized read requests directly to each replica rather than to the primary. Replicas execute and reply to these requests without taking any steps towards agreement. A client can continue after receiving $2f + 1$ matching replies. Because optimized reads are not serialized through the agreement protocol, other clients can issue conflicting, concurrent writes that prevent the client from receiving enough matching replies. When this happens, the client retransmits the request through the agreement protocol. This optimization is beneficial to workloads that contain a substantial percentage of read-only operations and exhibit few conflicting, concurrent writes. Importantly, when a backup replica is located nearer a client than the primary, that replica's reply will typically be received by the client before the primary's.

PBFT-CS cannot use this standard optimization without modification. A problem arises when a client issues a speculative request that depends on the predicted response to an optimized read request. PBFT-CS requires all non-faulty replicas to make a deterministic decision when verifying the dependencies on an operation. However, since optimized reads are *not* serialized by the agreement protocol, one non-faulty replica may see a conflicting write before responding to an optimized read, while another non-faulty replica sees the write after responding to the read. These two non-faulty replicas will thus respond to the optimized read with different values, and they will make different decisions when they verify the dependencies on a later speculative request. A non-faulty replica that sent a response that matches the first speculative response received by the client will commit the write operation, while other non-faulty replicas will not. Hence, writes may not depend on uncommitted optimized reads. This is enforced at each replica by not logging the response digest for such requests.

We address this problem by allowing a PBFT-CS client to resubmit optimized read requests through the full agreement protocol, forcing the replicas to agree on a common response. When write conflicts are low, the resubmitted read is likely to have the same reply as the initial optimized read, so a speculative prediction is likely to still be correct. After performing this procedure, we can send any dependent write requests, as they no longer depend on an optimized request.

There are three issues that must be considered for a read request to be submitted using this optimization.

- The request cannot read uncommitted state.
- The client should not follow a read with a write.
- The reply should not be completely predictable.

The first issue is required for consistency. A client cannot optimize a read request for a piece of state before all its write requests for that state are committed. Otherwise, it risks reading stale data when a sufficient number of backup replicas have not yet seen the client's previous writes. The data dependency tracking required to implement this policy is also used to propagate speculations, so no extra information needs to be maintained. Reads that do depend on uncommitted data may still be submitted through the agreement protocol as with write requests. Should a client desire a simpler policy for ensuring correctness, it can disable the read-only optimization while it has any uncommitted writes.

Second, consider a client that reads a value, performs a computation, and then writes back a new value. If the read request is initially sent optimized, issuing the write will force the read to be resubmitted. The "optimization" results in additional work. Clients that anticipate following a read by a write should decline to optimize the read.

Finally, if a client can predict the outcome of the request before receiving any replies (for instance, if it predicts that a locally-cached value has not become stale), then it should submit the request through the normal agreement protocol. Since the client does not need to wait for any replies, it is not hurt by the extra latency of waiting for agreement.

3.3 Handling failures

Speculation optimizes for reduced latency in the non-failure case, but it is important to ensure that correctness and liveness are maintained in the presence of faulty replicas. Failed speculations also increase the latency of a client's request, forcing it to roll back after having waited for the consensus response, and hurt throughput by forcing outstanding requests to become no-ops. It is important for our protocol to handle faults correctly in a way that still tries to preserve performance.

A speculation will fail on a client when the first reply it receives to a request does not match the consensus response. There are three cases in which this might happen:

- The most common case occurs when a write issued by another client conflicts with an optimized read. In an extreme instance, one replica's early reply could contain the stale data while all other replicas reply with current data.
- The second case occurs when there is a view change. PBFT ensures that committed requests will be ordered the same in the new view, but the client is speculating on uncommitted requests that the new replica could order differently. View changes may be the result of a bad primary, or they may be triggered by network conditions or proactive recovery [9].
- The third case occurs when the primary is faulty, and it either returns an incorrect speculative response or serializes a request differently when running the agreement protocol. We next examine this scenario further.

It is trivial for a client to detect a faulty primary: a request's early reply from the primary and the consensus reply will be in the same view and not match. If signed responses are used, the primary's bad reply can be given to other replicas as a proof of misbehavior. However, if simple message authentication codes (MACs) are used, the early reply cannot be used in this way since MACs do not provide non-repudiation.

The simplest solution to handling faults with MACs is for a client to stop speculating if the percentage of failed speculations it observes surpasses a threshold. PBFT-CS currently uses an arbitrary threshold of 1%. If a client observes that the percentage of failed speculations is greater than 1% over the past n early replies provided by a replica, it simply ceases to speculate on subsequent early replies from that replica. Although it will not speculate on subsequent replies, it can still track their accuracy and resume speculating on further replies if the percentage falls below a threshold. Our experimental results verify that at this threshold, PBFT-CS is still effective at reducing the average latency under light workloads.

3.4 Correctness

The speculative execution environment and PBFT protocol used in our system both have well-established correctness guarantees [7, 28]. We thus focus our attention on the modifications made to PBFT, to ensure that this protocol remains correct.

Our modified version of PBFT differs from the original in several key ways:

- A client may be sent a speculative response that differs from the final consensus value.
- A client may submit an operation that depends on a failed speculation.
- The primary may execute an operation before it commits.

We evaluate each modification independently.

Incorrect speculation A bad primary may send an incorrect speculative response to a client, in that it differs on the value or ordering of the final consensus value. We also consider in this class an honest primary that sends a speculative response to a client but is unable to complete agreement on this response due to a view change. In either case, the client will only see the consensus response once the operation has undergone agreement at a quorum of replicas. If the speculative response was incorrect, it is safe for the client to roll back the speculative execution and re-run using the consensus value, since PBFT ensures that all non-faulty replicas will agree on the consensus value.

Dependent operations A further complication arises when the client has issued subsequent requests that depend on the value of a speculative response. Here, the speculation protocol on the client ensures that it rolls back execution of any operations that have dependencies on the failed speculation. We must ensure that all valid replicas make an identical decision to abort each dependent operation by replacing it with a no-op.

Replicas maintain a log of the digests for each committed operation and truncate this log at deterministic intervals so that all non-faulty replicas have the same log state when processing a given operation. Predicated writes in PBFT-CS allow the client to express the speculation dependencies to the replicas. A non-faulty replica will not execute any operation that contains a dependency that does not match the corresponding digest in the log, or that does not have a matching log entry. Since the predicated write contains the same information used by the client when rolling back dependent operations, the replicas are guaranteed to abort any operation aborted by the client. If a client submits a dependency that has since been truncated from the log, it will also be aborted.

The only scenario where replicas are unable to deterministically decide whether a speculative response matches its agreed-upon value is when a speculative response was produced using the read-only optimization. Here, different replicas may have responded with different values to the read request. We explicitly avoid this case by making it an error to send a write request that depends on the reply to an optimized read request; correct clients will never issue such a request. Replicas do not store the responses to optimized reads in their log and hence always ignore any request sent by a faulty client with a dependency on an optimized read.

Speculative execution In our modified protocol, the primary executes client requests immediately upon their receipt, before the request has undergone agreement. The agreement protocol dictates that all non-faulty replicas commit operations in the order proposed by the primary, unless they execute a view change to elect a new primary. After a view change, the new primary may reorder some uncommitted operations executed by the previous primary, however, the PBFT view change protocol ensures that any committed operations persist into the new view. It is safe for the old primary to restore its state to the most recent committed operation since any incorrect speculative response will be rolled back by clients where necessary.

4 Discussion and future optimizations

In this section, we further explore the protocol design space for the use of client speculation with PBFT. We compare and contrast possible protocol alternatives with the PBFT-CS protocol that we have implemented.

4.1 Alternative failure handling strategies

We considered two alternative strategies for dealing with faulty primaries. First, we could allow clients to request a view change without providing a proof of misbehavior. This scheme would seem to significantly compromise liveness in a system containing faulty clients since they can force view changes at will. However, this is an existing problem in BFT state machine replication in the absence of signatures. A bad client in PBFT is always able to force a view change by sending a request to the primary with a bad authenticator that appears correct to the primary or by sending different requests to different replicas [7]. We could mitigate the damage a given bad client can do by having replicas make a local decision to ignore all requests from a client that 'framed' them. In this way a bad client can not initiate a view change after incriminating f primaries.

Alternatively, we could require signatures in communications between client and replicas. This is the most straight-forward solution, but entails significant CPU overhead. Compared to these two alternative designs, we chose to have PBFT-CS revert to a non-speculative protocol due to the simplicity of the design and higher performance in the absence of a faulty primary.

4.2 Coarse-grained dependency tracking

PBFT-CS tracks and specifies the dependencies of a speculative request at fine granularity. Thus, message size and state grow as the average number of dependencies for a given operation increases. To keep message size and state constant, we could use coarser-grained dependencies.

We could track dependencies on a per-client basis by ensuring that a replica executes a request from a client at logical timestamp T only if *all* outstanding requests from that client prior to time T have committed with the same value the client predicted.

Instead of maintaining a list of dependencies, each client would instead store a hash chained over all consensus responses and subsequent speculative responses. The client would append this hash to each operation in place of the dependency list. The client would also keep another hash chained only over consensus responses, which it would use to restore its dependency state after rolling back a failed speculation.

Each replica would maintain a hash chained over responses sent to the client and would execute an operation if the hash chain in the request matches its record of responses. Otherwise, it would execute a no-op.

We chose not to use this optimization in PBFT-CS since the use of chained hashes creates dependencies between all operations issued by a client even when no causal dependencies exist. This increases the cost of a failed speculation since the failure of one speculative request causes all subsequent in-progress speculative operations to abort. Coarse-grained dependency tracking also limits the opportunities for running speculative read operations while there are active speculative writes. Since speculative read responses are not serialized with respect to write operations, it is likely that the client will insert the read response in the wrong point in the hash chain, causing subsequent operations to abort.

4.3 Reads in the past

A read-only request need not circumvent the agreement protocol completely, as described in section 3.2.2. A client can instead take a hybrid approach for non-modifying requests: it can submit the request for full agreement and at the same time have the nearest replica immediately execute the request.

If the primary happens to be the nearest to the client, this is not a change from the normal protocol. When another replica is closer, the client can get a lower-latency first reply, plus having agreement eliminates the second consideration for optimized reads (in Section 3.2.2), that a client should not follow a read with a write.

However, this new optimization presents a problem when there are concurrent writes by multiple clients. A non-primary replica will execute an optimized request, and a client will speculate on its reply, in a sequential order that is likely different from the request's actual order in the agreement protocol. In essence, the read has been

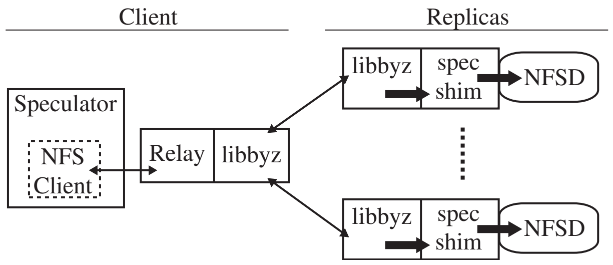


Figure 2: Speculative fault-tolerant NFS architecture

executed *in the past*, at a logical time when the replicas have not yet processed all operations that are undergoing agreement but when they still share a consistent state.

We could extend the PBFT-CS read-only optimization to also allow reads in the past. Under a typical configuration, there is only one round of agreement executing at any one time, with incoming requests buffered at the primary to run in the next batch of agreement. If we were to ensure that all buffered reads are reordered, when possible, to be serialized at the start of this next batch, it would be highly likely that no write will come between a read being received by a replica and the read being serialized after agreement.

Note that the primary may assign any order to requests within a batch as long as no operation is placed before one on which it depends. Recall that a PBFT-CS client will only optimize a read if the read has no outstanding write dependencies. Hence, the primary is free to move all speculative reads to the start of the batch. The primary executes these requests on a snapshot of the state taken before the batch began.

5 Implementation

We modified Castro and Liskov's PBFT library, *libbyz* [8], to implement the PBFT-CS protocol described in Section 3. We also modified BFS [8], a Byzantine-fault-tolerant replicated file service based on NFSv2, to support client speculation. The overall system can be divided into three parts as shown in Figure 2: the NFS client, a protocol relay, and the fault-tolerant service.

5.1 NFS client operation

Our client system uses the NFSv2 client module of the Speculator kernel [28], which provides process-level support for speculative execution. Speculator supports fine-grained dependency tracking and checkpointing of individual objects such as files and processes inside the Linux kernel. Local file systems are speculation-aware and can be accessed without triggering an output commit. Speculator buffers external output to the terminal, network, and other devices until the speculations on which they depend commit. Speculator rolls back process and OS state to checkpoints and restarts execution if a speculation fails.

To execute a remote NFS operation, Speculator first attaches a list of the process's dependencies to the message, then sends it to a relay process on the same machine. The relay interprets this list and attaches the correct predicates when sending the PBFT-CS request.

The relay brokers communication between the client and replicas. It appears to be a standard NFS server to the client, so the client need not deal with the PBFT-CS protocol. When the relay receives the first reply to a 1-reply speculation, the reply is logged and passed to the waiting NFS client. The NFS client recognizes speculative data, creates a new speculation, and waits for a confirmation message from the relay. Once the consensus reply is known, the relay sends either a `commit` message or a `rollback{reply}` message containing the correct response.

Our implementation speculates based on 0 replies for `GETATTR`, `SETATTR`, `WRITE`, `CREATE`, and `REMOVE` calls. It can speculate on 1 reply for `GETATTR`, `LOOKUP`, and `READ` calls. This list includes the most common NFS operations: we observed that at least 95% of all calls in all our benchmarks were handled speculatively. Note that we speculate on both 0 replies and 1 reply for `GETATTR` calls. The kernel can speculate as soon as it has attributes for a file. When the attributes are cached, 0 replies are needed, otherwise, the kernel waits for 1 reply before continuing.

5.2 PBFT-CS client operation

Speculation hides latency by allowing a single client to pipeline many requests; however, our PBFT implementation only allows for each PBFT-CS client to have a single outstanding request at any time. We work around this limitation by grouping up to 100 logical clients into a single client process.

NFS with 0-reply speculation requires its requests to be executed in the order they were issued. A PBFT-CS client process can tag each request with a sequence number so that the primary replica will only process requests from that client process's logical clients in the correct order. Of course, two different clients' requests can still be interleaved in any order by the primary.

To support this additional concurrency, we designed the client to use an event-driven API. User programs pass requests to libbyz and later receive two callbacks: one delivers the first reply and another delivers the consensus reply. The user program is responsible for monitoring libbyz's communication channels and timers.

5.3 Server operation

On the replicas, libbyz implements an event-based server that performs upcalls into the service when needed: to request non-deterministic data, to execute requests, and to construct error replies. The library handles all communication and state management, including checkpointing and recovery.

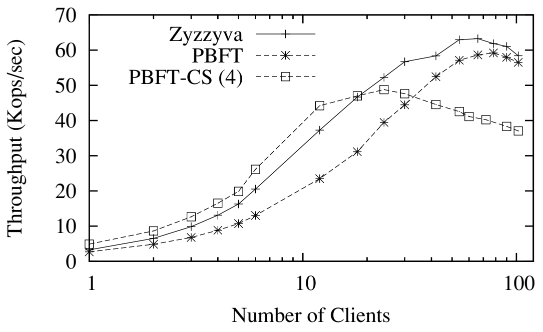


Figure 3: Server throughput in a LAN, measured on the **shared counter** service. PBFT-CS (4) is limited to four concurrent requests.

Overhead Source	Slowdown
Early replies	8.2%
Larger request	4.1%
Complex client	2.8%
Predicate checking	1.8%

Table 1: Major sources of overhead affecting throughput for PBFT-CS relative to PBFT.

A shim layer is used to manage dependencies on replicas. When writes need to be quashed due to failed speculative dependencies, the shim layer issues a no-op to the service instead. Thus, the underlying service is not exposed to details of the PBFT-CS protocol.

The primary will batch together all requests it receives while it is still agreeing on earlier requests. Batching is a general optimization that reduces the number of protocol instances that must be run, decreasing the number of communications and authentication operations [8, 22, 23, 37]. This implementation imposes a maximum batch size of 64 requests, a limit our benchmarks do run up against.

6 Evaluation

In this section, we quantify the performance of our PBFT-CS implementation. We have implemented a simple shared counter micro-benchmark and several NFS micro- and macro-benchmarks.

We compare PBFT-CS against two other Byzantine fault-tolerant agreement protocols: PBFT [8] and Zyzzyva [22]. PBFT is the base protocol we extend make use of client speculation. Its overall structure is illustrated in Figure 1. We use the tentative reply optimization, so each request must go through 4 communication phases before the client acquires a reply that it can act on.

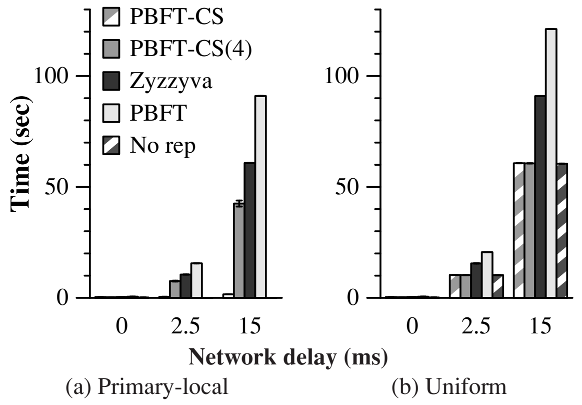


Figure 4: Time taken to run 2000 updates using the **shared counter** service. The primary-local topology (a) shows a client located at the same site as the primary. The uniform topology (b) shows a remote client equidistant from all sites. 0 ms (LAN) times for both graphs are (in bar order): 0.36 s, 0.27 s, 0.41 s, 0.54 s, and 0.16 s.

PBFT uses an adaptive batching protocol, allowing up to 64 requests to be handled in one agreement instance.

Zyzzyva is a recent agreement protocol that is heavily optimized for failure-free operation. When all replicas are non-faulty (as in our experiments), it takes only 3 phases for a client to possess a consensus reply. We run Kotla et al.'s implementation of Zyzzva, which uses a fixed batch size. We simulate an adaptive batching strategy by manually tuning the batch size as needed for best performance.

By comparison, a PBFT-CS client can continue executing speculatively after only 2 communication phases. We expect this to significantly reduce the effective latency of our clients. Note that requests still require 4 phases to *commit*, but we can handle those requests concurrently rather than sequentially. If we limit the number of in-flight requests to some number n, we call the protocol "PBFT-CS (n)."

6.1 Experimental setup

Each replica machine uses a single Intel Xeon 2.8 GHz processor with 512 MB RAM (sufficient for our applications). We always evaluate using four replicas without failures (unless noted). In our NFS comparisons, we use a single client that is identical in hardware to the replicas. Our counter service runs on an additional five client machines using Intel Pentium 4s or Xeons with clock speeds of 3.06–3.20 GHz and 1 GB RAM. All systems use a generic Red Hat Linux 2.4.21 kernel.

Our machines use gigabit Ethernet to communicate directly with a single switch. Experiments using the shared counter service were performed on a Cisco Catalyst 2970 gigabit switch; NFS used an Intel Express ES101TX 10/100 switch.

Our target usage scenario is a system that consists of several sites joined by moderate latency connections (but slower than LAN speeds). Each site has a high-speed LAN hosting one replica and several clients, and clients may also be located off-site from any replica. For comparison with other agreement protocols, we also consider using PBFT-CS in a LAN setting where all replicas and clients are on the same local segment.

Based on the above scenarios, we emulate a simplified test network using NISTNet [6] that inserts an equal amount of one-way latency between each site. We let this inserted *delay* be either 2.5 ms or 15 ms.

We also measure performance at clients located in different areas in our scenario. In the *primary-local topology*, the client is at the same site as the current primary replica. The *primary-remote topology* considers a client at different site hosting a backup replica. A client not present at any site is shown in the *uniform topology*, and we let the client have the same one-way latency to all replicas as between sites.

When comparing against a service with no replication in a given topology, we always assume that a client at a site can access its server using only the LAN. A client not at a site is still subject to added delay.

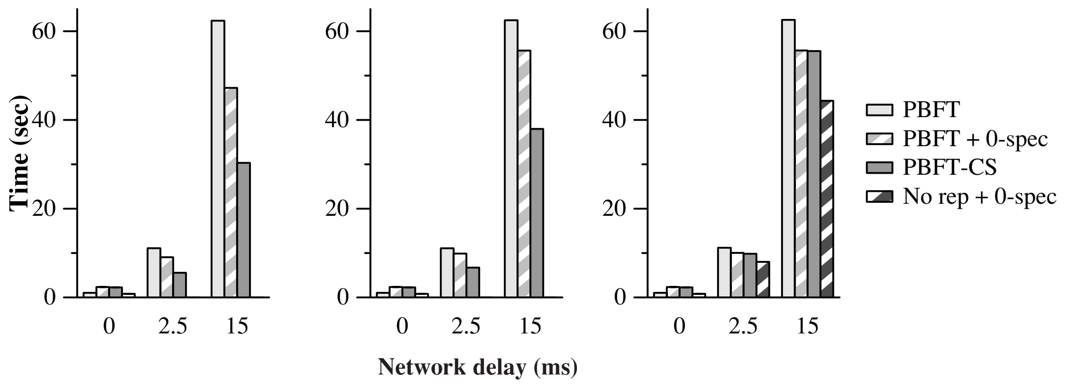
6.2 Counter throughput

We first examine the throughput of PBFT-CS using the counter service. Similar to Castro and Liskov's standard 0/0 benchmark [8], the counter's request and reply size are minimal. This service exposes only one operation: increment the counter and return its new value. Each reply contains a token that the client must present on its next request. This does add a small amount of processing time to each request, but it ensures that client requests must be submitted sequentially.

Our client is a simple loop that issues a fixed number of counter updates and records the total time spent. No state is externalized by the client, so we allow the client process to implement its own lightweight checkpoint mechanism. Checkpoint operations take negligible time, so our results focus on the characteristics of the protocol itself rather than our checkpoint mechanism.

We measure throughput by increasing the number of client processes per machine (up to 17 processes) until the server appears saturated. Graphs show the mean of at least 6 runs, and visible differences are statistically significant.

Figure 3 shows the measured throughput in a LAN configuration. We found that in this topology, a single PBFT-CS client gains no benefit from having more than 4 concurrent requests, and we enforce that limit on all clients. When we have 12 or fewer concurrent clients, PBFT-CS has 1.19–1.49× higher through-



(a) Primary-local (b) Primary-remote (c) Uniform

Figure 5: **Read-only** NFS micro-benchmark performance across different network topologies. The last three data sets use 0-reply speculation. At 0 ms, all three topologies are equivalent, so the same data is used for each graph. The *no rep* data show a lower bound for run time. There is only one *no rep* data set for primary-local and primary remote topologies, because the location of the server does not change with increasing latency. For these two graphs, the 0 ms bar applies to all latencies but is not repeated.

put than Zyzzyva and 1.79–2× higher throughput than PBFT.

In lightly loaded systems, the servers are not being fully utilized, and speculating clients can take advantage of the spare resources to decrease their own effective latency. As the server becomes more heavily loaded, those resources are no long free to use. As a result, PBFT-CS reaches its peak throughput before other protocols.

There is a trade-off of throughput for latency: PBFT-CS shows a peak throughput that is 17.6% lower than PBFT. We found four fundamental sources of overhead, summarized in Table 1. First, the client implementation for PBFT-CS uses an event-driven system to handle several logical clients, needed to support concurrent requests. This design does lead to a slower client than the one in PBFT, which can get by with a simpler blocking design. Second, we found that having the primary send early replies increases its time spent blocking while transmitting. Third, each predicate added to a request makes the request packet larger, and fourth, those predicates take additional work to verify on each replica.

6.3 Counter latency

We next examine how latency affects client performance under a light workload when the client is located at different sites. Figure 4 shows the time taken for a single counter client to issue 2000 requests in different topologies. In the LAN topology where no delay is added, a PBFT-CS client is able to complete the benchmark in 33% less time than PBFT, reflecting average run times of 357 ms and 538 ms respectively. When we increase the latency between sites, run time becomes dominated by number of communication phases. With a uniform topology (Figure 4b), PBFT-CS takes 50% less time than PBFT and 33% less time than Zyzzyva, and its runtime is only 1% slower than the unreplicated service. This matches our intuitive understanding of the protocol behavior described at the start of this section.

For PBFT-CS, the critical path is a round-trip communication with the primary replica. Moving to a primary-remote topology (bringing one backup replica closer) does not affect this critical path, and our measurements show no significant difference between primary-remote and uniform topologies.

Figure 4a presents results when using a primary-local topology. As latency increases and backup replicas move further from the client, performance does not degrade significantly, since the latency to the primary is fixed. At 15 ms latency, a client using PBFT takes 58× longer than with PBFT-CS. The combination of client speculation and a co-located primary achieves much of the performance benefit of a closely located non-replicated server, while providing all the guarantees of a geographically distributed replicated service that tolerates Byzantine faults.

These significant gains are directly attributable to the increased concurrency possible in the primary-local topology. When we limit PBFT-CS to only 4 outstanding requests, the client must then wait on requests to commit, reintroducing a dependence on communication delay. In topologies where the client does not have privileged access to the primary, as in the uniform topology, limiting concurrency has little effect.

6.4 NFS

We next examine PBFT-CS applied to an NFS server. Considering that the NFSv2 protocol is not explicitly designed for high-latency environments, we compare

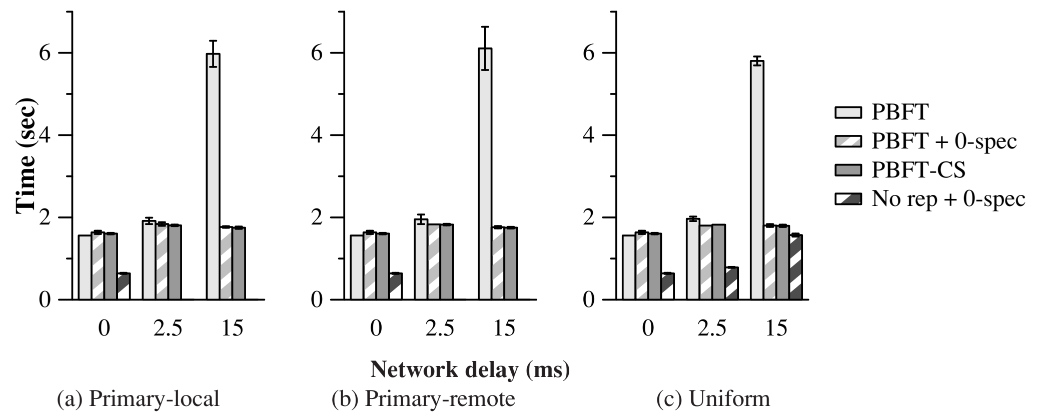


Figure 6: **Write-only** NFS micro-benchmark.

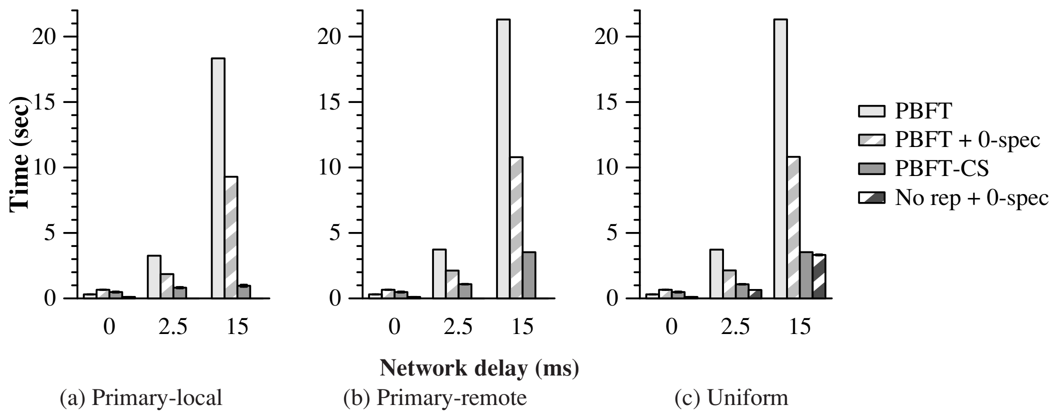


Figure 7: **Read/Write** NFS micro-benchmark.

against the variation of NFS that uses 0-reply speculation. All benchmarks begin with a freshly-mounted file system and an empty cache.

Unlike the counter service, this application has overhead associated with creating, committing, and rolling back to a checkpoint. Processes may have computation to perform between requests, and they may need to block before an output commit.

For comparison with non-speculative systems, we measure the performance of NFS under PBFT. Using our speculative NFS protocol, we measure PBFT using only 0-reply speculation (*PBFT + 0-spec*) and PBFT-CS. The difference between these two measurements show the benefit of 1-reply speculation. As a lower bound, we also measure the performance of a non-replicated NFS server that uses 0-reply speculation (*No rep + 0-spec*).

We use a vanilla kernel for evaluating non-speculative PBFT with a slight modification that increases the number of concurrent RPC requests allowed. Other benchmarks use the Speculator kernel.

In the *no replication* configuration, the NFS client uses a thin UDP relay on the local machine that stands in for the BFT relay.

Our modifications to the NFS client, the relay, and the replicated service have introduced additional overhead that is not present in the original PBFT. This inefficiency is particularly apparent in our 0 ms topologies, where PBFT-CS shows a 1.03–2.18× slowdown relative to PBFT across all our benchmarks. However, in all cases at higher latencies, client speculation results in a clear improvement, and we primarily address these configurations in the following sections.

At the time of publication, we had not yet ported our NFS server to use the Zyzzyva protocol, so we regretfully are unable to provide a direct comparison for these benchmarks.

All graphs show the mean of at least five measurements. Error bars are shown when the 95% confidence interval is above 1% of the mean value.

6.5 NFS: Read-only micro-benchmark

We first ran a read-only micro-benchmark that `greps` for a common string within the Linux headers. The total size of the searched files is about 9.1 MB. Most requests

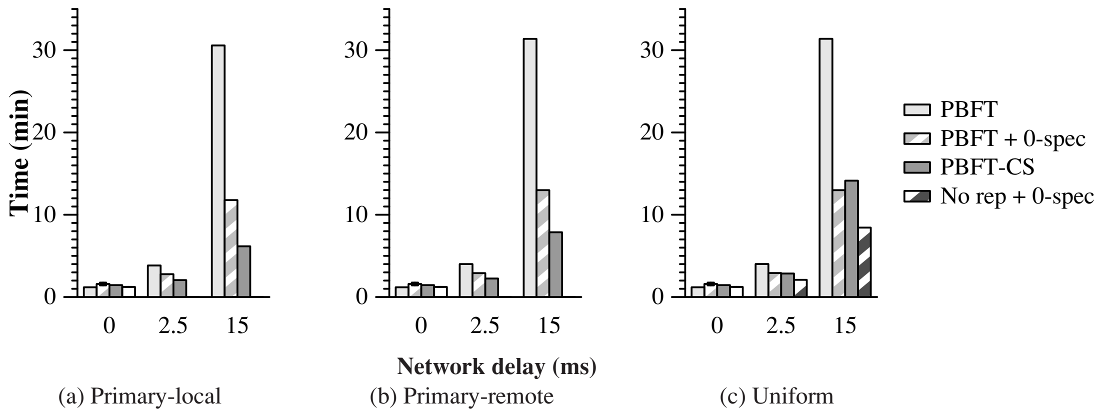


Figure 8: The **Apache build** NFS benchmark measures how long it takes to compile and link Apache 2.0.48.

in this benchmark are read-only and are optimized to circumvent agreement.

Figure 5 shows that PBFT takes 2.06× longer to complete than PBFT-CS at 15 ms. 0-reply speculation lets the client avoid blocking when revalidating a file after opening it. With PBFT-CS, we can additionally read from a file without delay: a nearby replica supplies all the speculative data. Without a nearby replica (in uniform topology), 1-reply speculation is not beneficial since optimized reads complete at about the same time the client gets its first reply.

6.6 NFS: Write-only micro-benchmark

We next ran a write-only micro-benchmark that writes 3.9 MB into an NFS file (Figure 6). All writes are issued asynchronously by the file system, and the client only blocks when the file is closed. In this case, speculation is not needed to increase the parallelism of the system.

There are a very small number of read requests in this benchmark, issued when first opening a file, so there is no practical opportunity to use 1-reply speculation. Speculation at 2.5 ms reduces the benchmark run time by only 6–7%. We found that within each latency (irrespective of topology), there is no statistical difference between PBFT+0-spec and PBFT-CS.

6.7 NFS: Read/write micro-benchmark

We next ran a read/write micro-benchmark that creates 100 4 KB files in a directory. For each file, the client creates and writes to a file; this includes read-only operations to read the directory entries. PBFT-CS never blocks on any of these operations.

In the primary-local topology, PBFT takes up to 19× longer to complete than PBFT-CS (Figure 7). Furthermore, PBFT-CS shows a resilience to changes in latency as it increases from 0-15 ms: PBFT-CS execution time doubles while PBFT takes 59× longer. On the primary-remote and uniform topologies, operations take longer to complete, but client speculation still speeds up run time by 6.03×.

6.8 NFS: Apache build macro-benchmark

Finally, we ran a benchmark that compiles and links Apache 2.0.48. This emulates the standard Andrew-style benchmark that has been widely used in the PBFT literature. This is intended to model a realistic and common workload, where speculation allows significant computation to be overlapped with I/O.

Within the primary-local topology, PBFT takes up to 5.0× longer to complete than PBFT-CS (Figure 8). In the uniform topology, PBFT takes up to 2.2× longer than PBFT-CS. Since files are often reused many times during the build process, there is less opportunity to benefit from 1-reply speculation. However, the relative difference in performance degradation as latency increases is still significant. With a co-located primary, PBFT-CS becomes 4.3× slower as delay increases to 15 ms, while PBFT slows down by a factor of 25.

6.9 Cost of failure / faulty primary

To measure the cost of speculation failures, we modified our PBFT-CS relay to inject faulty digests into early replies, simulating a primary that returns corrupted replies at a rate of 1%. Any speculation based on a corrupted reply will eventually be rolled back, and any dependent requests will be turned into no-ops on good replicas.

The results of this experiment are presented in Figure 9. We used the Apache build benchmark in the primary-local topology. The injected faults were responsible for slowdowns in PBFT-CS of 3%, 9%, and 29% at 0 ms, 2.5 ms, and 15 ms delay respectively.

These slowdowns are not identical because a client may have a greater number of requests in the pipeline for completion at a 15 ms delay than at a 0 ms delay. When one request fails, nearly all outstanding requests

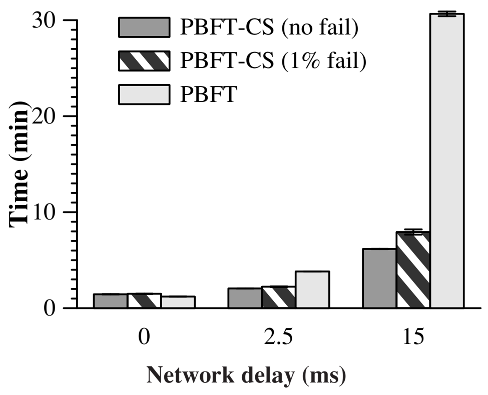


Figure 9: For the Apache build benchmark in the primary-local topology, PBFT-CS is at worst 29% slower when 1% of its speculations fail.

also fail. We observed that 1% of our speculations failed directly, and an additional 1%, 4%, and 5% of speculations (at 0 ms, 2.5 ms, and 15 ms respectively) failed due to their dependencies. These extra requests added unnecessary load to the replicas. By executing more requests in advance, clients must roll back a larger amount of state.

As discussed in section 3.3, once a client detects that 1% of requests are failing, it can stop trusting the primary to provide good first replies and disable its own speculation. If replies are signed, each primary can cause only a single failed speculation, and the resulting view change will dominate recovery time. For reference, over 100 failed speculations in this benchmark result from a 1% failure rate.

7 Related work

This paper contributes the first detailed design for applying client speculative execution to replicated state machine protocols. It also provides the first design and implementation that uses client speculation to hide latency in PBFT [8].

Speculator [28] was originally used to hide latency in distributed file systems, and thus our work shares many of Speculator's original goals. Speculator's distributed file system application assumes the existence of a central file server that always knows ground truth. No such entity exists in a replicated state machine. For instance, non-faulty replicas may disagree about the ordering of read-only requests as discussed in Section 3.2.2. Prior to this paper, Speculator was only used to speculate on zero replies. The possibility of also speculating on a single reply opens up several potential protocol optimizations that we have explored, including the possibility of generating early replies and optimizing agreement protocols for throughput.

Speculative execution is a general computer science concept that has been successfully applied in hardware architecture [15, 17, 35], distributed simulations [19], file I/O [10, 16], configuration management [36], deadlock detection [26], parallelizing security checks [29], transaction processing [20] and surviving software failures [12, 31]. This work contributes by applying speculation to another domain, replicated state machines.

There has also been extensive prior work in the development of replicated state machines, both in the fail-stop [24, 30, 34] and Byzantine [1, 8, 11, 21, 22, 32, 37] failure models. While Byzantine fault tolerance in particular has been an area of active research, it has seen relatively limited deployment due to its perceived complexity and performance limitations.

Our client-side speculation techniques apply equally well to reducing latency in both fail-stop and Byzantine fault tolerance protocols. However, they are particularly useful for protocols that tolerate Byzantine faults due to the higher latencies of such protocols.

PBFT [8] provides a canonical example of a Byzantine fault-tolerant replicated state machine, using multiple phases of replica-to-replica agreement to order each operation. Several systems since PBFT have aimed to reduce the latency in ordering client operations, typically by optimizing for the no-failure case [22] or for workloads with few concurrent writes [1, 11].

Byzantine quorum state machine replication protocols such as Q/U [1] build upon earlier work in Byzantine quorum agreement [3, 4, 13, 27], and provide lower latency in the optimal case. Q/U is able to respond to write requests in a single phase, provided that there are no write operations by other clients that modify the service state; inconsistent state caused by other clients requires a costly repair protocol. HQ [11] aimed to reduce the cost of repair, and reduces the number of replicas required in a Byzantine Quorum system from $5f+1$ to $3f+1$, but it introduces an additional phase to the optimized protocol.

Agreement protocols that use a primary replica are able to batch multiple requests into a single agreement operation, greatly reducing the overhead of the protocol and increasing throughput. While our protocol applies to both quorum and agreement protocols, the higher throughput offered by batched agreement, along with resilience during concurrent write workloads, makes them a better match for our techniques.

Our work on client speculation complements the server-side use of speculation in Zyzzyva [22]. In Zyzzyva, *replicas* execute operations speculatively based on an ordering provided by the primary, while in our system *clients* speculate based on an early response from the primary (or on 0 replies), with replicas executing only committed operations. These two approaches are complementary. Client speculation allows a client to issue a subsequent operation after only a single phase of communication with the primary, which is especially helpful for geographically dispersed deployments where some replicas are far from the client. Server speculation speeds up how fast replicas can supply a consensus response to the client, which would allow clients in our system to commit speculations faster. While we have evaluated client speculation on the PBFT protocol, it would apply equally well to Zyzzyva, where the client can receive early speculative *and* consensus responses, in the absence of failures.

8 Conclusions and future work

Replicated state machines are an important and widely-studied methodology for tolerating a wide range of faults. Unfortunately, while replicas should be distributed geographically for maximum fault tolerance, current replicated state machine protocols tend to magnify the effects of the long network latencies associated with geographic distribution. In this paper, we have shown how to use speculative execution at clients of a replicated service to reduce the impact of network and protocol latency. We outlined a general approach to using client speculation with replicated services, then implemented a detailed case study that applies our approach to a standard fault tolerant protocol (PBFT).

In the future, we hope to apply client speculation to a wider range of protocols and services. For example, adding client speculation to a protocol that uses server speculation [22] should allow clients to commit speculations faster. It may also be possible to apply client speculation to protocols that use more complex replication schemes, such as erasure encoding [18], although clients of such protocols may require more than one reply to predict the final response with high probability.

Acknowledgments

We would like to thank Miguel Castro, our shepherd Steven D. Gribble, and all anonymous reviewers for their valuable comments and suggestions. We would also like to thank Ramakrishna Kotla and Mike Dahlin for providing us with their implementation of Zyzzyva. The work has been supported by the National Science Foundation under awards CNS-0346686, CNS-0428107, CNS-0509093, and CNS-0614985. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF, the University of Michigan, or the U.S. government.

References

[1] ABD-EL-MALEK, M., GANGER, G. R., GOODSON, G. R., REITER, M. K., AND WYLIE, J. J. Fault-Scalable Byzantine Fault-Tolerant Services. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 59–74.

[2] AVIZIENIS, A. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering SE-11*, 12 (December 1985), 1491–1501.

[3] BEN-OR, M. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)* (New York, NY, USA, 1983), ACM, pp. 27–30.

[4] BRACHA, G., AND TOUEG, S. Resilient consensus protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing (PODC '83)* (New York, NY, USA, 1983), ACM, pp. 12–26.

[5] BRACHA, G., AND TOUEG, S. Asynchronous consensus and broadcast protocols. *J. ACM 32*, 4 (1985), 824–840.

[6] CARSON, M., AND SANTAY, D. NIST Net – A Linux-based Network Emulation Tool. *ACM SIGCOMM Computer Communication Review 33*, 3 (June 2003), 111–126.

[7] CASTRO, M. Practical Byzantine Fault Tolerance. Tech. Rep. MIT-LCS-TR-817, MIT, Jan 2001.

[8] CASTRO, M., AND LISKOV, B. Practical Byzantine Fault Tolerance. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation* (February 1999), pp. 173–186.

[9] CASTRO, M., AND LISKOV, B. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 2000 Symposium on Operating Systems Design and Implementation* (October 2000), pp. 19–33.

[10] CHANG, F., AND GIBSON, G. A. Automatic I/O hint generation through speculative execution. In *Proceedings of the 1999 Symposium on Operating Systems Design and Implementation* (February 1999), pp. 1–14.

[11] COWLING, J., MYERS, D., LISKOV, B., RODRIGUES, R., AND SHRIRA, L. HQ Replication: A Hybrid Quorum Protocol for Byzantine Fault Tolerance. In *Proceedings of the 2006 Symposium on Operating Systems Design and Implementation* (November 2006), pp. 177–190.

[12] CULLY, B., LEFEBVRE, G., MEYER, D., FEELEY, M., AND HUTCHINSON, N. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI '08)* (2008).

[13] DWORK, C., LYNCH, N., AND STOCKMEYER, L. Consensus in the presence of partial synchrony. *J. ACM 35*, 2 (1988), 288–323.

[14] ELNOZAHY, E. N., ALVISI, L., WANG, Y.-M., AND JOHNSON, D. B. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys 34*, 3 (September 2002), 375–408.

[15] FRANKLIN, M., AND SOHI, G. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers 45*, 5 (May 1996), 552–571.

[16] Fraser, K., and Chang, F. Operating system I/O speculation: How two invocations are faster than one. In *Proceedings of the 2003 USENIX Technical Conference* (June 2003), pp. 325–338.

[17] Hammond, L., Willey, M., and Olukotun, K. Data Speculation Support for a Chip Multiprocessor. In *Proceedings of the 1998 International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1998), pp. 58–69.

[18] Hendricks, J., Ganger, G. R., and Reiter, M. K. Low-Overhead Byzantine Fault-Tolerant Storage. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 73–86.

[19] Jefferson, D., Beckman, B., Wieland, F., Blume, L., DiLoreto, M., Hontalas, P., Laroche, P., Sturdevant, K., Tupman, J., Warren, V., Wedel, J., Younger, H., and Bellenot, S. Distributed Simulation and the Time Warp Operating System. In *Proceedings of the 1987 Symposium on Operating Systems Principles* (November 1987), pp. 77–93.

[20] Kemme, B., Pedone, F., Alonso, G., and Schiper, A. e. Processing transactions over optimistic atomic broadcast protocols. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems* (Washington, DC, USA, 1999), IEEE Computer Society, p. 424.

[21] Kihlstrom, K. P., Moser, L. E., and Melliar-Smith, P. M. The SecureRing protocols for securing group communication. In *Proceedings of the 1998 Hawaii International Conference on System Sciences* (1998), vol. 3, pp. 317–326.

[22] Kotla, R., Alvisi, L., Dahlin, M., Clement, A., and Wong, E. Zyzzyva: Speculative Byzantine Fault Tolerance. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 45–58.

[23] Kotla, R., and Dahlin, M. High throughput byzantine fault tolerance. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks* (Washington, DC, USA, 2004), IEEE Computer Society, p. 575.

[24] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978), 558–565.

[25] Lamport, L. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May 1998), 133–169.

[26] Li, T., Ellis, C. S., Lebeck, A. R., and Sorin, D. J. Pulse: A dynamic deadlock detection mechanism using speculative execution. In *Proceedings of the 2005 USENIX Technical Conference* (April 2005), pp. 31–44.

[27] Malkhi, D., and Reiter, M. Byzantine Quorum Systems. *Distributed Computing 11*, 4 (1998), 203–213.

[28] Nightingale, E. B., Chen, P. M., and Flinn, J. Speculative execution in a distributed file system. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 191–205.

[29] Nightingale, E. B., Peek, D., Chen, P. M., and Flinn, J. Parallelizing security checks on commodity hardware. In *Proceedings of the 2008 International Conference on Architectural Support for Programming Languages and Operating Systems* (March 2008), pp. 308–318.

[30] Oki, B., and Liskov, B. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing* (1988), pp. 8–17.

[31] Qin, F., Tucek, J., Sundaresan, J., and Zhou, Y. Rx: Treating bugs as allergies–a safe method to survive software failure. In *Proceedings of the 2005 Symposium on Operating Systems Principles* (October 2005), pp. 235–248.

[32] Reiter, M. K. The rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, vol. 938. Springer-Verlag, Berlin Germany, 1995, pp. 99–110.

[33] Rodrigues, R., Castro, M., and Liskov, B. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 2001 Symposium on Operating Systems Principles* (October 2001), pp. 236–269.

[34] Schneider, F. B. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22*, 4 (December 1990), 299–319.

[35] Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. A scalable approach to thread-level speculation. In *Proceedings of the 2000 International Symposium on Computer Architecture* (June 2000), pp. 1–24.

[36] Su, Y.-Y., Attariyan, M., and Flinn, J. AutoBash: improving configuration management with operating system causality analysis. In *Proceedings of the 2007 Symposium on Operating Systems Principles* (October 2007), pp. 237–250.

[37] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L., and Dahlin, M. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 2003 Symposium on Operating Systems Principles* (October 2003), pp. 253–267.

Cimbiosys: A Platform for Content-based Partial Replication

Venugopalan Ramasubramanian[1], Thomas L. Rodeheffer[1], Douglas B. Terry[1], Meg Walraed-Sullivan[2], Ted Wobber[1], Catherine C. Marshall[1], Amin Vahdat[2]

[1]*Microsoft Research, Silicon Valley* [2]*University of California, San Diego*

Abstract

ncreasingly people manage and share information cross a wide variety of computing devices from cell hones to Internet services. Selective replication of ontent is essential because devices, especially portable nes, have limited resources for storage and communica-on. *Cimbiosys* is a novel replication platform that per-its each device to define its own content-based filtering riteria and to share updates directly with other devices.

In the face of fluid network connectivity, redefinable ontent filters, and changing content, Cimbiosys ensures vo properties not achieved by previous systems. First, very device eventually stores exactly those items whose test version matches its filter. Second, every device epresents its replication-specific metadata in a compact orm, with state proportional to the number of devices ather than the number of items. Such compact repre-entation results in low data synchronization overhead, hich permits ad hoc replication between newly encoun-ered devices and frequent replication between estab-shed partners, even over low bandwidth wireless net-orks.

Introduction

elivering information that is relevant to different eople—or is appropriate for different devices—requires ystem support for a richer notion of data synchroniza-on, one that incorporates personalized content filtering. many social and work settings, where bandwidth, stor-ge, and human attention may be at a premium, filtering nables information to spread according to interests and equirements. Personal information needs do not always dhere to the rigid organizational structures imposed by ata providers [3], but rather can often be characterized y flexible query-like predicates over the contents of di-erse data collections.

At the same time, timely and robust information shar-ng cannot always rely on established Internet connectiv-y or depend on centrally managed storage. Communi-ation between devices may be ad hoc, taking advantage f the proximity of neighboring devices and the avail-bility of particular content. For example, in the wake of urricane Katrina, disaster workers needed to quickly set p ad hoc networks in which communication and control ere distributed and egalitarian [5].

In this paper, we present *Cimbiosys*, a replicated storage platform designed to support collaboration within loosely-organized communities with applications such as home media management and shared calendars and to facilitate the interplay between mobile devices and cloud-based services. The main contribution of this work is demonstrating how to permit content-based partial replication among peers while providing two important system properties:

- *Eventual filter consistency*: Each device eventually stores precisely those items that would be returned by running its custom filter query against the full data collection.
- *Eventual knowledge singularity*: The state that is transmitted between devices in synchronization requests and is used to identify unknown latest versions converges to a size that is proportional to the number of replicas in the system rather than the number of stored items.

Eventual consistency has long been demanded by applications and provided in replicated systems. Ensuring eventual filter consistency in a system that permits peer-to-peer synchronization between devices with individual, content-based filters is more challenging. Not only may a device's interest in specific items fluctuate over time as the items are updated, but a device may vary its filtering criteria, causing items with stable contents to enter and leave the device's interest set. The next section expands on the substantial challenges of content-based partial replication.

Eventual knowledge singularity is a new property we have defined to convey the importance of compact synchronization-specific state in making economical use of bandwidth and system resources. Essentially, eventual filter consistency is an important correctness property while knowledge singularity is hidden from applications but provides performance and convergence benefits. In particular, this property allows Cimbiosys to use brief intervals of connectivity between peer devices and permits more frequent exchanges between regular synchronization partners, thereby reducing convergence delays. By contrast, conventional synchronization techniques that exchange per-item version vectors or rely on

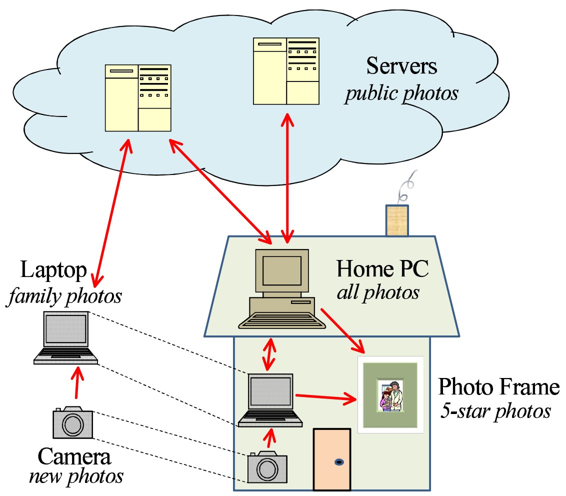


Figure 1: Photo sharing

operation logs make less effective use of relatively slow or intermittent connections. In such systems, the data exchanged during synchronization is roughly proportional to the collection size or dependent on the update rate; this limitation becomes important as collection sizes grow into the tens of thousands of items and items are updated repeatedly.

2 Challenges

To further illustrate the needs of applications that manage partially replicated data, consider the photo sharing scenario depicted in Figure 1. Alice is traveling in Thailand, photoblogging as she goes. Each night, the day's photos are copied from Alice's camera to her laptop. When she reaches a town with an Internet café, she uploads select photos to her Flickr account. After Alice returns from her trip, her photos are synchronized with the master collection on her PC. She spends several weeks working with her new photos on the PC, rating them using one to five stars, adding additional tags, and cropping or retouching photos. Five-star photos are uploaded via a direct WiFi connection to her living room's photo frame. Photos that Alice tags "public" are uploaded to a travel photoset on Flickr and onto a photojournalism web site. A copy of all of her family photos are retained on her laptop, so she'll have them with her when she travels again.

This scenario reveals an implicit set of requirements for a modern storage platform:

- Updates may originate from multiple sites and produce new versions of items that must be selectively disseminated to various devices.
- Interdevice communication may be ad hoc, taking advantage of device proximity and the availability of particular content.
- Not all devices (or even cloud-resident services) store complete collections, and the items of interest vary across devices according to their uses and capabilities.

At first blush, adding content-based filtering to a replication protocol may seem straightforward. Start with any protocol that fully replicates data and guarantees eventual consistency. Whenever a data item is about to be sent via this protocol, check the contents of the item against the destination device's filter. If the item matches, and hence is of interest to the destination, then continue to send the item; if it does not match, then ignore the item. Unfortunately, this simple scheme does not ensure eventual filter consistency.

Content-based filtering for devices with arbitrary communication topologies introduces five key challenges:

- *effective connectivity*: ensuring, in the face of varying device-specific filters, that every item has a path by which it can flow to all interested parties;
- *partial synchronization*: permitting incremental synchronization between peers with overlapping interests without wasting bandwidth on duplicate items or excessive exchanges of metadata;
- *item move-outs*: informing devices of items they store that no longer match their filters due to more recent updates;
- *out-of-filter updates*: determining how to propagate and when to safely discard updated items that do not match the updating device's own filter; and
- *filter changes*: allowing a device to modify its filter without completely discarding previously stored items or failing to receive items that match its new filter.

Unless these issues are explicitly addressed by the replication protocol design, they can prevent eventual filter consistency. We now describe each of these problems in more detail; solutions are presented in later sections.

A synchronization *topology* can be viewed as a graph where devices (or services) are the nodes, and edges indicate synchronization partnerships between pairs. Custom synchronization topologies that permit indirect communication between devices are desirable; Alice's photo frame never directly synchronizes with her camera, for instance. In a fully replicated system, eventual consistency can be achieved as long as the topology graph is connected and devices at least occasionally synchronize with their neighbors. As long as these basic conditions are met, *topology-independent* protocols accomodate arbitrary communication patterns. In a system with partially replicated collections, additional issues arise. For example, in the scenario in Figure 1, if Alice's home PC never directly synchronized with her laptop, then the only path for routing new, tagged photos from Alice's laptop to her PC would be through services in the cloud, such as Flickr. In this case, the PC would only receive laptop-resident photos that are tagged as "public" and, hence, have been uploaded to a photo-sharing service. Section 7 discusses the topology constraints enforced by Cimbiosys to ensure effective connectivity.

The problem of *partial synchronization* arises when a device synchronizes from a partner that can only supply some of the items that match the device's filter. For example, while Alice is traveling and uploading select photos to Flickr, her home PC may synchronize daily with the service and obtain these new photos. When Alice returns home and her laptop synchronizes directly with her PC, the PC should not assume that it has already received from Flickr any photos taken more than a day ago. In general, a device may receive some items that match its filter from one synchronization partner and other items of interest from other partners. Section 4.2 introduces the notion of *item-set knowledge* to deal with this issue.

An item is said to *move out* of a device's interest set when an update to the item causes it to no longer match the device's filter. For example, suppose that Alice decides that one of her public photos is a bit too revealing, and so she edits the photo on her PC to remove the "public" tag. Using the simple replication approach outlined earlier, this updated photo would not be sent to Flickr when it next synchronizes with Alice's PC. However, the previous version of this photo, the one marked as public, would remain indefinitely on Flickr's web site, contrary to Alice's intentions (and violating eventual filter consistency). Replication protocols that support content-based filtering not only must selectively propagate updated items but also must inform devices of items that should be discarded. Section 5.1 indicates the conditions under which Cimbiosys delivers move-out notifications during synchronization.

The fourth challenge is dealing with *out-of-filter updates*. A device might update an item producing a version that does not match the device's own filter. For example, suppose that Alice is working on her laptop and edits one of her private photos to remove the "family" tag (perhaps a photo of her sister's ex-husband). In this case, Alice's laptop cannot discard the photo immediately, even though it does not match the laptop's filter, since doing so would prevent other devices from learning of this edit; the photo can only be discarded by the laptop after it synchronizes with the home PC and sends it the new version. In some situations, none of a device's regular synchronization partners may be interested in out-of-filter updates that it makes. Section 5.2 addresses this issue.

A final challenge arises from the need to support changing filters. A person's information needs may vary over time, causing her to change some devices' filters. For example, Alice might decide one day that she wants only 5-star photos uploaded to the photojournalism web site rather than all of her public photos. One option is for a device, upon a change to its filter, to discard all of its locally stored items, reset its synchronization state, and essentially restart as a new replica. However, this approach wastes critical resources, such as network bandwidth and energy, and may disrupt the person's work. Section 5.3 details our approach to filter changes.

3 Cimbiosys Platform

Cimbiosys is a platform developed to support a variety of applications that manage data on mobile devices, personal computers, and cloud-based services. It was developed as part of a research project exploring issues in community information management (CIM).

3.1 System model

In the Cimbiosys distributed architecture, each participating node, hereafter simply called a *device*, stores full or partial copies of one or more data collections. A *collection*, for instance, might be an individual's digital photo album, a family's calendar, a shared video library, or a company's customer database. Each collection is managed separately and consists of a set of items that are not shared with other collections.

An *item* is an XML object plus an optional associated file. For example, a photo item stores its JPEG data in a conventional file and the associated XML object holds descriptive information, such as when the photo was taken, its resolution, a quality rating, and human-supplied keywords.

A *replica* contains copies of some or all of the items in a given collection. A device can hold any number of replicas of different collections. For simplicity, all of the examples used in this paper involve a single collection and a single replica per device.

Each device sharing a collection maintains its own replica of the items of interest. The set of items included in a device's replica is specified by a *filter*, which is a selection predicate over the items' XML contents. For example, a filter might select e-mail messages from a particular individual, files tagged with certain keywords, or photos with a 5-star rating. The default "*" filter indicates that the device is interested in all items, and hence stores a full replica of the collection. Users can set different filters for each device and can change these filters over time.

Each device is allowed to read its locally stored items and update those items at any time, as long as such updates are in accordance with the collection's *access control policy*. Update operations are applied directly to items in the device's local replica; such operations are not logged or explicitly recorded. Updates produce new *versions* of items that are later sent to other replicas via a device-to-device *synchronization protocol*. Devices generally have regular synchronization partners but may also synchronize with any replica that they encounter.

A device can join the system simply by creating a new (empty) replica of some collection and then synchronizing with some existing replica(s). Collections and their replicas can be discovered by a variety of means, including social networking web sites, e-mail invitations, naming directories, and wireless discovery protocols.

A replica may remain disconnected from the rest of the system for an arbitrary amount of time due to device failures or lack of network connectivity. However, we assume that each device eventually recovers with its persistent storage intact, occasionally communicates with other devices, and correctly executes the synchronization protocol. A device can permanently retire and discard its local replica but must first synchronize with some other device to ensure that updates are not lost.

At any point in time, a replica may hold older versions of items that have been updated elsewhere, and it may not have learned yet of recently created or deleted items. The Cimbiosys synchronization protocol guarantees eventual filter consistency. That is, a replica eventually receives all versions of items that match its filter and have not been overwritten by later versions, and the replica eventually discards items that are updated in such a way that their contents no longer match the replica's filter.

Cimbiosys does not provide other guarantees such as causal consistency or multi-item coherence. In particular, versions may be received by a device in a different order than they were produced. Moreover, a set of versions for items that were updated atomically at one device may be partially received by another device whose filter only matches a subset of the items.

Naturally, because Cimbiosys allows updates to be made at any replica without locking, two (or more) devices may perform concurrent updates to the same item. Such updates result in conflicting versions that are propagated throughout the system using the synchronization protocol. Any device whose filter selects both conflicting versions may detect the conflict and either resolve it automatically or store both versions pending manual resolution. Resolving a conflict produces a new version of the item that supersedes all known conflicting versions. Any existing technique for detecting conflicts, such as per-item versions vectors [16] or concise predecessor vectors [12], could be adopted for use with content-based partial replication. Thus, no further discussion of conflict management appears in this paper.

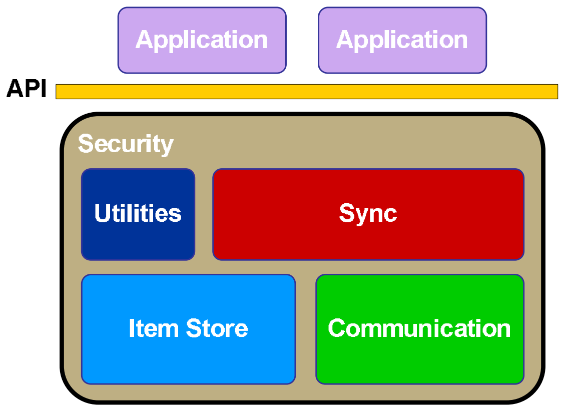


Figure 2: Cimbiosys software architecture

3.2 Software components

Each device in Cimbiosys runs the set of software modules depicted in Figure 2. The *Item Store* manages the items for local replicas of one or more collections. The file portion of each item is stored in a special directory in the device's local file system. XML objects are stored in an SQL Server (Compact Edition) database where they can be queried and updated transactionally.

The *Communication* module is responsible for transmitting data to other devices using available networks, such as the Ethernet, WiFi, cellular, or Bluetooth. It also encapsulates the transport protocol used by the Sync module. Devices are free to use a variety of transport protocols, including SOAP-based RPC, HTTP, and Microsoft's FeedSync, a set of simple extensions to RSS. Of course, any two devices must agree on the network and transport protocol that they use during synchronization.

The *Sync* module implements the synchronization protocol described in Section 4. During synchronization, it enumerates versions of items in the local Item Store that are unknown to the remote sync partner and sends these along with the appropriate metadata. The remote partner then adds the received items to its Item Store, possibly replacing older versions of these items. We are considering allowing devices to keep multiple versions if requested by an application, but our current implementation retains only the latest known version of each item.

Cimbiosys also includes a number of *Utilities* for recording information about regular synchronization partners, naming collections and devices, managing access controls, and performing other configuration functions.

Security considerations permeate the Cimbiosys design. For example, all versions of items are digitally signed by the originating device, and collection-specific policies dictate which devices are allowed to create, update, and delete items in a collection. Versions produced by a device without write access to the collection (or to the specific items) are rejected during synchronization. A full discussion of the access control design can be found in a companion paper [22]. Additionally, techniques have been developed for recovering from corrupt versions that are introduced through malice or misuse [11].

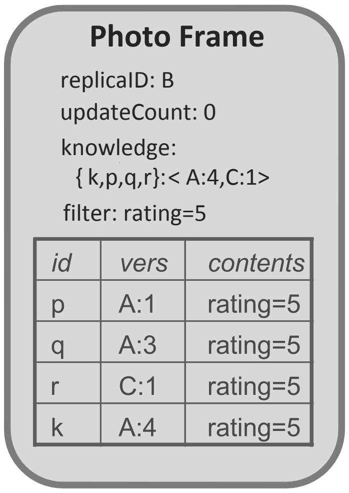
Applications interact with the Cimbiosys platform using a specially developed application programming interface (API). Through this API, an application can create a new collection, create a local replica for an existing collection, add items to a collection, update and delete items, run queries over items, initiate synchronization between a local and a remote replica, establish regular synchronization partnerships, change access permissions, and change a replica's filter. Legacy applications that read and write local files, and do not use the Cimbiosys API, are supported by "watcher" processes that monitor file system directories and import files into (or delete items from) a local replica.

3.3 Implementation and validation

Cimbiosys has been implemented in two different environments. One implementation is in C# using Microsoft's .NET Framework running on Windows. We plan to port this code to Windows Mobile 6.0 so it can run on handheld mobile devices. The other implementation is in Mace, a C++ language extension that supports distributed systems development [8]. Both implementations are used in the evaluation presented in Section 8.

Additionally, the synchronization protocol has been fully specified in TLA+ [10]. Extensive model checking has been performed on both the TLA+ specification and the Mace implementation to ensure that the protocol meets the stated design goals, that is, achieves eventual filter consistency and eventual knowledge singularity under a variety of operating conditions.

Two applications have been designed and are intended for deployment in our lab. *Cimetric*, implemented in C#, is a collaborative authoring tool. It coordinates access and updates to the complex, heterogeneous set of text, graphics, and data files created and modified in the process of writing a paper. Authors receive their own replicas of the paper, perform local updates, and make those updates visible to coauthors when they are ready to share a new version. *CimBib* is designed as a bibliographic database and personal digital library in which colleagues can share references to local and remote copies of published papers as well as personal annotations and recommendations; this application is still in a user-centered design phase. The designs of both Cimetric and CimBib were informed by a qualitative field study of scholarly writing and reference use [13].



id	vers	contents
p	A:1	rating=5
q	A:3	rating=5
r	C:1	rating=5
k	A:4	rating=5

Figure 3: Sample metadata held on the photo frame

4 CIM Sync Basics

The next three sections focus on a key aspect of the Cimbiosys platform, the synchronization protocol. The basic protocol is introduced in this section; Sections 5 and 6 address how the protocol meets the challenges of filter consistency (storing the items that currently match a replica's filter and no other items) and knowledge singularity (operating efficiently by optimizing the metadata that is exchanged during synchronization).

4.1 Metadata

The CIM Sync protocol relies on both per-item and per-replica metadata. Each collection and each item in a collection has a unique identifier, as does each replica of a collection. Each version of an item also has a unique identifier called its *version-id*. Whenever an item is created, updated, or deleted, the replica on which this operation is performed creates a new version-id for the item consisting of the replica's identifier coupled with a counter of the number of update operations that have been performed by that replica. Deleted items are simply marked as deleted; such items are treated as out-of-filter versions as discussed in Section 5.2 and are eventually discarded by all replicas.

For each item in a replica, the Cimbiosys item store maintains the item's unique identifier, version-id, XML+file contents, deleted bit, and additional information used to detect whether different versions of the item are in conflict (similar to the made-with knowledge used in WinFS [15]). Only the latest known version of each item is retained in the item store. Older versions are considered obsolete.

Figure 3 depicts the data and metadata maintained by a sample replica in our photo sharing scenario. This particular replica, the digital photo frame, is known as replica B. Note that uppercase letters are used throughout this paper as unique replica identifiers while lowercase letters are used as unique item identifiers. This replica has not performed any local updates, and hence its $updateCount$ is zero. Its filter indicates that it is interested only in photos with a 5-star rating. The replica's item store is shown as a table at the bottom of the figure. It stores four photos: items p, q, r, and k. Every item has a unique version-id. Item p, for instance, has a version-id of A:1, meaning that this version was produced by replica A's first update operation, and has a rating of 5 stars. Each item has additional data and metadata that is not shown in the figure, such as the actual photo contents and the deleted bit. Finally, this replica has knowledge about the items that it stores as described next.

4.2 Item-set knowledge

Each replica maintains *knowledge* recording the set of versions that are known to the replica. Conceptually, a replica's knowledge is simply a set of version-ids; it contains identifiers for any versions that (a) match the replica's filter and are stored in its item store, (b) are known to be obsolete, or (c) are known to not match the replica's filter. Including the third class of versions, *out-of-filter versions*, and using a novel representation called *item-set knowledge* distinguishes the knowledge used in CIM Sync from that of other replication protocols like Bayou [18] that do not support content-based partial replication.

Knowledge is represented as one or more fragments where each fragment is a version vector [16] and an associated explicit set of item ids. The version vector component indicates, for each replica that has updated any item in the collection, the latest known version-id generated by the replica. Semantically, if a replica holds a knowledge fragment S:V then the replica knows all versions of items in the set S whose version-ids are included in the version vector V. When a replica's knowledge contains multiple fragments, the replica's overall knowledge is the union of the version-ids from each fragment. Note that, from its knowledge alone, a replica cannot determine whether a known version is stored, obsolete, or out-of-filter.

For example, replica B in Figure 3 has a single knowledge fragment whose item-set is $\{k, p, q, r\}$, the ids of the four items that are stored by this replica, and a version vector of <A:4, C:1>. Replica B, the photo frame, does not appear in the version vector since it never directly updates items and hence does not generate any versions. Replica B's knowledge indicates that the device is aware of any versions of items k, p, q, or r with a version-id of A:1, A:2, A:3, A:4, or C:1. It does not mean, however, that each of these version-ids is for a current or obsolete version of one of these items. To permit a compact knowledge representation, the version vector may include version-ids for items that are not in the associated set; technically, those versions are not known to the replica. For instance, version A:2 could be the latest version of some item u that is not stored by replica B and that may or may not match its filter.

A knowledge fragment may specify "*" as the item-set, meaning that the set includes all items in the collection. Such fragments are called *star-knowledge*. In a system consisting entirely of full replicas, each replica's knowledge is always a single star-knowledge fragment. Partial replicas introduce the need for item-set knowledge in addition to star-knowledge. In a system with a mix of full and partial replicas, any replica may have both star-knowledge and any number of item-set knowledge fragments, at least temporarily. For instance, after synchronizing from a partial replica, a full replica may end up with item-set knowledge reflecting the set of received items.

4.3 Filtered synchronization

Cimbiosys uses a one-way, pull-style synchronization protocol. A replica, called the *target replica*, initiates synchronization with another replica, called the *source replica*. Each device generally plays the role of the target replica for some synchronization sessions and the source replica for others. Two-way synchronization requires a pair of devices to synchronize, switch roles, and then synchronize again.

The target replica starts by sending a SyncRequest message that includes the target's knowledge and its filter. The target is not sent any versions that are already included in its knowledge or that are not of interest. In particular, the source replica checks its item store for any items whose version-ids are not known to the target replica and whose XML contents match the target's filter. The XML contents, file contents, and metadata for each of these items are returned to the target. If possible, as discussed in Section 5.1, the source replica also informs the target replica of items that no longer match its filter. Finally, the source replica responds with a SyncComplete message including one or more knowledge fragments that are added to the target's knowledge. At the very least, this *learned knowledge* includes knowledge pertaining to items transmitted during this synchronization session but may include additional version-ids as discussed in Section 6.1.

The messages received by the target replica can be applied to its item store individually or as a single atomic transaction. Updating items (and the replica's knowledge) as new versions are received allows progress to be made even when a connection is interrupted before the synchronization protocol completes. The knowledge-driven nature of the protocol makes it resilient to device crashes and lost messages.

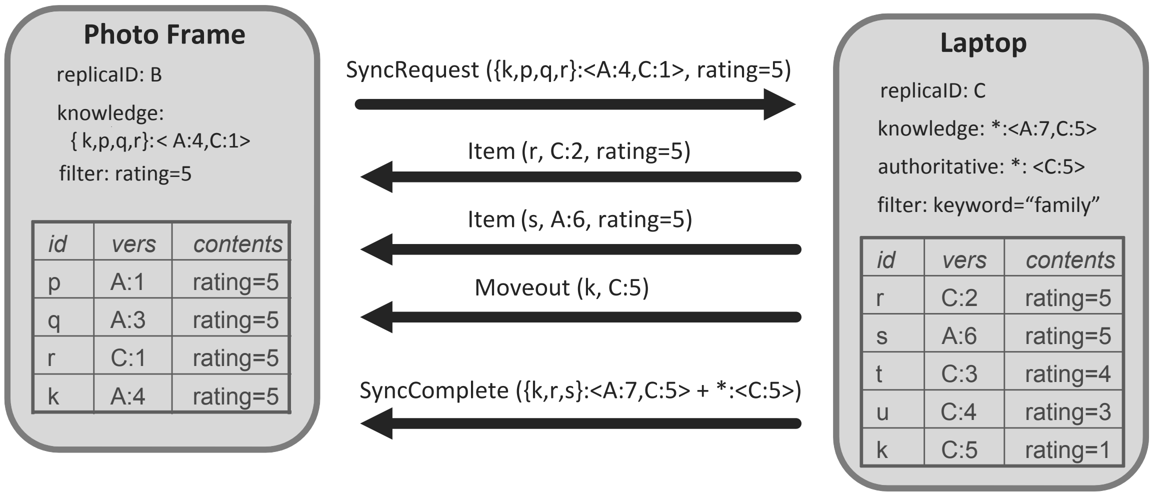


Figure 4: Example synchronization between a target replica, the photo frame, and a source replica, the laptop

Figure 4 illustrates a synchronization session from our scenario in which the digital photo frame (replica B) requests items from the laptop (replica C). The state shown for each device is the metadata and item store *before* synchronization. The arrows show the messages that are sent during synchronization. Note that the photo frame's knowledge that is sent in the SyncRequest message specifies that it knows about four items, but has not seen any updates from the laptop since version C:1. The laptop, the source replica in this example, returns a more recent version of item r that it produced and a new item s that had been created at replica A. Item k had also been updated on the laptop to reduce the photos rating; hence the laptop notifies that photo frame that this item is no longer of interest. The final message informs the photo frame of the knowledge it learned from the laptop. This learned knowledge consists of two knowledge fragments, separated by a plus sign, which means that the photo frame will end up with three knowledge fragments after processing the SyncComplete message.

The following sections describe in more detail specific protocol features devised to support the requirements of partial replication.

5 Eventual Filter Consistency

Although the use of item-set knowledge in the CIM Sync protocol guarantees that replicas eventually receive all items of interest (assuming sufficient effective connectivity), it does not ensure eventual filter consistency. This section presents additional techniques needed to deal with move-outs, out-of-filter updates, and filter changes.

5.1 Move-out notifications

During synchronization, the target replica may receive *move-out notifications* from the source replica when items have later versions that no longer match its filter. These cause the target to remove specified items from its item store. There are two conditions under which the source returns move-out notifications.

The simplest condition is when the source replica stores an item whose version is not known to the target replica and whose contents do not match the target's filter. The source can send a move-out notification for any such item. This is the condition illustrated in Figure 4 where the laptop sends a move-out notification for item k, whose rating had been reduced.

A target replica may receive move-out notifications for items that it does not store, such as items that are updated and continue to not match the target's filter, a potentially common occurrence. For example, suppose that the laptop in Figure 4 updated item t producing version C:6 in which the rating was unchanged but a new caption was added to this photo. In this case, when the photo frame next synchronizes from the laptop, it would be sent a move-out notification for item t even though it does not store this item and perhaps never did. Such spurious notifications do not affect eventual filter consistency since they will simply be ignored by the receiving replica, but they do consume network and processing resources.

To avoid spurious move-out notifications, a SyncRequest message may optionally include a set of identifiers for items that are stored by the requesting replica. The source replica only sends move-out notifications for items that are in this set. Replicas cache this item set for their regular synchronization partners, allowing these partners to send deltas, that is, to send just the set of newly acquired items.

Sending move-out notifications for items that are stored at the source replica is insufficient. Consider the case of a replicated customer relationship database in which a server holds the complete database, Bob's laptop holds items for all California customers, and his cell phone stores items for customers that live in Los Angeles. Bob's cell phone synchronizes periodically with his laptop but never directly with the server database. Suppose that a customer moves from Los Angeles to Chicago. When Bob's laptop synchronizes with the server, it receives a move-out notification causing the laptop to drop this customer from its local replica. But then how does Bob's cell phone learn that it also should discard this item?

The second condition for sending a move-out notification for an item is as follows: the target replica stores the item, the source replica does not store the item, the source replica's filter is no more restrictive than the target's filter, and the source's knowledge for this item is greater than the target's knowledge. In other words, if the source is interested in all items of interest to the target and is more knowledgeable than the target, it can deduce that any items it does not store should also be removed from the target's item store. This relies on the source being informed of the set of items that are stored by the target.

5.2 Out-of-filter updates

To preserve versions produced by out-of-filter updates, the updated items are placed in a special portion of the updating replica's item store called the *push-out store*. Items in the push-out store are not visible to applications, but are treated like any other item during synchronization. In particular, such items are sent to a synchronization partner if they match its filter, and may be overwritten by items received from a sync partner, possibly causing the item to move back into the regular item store.

Unfortunately, a replica might not have any synchronization partner whose filter matches the items in its push-out store. Thus, when synchronizing with any replica with an equal or less restrictive filter, a replica sends all items in its push-out store, and then optionally discards these items once it learns that they were successfully received by the target replica. This partner accepts these items even if they don't match its filter. Such items may end up in the target replica's push-out store, from where they are passed to another replica. However, this could lead to situations in which two replicas play "hot potato" by passing back and forth an item that matches neither of their filters. Section 7 discusses restrictions that Cimbiosys places on the synchronization topology to avoid the hot potato problem and guarantee that out-of-filter updates eventually reach all interested replicas.

5.3 Changing filters

Cimbiosys permits arbitrary filter changes while allowing replicas to retain as many items as possible. When a replica changes its filter it may need to discard items or knowledge or both depending on the nature of the filter change. If the new filter is more restrictive than the previous filter, that is, if it matches fewer items, then items that no longer match the filter are moved to the replica's push-out store. The replica cannot simply discard such items since it may be the only replica that holds the latest versions. As discussed above, items from the replica's push-out store will eventually be discarded after they are passed to another replica (or it is determined that they are already stored by another replica). Although some in-filter versions may become out-of-filter versions, the replica's knowledge does not change.

If the new filter is less restrictive than the previous filter, then previously out-of-filter versions may now match the new filter. Such versions need to be removed from the replica's knowledge so that the replica will receive them during future synchronizations. Unfortunately, the replica cannot determine which versions in its knowledge are out-of-filter and which are obsolete. So, conservatively, its knowledge must be retracted to include only versions of items that it already stores. The representation of item-set knowledge makes retraction easy. Knowledge fragments with explicit item-sets retain the same version vector but with a possibly smaller set of items; any star-knowledge fragments are converted to item-set knowledge.

If the new filter is neither less restrictive nor more restrictive than the previous filter, that is, if the old and new filters are incomparable, then both cases apply. The replica may need to move non-matching items to its push-out store. The replica also needs to retract its knowledge.

Since replicas are allowed to change their filters at any time, a replica may receive out-of-date move-out notifications based on a previous filter. To guard against processing out-of-date notifications, a replica increments a counter whenever it updates its filter. Essentially, this counter serves as a version identifier for the replica's filter. The filter version number is included in each synchronization request and is returned in each move-out notification. Move-out notifications that include old filter version numbers are simply ignored by the receiving replica.

6 Eventual Knowledge Singularity

In this section, we propose mechanisms by which replicas acquire and compact their knowledge. Although the number of fragments in a replica's knowledge may temporarily grow after synchronization, the knowledge tends to converge towards a single star-knowledge fragment represented as a single version vector. This section shows how we achieve the desired state of knowledge singularity for both full and partial replicas.

6.1 Acquiring knowledge

As replicas receive items during synchronization, they add the items' version-ids to their knowledge, but require some other means of learning about obsolete and out-of-filter versions. The SyncComplete message at the end of the synchronization protocol conveys knowledge that the target replica learned during this sync session. The target replica adds this learned knowledge to its own knowledge, generally as new knowledge fragments. This knowledge can include any version-ids for items currently stored by the source replica as well as any ids for versions that the source knows to be obsolete. It may not, however, include versions that are out-of-filter at the source replica but could match the target replica's filter as this would cause the target replica to fail to receive such versions from other replicas.

The learned knowledge, therefore, depends on the relationship between the filters of the synchronizing replicas. If the source replica's filter is no more restrictive than the target's filter, that is, if any item that matches the target's filter also matches the source's filter, then the source replica can send its complete knowledge in the SyncComplete message; any out-of-filter versions included in the source's knowledge will also be out-of-filter with respect to the target replica. In other cases in which the target has a broader filter or a disjoint filter compared to the source, the source replica must restrict the conveyed learned knowledge to those items that it actually stores. Figure 4 shows an example of disjoint filters; the photo frame's filter is based on the rating attribute and the laptop's filter is based on the value of the photo's keyword (in this case, "family").

6.2 Compacting knowledge

Whenever a replica synchronizes with another replica, it receives new knowledge fragments. To reduce the number of fragments in its knowledge and the overall size, a replica can compact its knowledge using a set of simple rules. For example, suppose the replica's knowledge includes two fragments, S_1:V_1 and S_2:V_2. If the set S_1 is a subset of set S_2 and the version vector V_2 dominates V_1 (i.e. any versions in V_1 are also included in V_2), then the fragment S_1:V_1 is redundant and can be discarded. If V_1 and V_2 are identical, then the sets S_1 and S_2 can be combined into a single knowledge fragment. Table 1 enumerates compaction rules that can be applied to any pair of knowledge fragments.

While these knowledge compaction rules are effective, they don't always lead to compact knowledge in practice. Consider the case of Alice who edits photo r on her laptop (replica C) producing a new version with version-id C:1, then edits this same photo again to produce a newer version C:2. Alice also adds keywords to photos t, u, and k, producing versions C:3, C:4, and C:5. Suppose that

$S_1{:}V_1 + S_2{:}V_2 \Rightarrow$

	$S_1 \subset S_2$	$S_1 = S_2$	$S_1 \supset S_2$	*otherwise*
$V_1 \subset V_2$	$S_2{:}V_2$	$S_2{:}V_2$	$S_2{:}V_2$ + $S_1\text{-}S_2{:}V_1$	$S_2{:}V_2$ + $S_1\text{-}S_2{:}V_1$
$V_1 = V_2$	$S_2{:}V_2$	$S_1{:}V_1$	$S_1{:}V_1$	$S_1{\cup}S_2{:}V_1$
$V_1 \supset V_2$	$S_1{:}V_1$ + $S_2\text{-}S_1{:}V_2$	$S_1{:}V_1$	$S_1{:}V_1$	$S_1{:}V_1$ + $S_2\text{-}S_1{:}V_2$
otherwise	$S_1{:}V_1{\cup}V_2$ + $S_2\text{-}S_1{:}V_2$	$S_1{:}V_1{\cup}V_2$	$S_2{:}V_1{\cup}V_2$ + $S_1\text{-}S_2{:}V_1$	$S_1{:}V_1$ + $S_2{:}V_2$

Table 1: Knowledge compaction rules

these items all match replica C's filter and are never updated by other replicas. The state of replica C on Alice's laptop is as shown in Figure 4. When Alice's home PC (replica A) synchronizes from her laptop, it will receive these items and the associated learned knowledge. The home PC's knowledge would become something similar to $*$:<A:9> + $\{k, r, t, u\}$:<A:7, C:5>. Unfortunately, this knowledge cannot be compacted. This problem is addressed in the remainder of this section.

6.3 Authoritative versions

Key to reducing the number of fragments in a replica's knowledge is the notion of authority. A replica is *authoritative* for a version of an item if it either stores the item or knows the item to be obsolete. Recall from Section 6.1 that version-ids for any stored or obsolete versions can be included in the learned knowledge acquired by a target replica at the completion of the synchronization process. The source replica, therefore, can return a learned knowledge fragment in which the item-set is "*" (i.e. all items in the collection) and the associated version vector includes identifiers for its authoritative versions. In other words, during synchronization, the target replica learns of any versions of any items for which the source replica is authoritative. Moreover, when the target replica's filter is equal to or less restrictive than the source's filter, the target replica becomes an authority for all of the source replica's authoritative versions.

In our previous example, the laptop (replica C) is authoritative for all of the versions that it produced, that is, for versions C:1 through C:5. Thus, replica C sends $*$:<C:5> as learned knowledge when synchronizing to any other replica. This knowledge fragment is merged into the receiving replica's star-knowledge, and hence does not lead to an increase in the overall number of knowledge fragments. A replica's star-knowledge grows so that it eventually dominates other knowledge fragments, which can then be discarded using the compaction rules in Table 1.

6.4 Transferring authority

One practical issue remains, namely how to transfer authority when an item is no longer of interest to the authoritative replica, whether due to out-of-filter updates or to filter changes. Such operations cause items to be placed in a replica's push-out store. The replica will cease to be authoritative for its own versions that are pushed to another replica and then discarded. Requiring a replica to store indefinitely all of the items that it creates or updates would be unreasonable. For instance, a digital camera often offloads its photos to a laptop in order to free up storage space for new photos. In practice, the system simply needs to maintain the invariant that there exists at least one replica that is authoritative for every version ever generated.

In Cimbiosys, when a replica sends the items in its push-out store to a replica with a less restrictive filter, the receiving replica becomes authoritative for these items. The sending replica can then discard such items without violating the system-wide invariant. Each replica records the version-id of the most recent version it has generated for which it is no longer authoritative. The replica then knows that it is authoritative for any versions it has produced with greater version-ids. The learned knowledge sent by a replica is a star-knowledge fragment containing the range of version-ids from the first version generated after its last push-out to its most recently generated version. A replica that has received multiple star-knowledge fragments containing overlapping or contiguous version ranges can combine these together into a single fragment.

For example, suppose Alice's laptop (replica C) changes its filter so that it no longer wants items with ratings below three. Version C:5 of item k no longer matches. After pushing this item to Alice's home PC (replica A), as well as sending the latest versions of all other items, the home PC will have learned $*$:<C:5>. At this point, the laptop discards item k and records C:5 as its last unauthoritative version. Now, suppose that Alice performs three more updates from her laptop producing versions with identifiers C:6, C:7, and C:8. During synchronization to another replica, say Alice's photo frame (replica B), the laptop will pass $*$:<C:6..C:8> as learned knowledge. When the photo frame synchronizes from the home PC, it will receive learned knowledge of $*$:<C:5> in addition to knowledge of other versions for which Alice's home PC is authoritative. The photo frame then combines the knowledge received from the laptop with that received from the home PC to get a knowledge fragment of $*$:<C:8>, which in turn is merged with its other star-knowledge.

As a replica synchronizes from other replicas, it acquires star-knowledge fragments from each of these sync partners. Such fragments are combined together into a single star-knowledge fragment that is monotonically increasing (provided the replica does not expand its filter). As long as each replica regularly synchronizes with a set of partners that collectively know about all versions in the system, each replica will converge towards singular knowledge. Clearly, a device that synchronizes directly with every other device will receive a complete set of star-knowledge. The following section describes how Cimbiosys ensures that replicas are configured in a suitable topology without requiring full interconnectivity.

7 Filter-based Tree Topologies

The CIM Sync protocol can be used by any set of replicas with arbitrary filters and arbitrary synchronization patterns. When a replica synchronizes with any other replica, it will receive all versions stored by its partner that match its filter, and it will receive whatever move-out notifications can be generated by the partner. Moreover, a replica never receives the same version from multiple synchronization partners (unless it engages in parallel synchronizations or changes its filter). But additional constraints must be placed on the synchronization topology in order to achieve eventual filter consistency and eventual knowledge singularity.

Cimbiosys forces replicas of a given collection to configure themselves into a hierarchically filtered tree topology. In particular, each replica has a single parent replica, except for the replica at the root of the tree, and a replica's filter must be at least as restrictive as that of its parent. In other words, a parent replica stores any items that are stored by any of its children. The replica at the root of the tree has a filter that matches all items; that is, it stores a full copy of the collection. This root replica is called the *reference replica* for the collection. Parent and child replicas are required to perform synchronization in both directions, at least occasionally, but may also synchronize with other replicas.

Constructing the tree is easy. When a new replica is created for a collection, it asks an existing replica to be its parent. If the filter of the requested parent is too restrictive, then the new replica walks up the existing tree until it finds a replica that can serve as its parent. At the very least, the reference replica can always serve as a parent for any replica with an arbitrary filter. If a replica wishes to retire gracefully from a collection, then this replica should notify its children so they can select a new parent. The retiring replica's parent, for instance, can serve as the new parent for its children, or, in some cases, one of the existing children can be promoted to be the parent of its siblings. A replica can change its parent at any time as long as it chooses a new parent with a suitable filter and does not violate the tree structure. For instance, a replica may be required to find a new parent when it expands its filter or its previous parent is unreachable for an extended period of time.

The tree synchronization topology provides four important benefits.

One, the synchronization topology ensures effective connectivity. That is, groups of replicas for the same collection cannot remain disconnected indefinitely, assuming periodic synchronization between parents and children. Moreover, each version of an item has a guaranteed path by which it can travel from the originating replica to any other replica whose filter matches the version. Specifically, when a new version is created, it can flow up the tree from child to parent replicas until it reaches common ancestors, including the reference replica. Any versions held by the reference replica can flow to any other replica over a path of replicas with increasingly restrictive filters.

Two, move-out notifications can be delivered by a parent to any of its children. Recall from Section 5 that move-out notifications can be sent when the source replica has a filter than is no more restrictive than the target. This is exactly the case for replicas with a parent-child relationship. Thus, the tree topology guarantees that all replicas are able to receive appropriate move-out notifications. Essentially, such notifications flow down the tree.

Three, out-of-filter versions in a replica's push-out store flow up the tree until they reach replicas that are interested in those items. During synchronization from a child replica to its parent, the child sends all of the items in its push-out store, regardless of whether they match the parent's filter. The tree topology prevents replicas from playing "hot potato" with out-of-filter versions.

Four, the tree topology ensures eventual knowledge singularity. As authoritative versions are passed up the tree, a parent replica assumes authority for any versions generated by any of its children or their descendants. Eventually, all authoritative versions arrive at the reference replica, which produces a single star-knowledge fragment containing all of these versions. This star-knowledge fragment is then passed down the tree from the reference replica to all other replicas during parent-to-child synchronizations. In the absence of further updates or filter changes, each replica's knowledge will eventually converge to that of the reference replica.

Although these benefits argue convincingly for having a tree-structured synchronization topology, extended synchronization patterns are not prevented. In Cimbiosys, a replica can choose arbitrary synchronization partners (in addition to its parent and children). The only restriction is that the overall synchronization topology must include an embedded tree with a reference replica.

All practical usage scenarios that we've envisioned meet this condition. In the photo sharing scenario presented in Section 2, Alice's home PC serves as the reference replica for her photo collection. Her laptop and digital photo frame synchronize directly with this PC, and treat it as their parent, as do the cloud-based services that contain selected photos. However, Alice's laptop might also sync with such services on occasion or sync directly with friends' laptops. Cloud-based services might replicate data among themselves for geographic scaling, unbeknownst to the reference replica. The digital camera, which only synchronizes with the laptop, uses the laptop as its parent replica. The overlaid tree topology ensures that Alice's new photos will eventually find their way into her master photo collection as well as onto other devices with selective filters.

8 Evaluation

In this section, we present an evaluation of Cimbiosys based on our two implementations, one in C# for Windows platforms and one in Mace for Linux platforms. In particular, we answer the following questions with respect to the goals of Cimbiosys:

- Does Cimbiosys achieve eventual filter consistency in the presence of move-outs, out-of-filter updates, and changing filters?
- How concise is the knowledge representation in Cimbiosys as compared to protocols with per-item knowledge, and does the reduction in knowledge size lead to more efficient synchronizations?
- What are the benefits of leveraging filter relationships between replicas, and how do non-hierarchical synchronizations affect the performance of Cimbiosys?

8.1 Experiments on the C# implementation

We performed experiments on the C# implementation by running 10 replicas on the same computer. The replicas formed a three-level hierarchy based on filter relationships with one full replica at the top, three partial replicas in the middle, and six more partial replicas at the bottom. Each replica's filter was less restrictive than the filters of any replica at a lower level.

The experimental workload had five serial phases consisting of different kinds of updates to the system. Each update consisted of a randomly chosen replica modifying the content of a randomly chosen item in its item store. Throughout the experiment, replicas synchronized with randomly chosen partners at regular intervals.

1. *insert phase*: Randomly chosen replicas inserted a total of 1000 items into their respective item stores at the start of the experiment. 600 synchronizations followed the inserts.
2. *update phase*: 1000 updates were performed, none of which triggered move-outs at any replicas. There

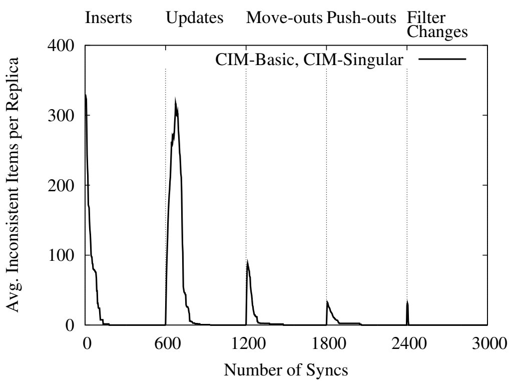


Figure 5: Average inconsistent items per replica vs. time

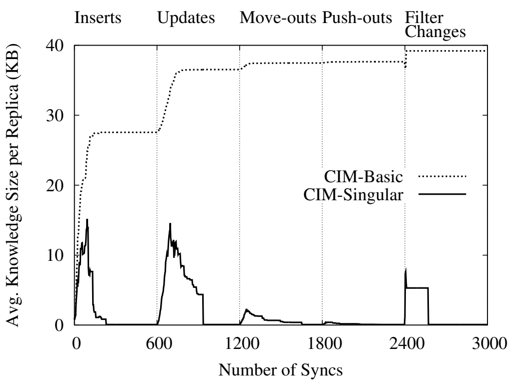


Figure 6: Average size of knowledge per replica vs. time

were 600 synchronizations in this phase, and the updates happened during the start of the phase at the rate of 10 updates between each synchronization.

3. *move-out phase*: Replicas updated 100 items; the updated content continued to match the updater's filter even though it might move out of other replicas' filters. 600 synchronizations followed.

4. *push-out phase*: Replicas performed a total of 50 out-of-filter updates. That is, the updated content did not match the updating replica's filter. Another 600 synchronizations followed.

5. *filter-change phase*: Three randomly chosen partial replicas changed their filters to new non-overlapping filters. A final 300 synchronizations ended the experiment.

We evaluated two variants of the Cimbiosys system. The first variant, called *CIM-Basic*, implemented all the core mechanisms described in Section 5 for achieving eventual filter consistency. The second variant, called *CIM-Singular*, implemented the additional mechanisms for the accumulation of authoritative knowledge in order to achieve eventual knowledge singularity as presented in Section 6.

Results

We first show the progress made by replicas in achieving eventual filter consistency. Figure 5 plots the average number of inconsistencies in a replica's item store over time. Here, an inconsistency at a replica R at a certain time includes three cases: a) an item present in R's store is obsolete, b) the latest version of an item matches R's filter but no version of the item is present in R's store, and c) an item is present in R's store but does not match R's filter. We counted these inconsistencies by tracking the global state of the system.

Figure 5 confirms that both CIM-Basic and CIM-Singular eventually achieve a state of zero inconsistencies in the presence of partial synchronization, move-outs, out-of-filter updates, and filter changes. They also converge at the same rate (and the graphs are identical) because they share the same core mechanisms to support partial replication.

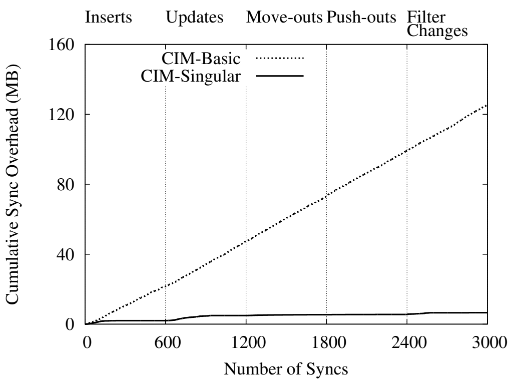


Figure 7: Cumulative synchronization overhead incurred vs. time

We next evaluate knowledge compaction in Cimbiosys. Figure 6 shows the average size of the knowledge of each replica over time. As expected, the size of knowledge in CIM-Basic increases as updates are performed and reaches a peak value dependent on the number of items stored in the replica and the number of updates performed to each item. In CIM-Singular, however, knowledge is fragmented in the initial stages but eventually converges to the size of a single version vector at the end of each phase. In other words, CIM-Singular achieves eventual knowledge singularity.

Figure 7 demonstrates the positive effect that knowledge compaction has on synchronization overhead. It shows the cumulative overhead incurred during synchronizations in the insert and the update phases. The overhead includes the cost of transmitting knowledge from the target to the source in the initial SyncRequest message and from the source to the target in the final SyncComplete message.

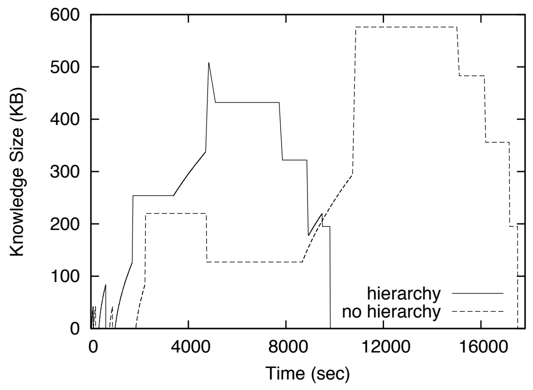


Figure 8: Effect of leveraging filter relationships

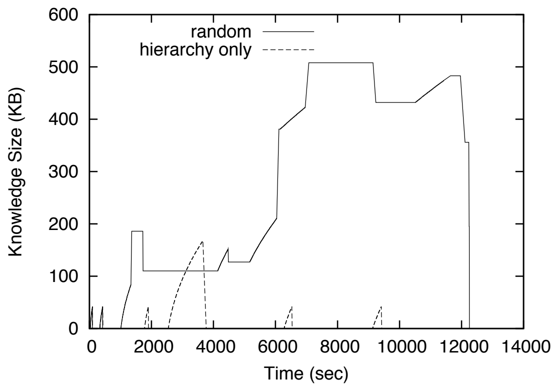


Figure 9: Effects of out-of-hierarchy synchronization

Knowledge compaction provides a significant reduction in the sync overhead over a period of time as evident from the difference between CIM-Basic and CIM-Singular in the figure. Low synchronization overhead means that replicas can synchronize more often and learn updates sooner with the same bandwidth budget. It also enables effective synchronization for replicas on bandwidth-constrained mobile devices.

8.2 Experiments on the Mace implementation

We evaluated the Mace implementation of Cimbiosys using ModelNet [21] to simulate a variety of network topologies on a cluster of machines.

For these experiments, we used a system of 10 replicas, a binary-tree filter hierarchy, and a collection size of 10,000 items, which reflects the average size of a consumer photo collection. Using ModelNet, we emulated a clique of 10 routers, each connected to a single replica. The link speed between all routers and replicas was set to 100 Mbps. The trends in the experimental results were similar with lower bandwidths.

Each experiment consisted of two phases. During phase 1, replicas created items such that 10,000 total items existed in the system at the conclusion of this phase. During phase 2, synchronizations proceeded until the knowledge at all replicas converged to a stable state.

Results

The general trends in the size of knowledge and the sync overhead for the MACE experiments were similar to the results of the C# experiments discussed earlier, and so we do not present them here. Instead, we focus on evaluating the impacts of filter relationships and synchronization patterns.

We first discuss the effects of leveraging the hierarchical filter relationships overlaid upon the network topology. We performed experiments where each replica chose a parent or a child as its synchronization partner 50% of the time and an arbitrary replica at other times. In the first experiment, called *hierarchy*, replicas would synchronize as parents or children when their filters were in the proper relation according to the filter hierarchy. In the second experiment, called *no hierarchy*, every synchronization was treated as if the filters were unrelated.

Figure 8 shows the benefits of leveraging parent-child relationships between replicas. Replicas can accept knowledge from their parents and can then directly merge this knowledge with their own, as they know after synchronizing with a parent that all versions included in the parent's knowledge should be included in their own. Similarly, replicas can become authoritative for versions authored by their descendants, and this information can flow up the hierarchy until it reaches a reference replica, at which point it flows downward in a compact form. Without a hierarchy, replicas can only claim authority over versions they themselves store. We can still achieve eventual knowledge singularity without a filter hierarchy but it takes longer for replicas to reach that state.

Finally, we discuss how the choice of synchronization partners (only parent or children versus arbitrary replicas) affects the performance of Cimbiosys. Figure 9 compares an experiment in which replicas only synchronized with their parents and children with an experiment in which the replicas selected synchronization peers at random. As the figure shows, restricting synchronizations to parents and children allows knowledge to converge much more quickly. This is because knowledge tends to flow within a hierarchy in a more compact form. On the other hand, synchronizations with arbitrary peers may allow quicker exchange of updated items between replicas at the cost of increased fragmentation in knowledge.

9 Related Work

The Cimbiosys design presented in this paper builds upon previous work on content-based filtering and especially weak-consistency replication protocols. In this section, we discuss related work with an eye toward how the systems fall short of meeting the challenges intro-

System	Selection criteria	Partial sync	Effective connectivity	Move-outs	Out-of-filter updates	Filter changes
Cimbiosys	Content-based filters	Item-set knowledge	Filter-constrained embedded tree topology	Explicit move-out notifications	Push-out store	Knowledge retraction and push-out store
Ficus	File IDs	Metadata exchange	Per-file ring topology	Cannot occur	Cannot occur	Not addressed
PRACTI	File IDs / directories	Log exchange	Policy	Not addressed	Not addressed	Not addressed
EnsemBlue	File IDs + persistent queries	Client-server	Client-server	Not addressed	Write back to server	Not addressed
Perspective	Views, i.e. attribute-based filters	Log or metadata exchange	Not addressed	Logged pre and post versions	Retain until pulled by device	Not addressed

Table 2: Key design decisions in Cimbiosys and related work.

duced by content-based replication with a peer-to-peer synchronization model, particularly in an environment characterized by changing content, user interests, and device connectivity.

The HomeViews system has the similiar goal of supporting selective data sharing in a peer-to-peer system model [6]. It allows users to export their data, including digital photos and other files, as views defined by content-based queries written in SQL. Although views are essentially equivalent to filters in Cimbiosys, they are defined by the data exporter rather than by the devices that import the data. Moreover, data is not replicated among devices but rather views are accessed remotely and searched via distributed queries.

The filters supported in Cimbiosys also resemble those of content-based publish/subscribe systems, though such systems offer a completely different replication model [1, 4]. Subscribers in a pub/sub system advertise their filters to a collection of brokers, which build routing tables used to route events from a publisher to the set of interested subscribers. Each event is independent and stored temporarily in the brokers' message queues. New subscribers (or those with new filters) observe only future events. In Cimbiosys, on the other hand, replicas eventually and persistently store all items that match their filters, can update items, and disseminate new and updated items among themselves through direct communication.

Some systems support partial replication but with a client-server model. Coda, for instance, allows clients to cache some or all of the files residing on a server, thereby supporting disconnected operation on mobile devices [9]. A hoard profile, which could be considered a type of filter, specifies the files of interest to each client, though Coda clients may cache other files based on access patterns. Clients reconcile their local changes directly with the server(s). BlueFS [14] provides a similar system model but emphasizes energy efficiency when dealing with small, mobile devices. As opposed to Cimbiosys, neither Coda nor BlueFS permits clients to share updates directly with each other.

EnsemBlue [17] extends BlueFS by allowing disconnected clients to organize into a temporary ensemble headed by a client acting in place of the server. Notably, EnsemBlue supports persistent queries that can be used by clients, along with server-provided callbacks for cache invalidation, to provide a form of content-based replication. Select operations on files that match a persistent query are logged by the server in a special file that can be retrieved and read by clients. A client then explicitly fetches new files that match its query and discards updated files that no longer match the query. Unlike Cimbiosys, the burden is placed on servers to record which files are cached where and on clients to fetch updated files in order to determine whether the contents are of interest.

Some topology-independent replication systems allow arbitrary communication patterns but lack support for content-based filters. Bayou, for instance, includes an efficient log-based, peer-to-peer synchronization protocol but assumes that all replicas are interested in all items [18]. WinFS, like Bayou, maintains a single version vector per replica that is transmitted on every synchronization, but uses state-exchange rather than log-exchange [15]. WinFS supports replication of arbitrary file folders but not per-replica filters. Cimbiosys extends the WinFS design to support content-based filtering while ensuring eventual filter consistency; the eventual knowledge singularity property ensures that the per-replica overhead converges to a single version vector as in Bayou and WinFS.

A few other systems have combined topology independence with some form of partial replication. One early peer-to-peer replication system, Ficus [7], was extended to support selective replication [19]. Each replica can store an arbitrary subset of a file system volume and can alter the set of locally stored files at any time. Because the set of interesting files is explicitly specified by file ids, and not based on file contents, several of the key concerns with content-based filtering do not arise in Ficus, including out-of-filter updates and move-outs. Synchronization is a heavy weight operation since a replica must pull information about all of the files stored on a remote replica in order to determine those that have been updated or newly created. To reduce communication costs and ensure effective connectivity, the sites replicating a given file are organized into a ring where synchronizations occur between neighbors in the ring, essentially renouncing topology-independence.

PRACTI is another replication system with topology-independence and partial replication (and arbitrary consistency) [2]. In PRACTI, each replica maintains a log of invalidations for objects that have been updated. A synchronization protocol similar to Bayou's exchanges log entries between pairs of replicas. Partial replication is achieved by allowing replicas to selectively fetch invalidated objects. Imprecise invalidations that cover a range of objects let partial replicas maintain smaller logs. While PRACTI permits each replica to define its own "interest set", the current design equates interest sets with file folders, and issues such as effective connectivity are left as policy decisions. Adding practical support for content-based filtering to PRACTI would require many of the techniques developed in Cimbiosys.

More recently, the Perspective project at CMU has been exploring a replication paradigm most closely resembling that of Cimbiosys, but with a very different system design [20]. Each device in Perspective defines an attribute-based filter called a "view". Only files included in a device's view are stored on the device. Unlike Cimbiosys, each device is aware of all other devices and their views; hence, Perspective is more suitable for a small, fixed set of devices, such as those in a consumer's home media system. Upon updating a file, a device sends a notification to all other available devices. Devices, in turn, fetch the updated files on demand. A disconnected device that misses update notifications is later brought up-to-date by synchronizing directly with other devices. A device can modify its view at any time, but it must inform the other devices and behave as a new replica during synchronization to obtain the files that match its new view. Cimbiosys, by contrast, allows content-based filters, bandwidth-efficient synchronization, incremental filter changes, incomplete knowledge of other replicas, and arbitrary synchronization partners.

Table 2 summarizes the key design decisions in previous partial replication systems as well as Cimbiosys. It focuses on the steps taken by the designers of these systems to address the five key challenges of content-based partial replication presented in Section 2.

10 Conclusion

Cimbiosys is a new storage platform that provides filtered replication of content through peer-to-peer synchronization. Its design was motivated by the needs of loosely-organized communities and of individuals managing multiple devices. Cimbiosys allows each device to express its individual information needs as a content-based filter, permits devices to enter or leave the system without global coordination, accommodates dynamically changing content and filters, efficiently propagates updated items while avoiding duplicate delivery, exploits opportunistic encounters between devices with overlapping filters, and supports flexible synchronization topologies (within certain constraints).

Eventual filter consistency, whereby a device's replica converges towards a state containing exactly those items that match its filter and nothing more, is achieved through a combination of novel technologies and pragmatic design decisions. Item-set knowledge, compactly represented as one or more version vectors and associated items, records not only the versions that have been received by a device but also obsolete versions and versions of items that no longer match its filter. Given a device's knowledge and filter, the synchronization protocol can readily determine exactly those versions of interest, thus meeting the challenge of partial synchronization. Under specific conditions, devices receive move-out notifications during synchronization and can discard out-of-filter versions without losing updates. When modifying its filter, a device can adjust its knowledge so that its local item store is incrementally updated to match its new filter.

Remarkably, knowledge converges towards a single version vector for all devices, with full or partially replicated contents. This eventual knowledge singularity property is achieved by ensuring that at least one device is authoritative for every version ever generated, transmitting star-knowledge for authoritative versions during synchronization, and compacting knowledge fragments. Our experimental evaluation, which was based on implementations of our protocol, as well as model checking performed on a formal specification, demonstrate that eventual knowledge singularity is indeed realized if updates cease for a sufficiently long period. In a system with frequent updates and filter changes, devices may never actually reach knowledge singularity, but the techniques used to drive the system in that direction serve to keep knowledge to a manageable size.

Using the CIM Sync protocol, eventual filter consistency and knowledge singularity will be attained in systems where every device synchronizes occasionally with every other device. However, requiring full inter-device connectivity is unrealistic in many of the scenarios that we wish to support. By enforcing a hierarchically filtered tree topology, Cimbiosys maintains the desired properties while providing some degree of flexibility in establishing synchronization partnerships and still allowing ad hoc communication between peers.

Acknowledgements

We thank Rama Kotla, who made considerable suggestions for improving this paper, Daniel Peek for many early design discussions, Mike Dahlin for aiding our understanding of PRACTI, Brandon Salmon for answering our questions about Perspective, Robbert van Renesse for his insightful comments on an earlier version, and our shepherd, Steve Gribble, for his advice and guidance in producing the final version of this paper. Chip Killian, James W. Anderson, and Ryan Braud provided much assistance with Mace and ModelNet. Lev Novik, Mike Clark, and Moe Khosravy were a valuable source of information on the Microsoft Sync Framework and contributed product code to our C# implementation.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–61, May 1999.

[2] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 59–72, May 2006.

[3] P. Dourish, W. K. Edwards, A. Lamarca, and M. Salisbury. Presto: An experimental architecture for fluid interactive document spaces. *ACM Transactions on Computer-Human Interaction*, 6(2):133–161, June 1999.

[4] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.

[5] S. Farnham, E. Pedersen, and R. Kirkpatrick. Observation of Katrina/Rita Groove deployment: Addressing social and communication challenges of ephemeral groups. In *Proceedings of the 3rd International ISCRAM Conference*, pages 39–49, May 2006.

[6] R. Geambasu, M. Balazinska, S. D. Gribble, and H. M. Levy. HomeViews: Peer-to-peer middleware for personal data sharing applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–246, June 2007.

[7] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page, Jr., G. J. Popek, and D. Rothmeir. Implementation of the Ficus replicated file system. In *Proceedings of the Summer USENIX Conference*, pages 63–71, June 1990.

[8] C. E. Killian, J. W. Anderson, R. Braud, R. Jhala, and A. M. Vahdat. Mace: Language support for building distributed systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 179–188, June 2007.

[9] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, Feb. 1992.

[10] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2003.

[11] P. Mahajan, R. Kotla, C. C. Marshall, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber. Effective and efficient compromise recovery for weakly consistent replication. In *Proceedings of the EuroSys 2009 Conference*, Apr. 2009.

[12] D. Malkhi and D. B. Terry. Concise version vectors in WinFS. *Distributed Computing*, 20(3):209–219, Oct. 2007.

[13] C. C. Marshall. From writing and analysis to the repository: Taking the scholars' perspective on scholarly archiving. In *Proceedings of the 8th ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)*, pages 251–260, June 2008.

[14] E. B. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the Blue file system. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 363–378, Dec. 2004.

[15] L. Novik, I. Hudis, D. B. Terry, S. Anand, V. Jhaveri, A. Shah, and Y. Wu. Peer-to-peer replication in WinFS. Technical Report MSR-TR-2006-78, Microsoft Research, June 2006.

[16] D. S. Parker, Jr., G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. A. Edwards, S. Kiser, and C. S. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[17] D. Peek and J. Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 219–232, Nov. 2006.

[18] K. Petersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*, pages 288–301, Oct. 1997.

[19] D. Ratner, P. L. Reiher, G. J. Popek, and R. G. Guy. Peer replication with selective control. In *Proceedings of the First International Conference on Mobile Data Access (MDA)*, pages 169–181, Dec. 1999.

[20] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST)*, Feb. 2009.

[21] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becke. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th USENIX Symposium on Operating System Design and Implementation (OSDI)*, pages 271–284, Dec. 2002.

[22] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Access control for peer-to-peer replication. Technical Report MSR-TR-2009-15, Microsoft Research, Feb. 2009.

RPC Chains: Efficient Client-Server Communication in Geodistributed Systems

Yee Jiun Song[1,2] Marcos K. Aguilera[1] Ramakrishna Kotla[1] Dahlia Malkhi[1]
[1] *Microsoft Research Silicon Valley* [2] *Cornell University*

Abstract

We propose the RPC chain, a simple but powerful communication primitive that allows an application to reduce the performance effects of wide-area links on enterprise and data center applications that span multiple sites. This primitive chains together multiple RPC invocations so that the computation can flow from one server to the next without involving the client every time. We demonstrate that RPC chains can significantly reduce end-to-end latency and network bandwidth in a storage application and a web application.

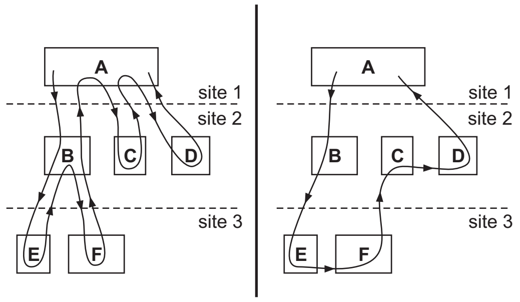


Figure 1: (**Left**) Standard RPCs. (**Right**) RPC chain.

1 Introduction

Distributed enterprise applications, such as web applications, are often built from more basic services, such as storage services, database management systems, authentication and configuration services, and services for interfacing with external components (e.g., credit card processing, banking, vendors, etc). As systems become larger, more complex, and more ubiquitous, there is a corresponding increase in the number, diversity, and geographical dispersion of the remote services that they use. For instance, Hotmail and Live Messenger share an address book service and an authentication service; there are also services specialized for each application, say, for email storage or virus scanning. These services are heterogeneous; they are often developed by different teams and are *geo-distributed*, running in different parts of the world.

Geo-distribution provides many benefits: high availability, disaster tolerance, locality, and ability to scale beyond one data center or site. However, the thin and slow links connecting different sites pose challenges, especially in an enterprise setting, where applications have strict performance requirements. For instance, web applications should ideally respond within one second [13].

The most common primitives for inter-service communication are remote procedure calls (RPC's) or RPC-like mechanisms. RPC's can impose undesirable communication patterns and overheads when a client needs to make multiple calls to servers. This is because RPC's impose communication of the form $A{-}B{-}A$ (A calls B which returns to A) even though this pattern may not be optimal. For example, in Figure 1 left, a client A in site 1 uses RPC's to consecutively call servers B, C, and D in site 2. Server B, in turn, calls servers E and F in site 3. The use of RPC's forces the execution to return to A and B multiple times, causing 10 crossings of inter-site links

We propose a simple but more general communication primitive called a *Chain of Remote Procedure Calls*, or simply *RPC chain*, which allows a client to call multiple servers in succession ($A{-}B_1{-}B_2{-}\cdots{-}A$), where the request flows from server to server without involving the client every time. The result is a much improved communication pattern, with fewer communication hops, lower end-to-end latency, and often lower bandwidth consumption. In Figure 1 right, we see how an RPC chain reduces the number of inter-site crossings to 4. The example in this figure is representative of a web mail application, where host A is a web server that retrieves a message from an email server B, then retrieves an associated calendar entry from a calendar service C, and finally retrieves relevant ads from an ad server D.

The key idea of RPC chains is to embed the chaining logic as part of the RPC call. This logic can be a generic function, constrained by some simple isolation mechanisms. RPC chains have three important features:

- *(1) Server modularity.* What made RPC's so successful is the clean decoupling of server code, which allows servers to be developed independently of each other and the client. RPC chains preserve this attribute, even allowing existing legacy RPC's to be part of a chain through simple wrappers.
- *(2) Chain composability.* If a server in the chain itself wishes to call another server, this nested call can be simply added to the chain in flux. In Figure 1, when client A starts the chain, it intends to call only servers B, C, and D. But server B wants to call servers E and F, and so it adds them to the chain.
- *(3) Chain dynamicity.* The services that a host calls need not be defined a priori; they can vary dynamically during execution. In the left figure, the fact that client A calls servers C and D need not be known before A calls server B; it can depend on the result returned by B. For example, an error condition may cause a chain to end immediately instead of continuing on to the next server.

We demonstrate RPC chains through a storage and a web application. For the storage application, we show how a storage server can be enabled to use RPC chains, and we give a simple use in which a client can copy data between servers without having to handle the data itself. This speeds up the copying and saves significant bandwidth. For the web application, we implement a simple web mail service that uses chains to reduce the overheads of an ad server.

The paper is organized as follows. We explain the setting for RPC chains in Section 2. Section 3 covers the design of RPC chains and Section 4 covers applications. We evaluate RPC chains in Section 5, and we explain their limitations in Section 6. A discussion follows in Section 7. We discuss related work in Section 8 and we conclude the paper in Section 9.

2 Setting

We consider enterprise systems that span geographically-diverse sites, where each site is a local area network. Sites are connected to each other through thinner and slower wide area links. Wide-area links can be made faster by improving the underlying network, and lots of progress has been made here, but this progress is hindered by economic barriers (e.g., legacy infrastructure), technological obstacles (e.g., switching speeds), and fundamental physical limitations (e.g., speed of light). Thus, the large discrepancy between the performance of local and wide-area links will continue.

Unlike the Internet as a whole, enterprise systems operate in a trusted environment with a single administrative domain and experience little churn. These systems may contain a wide range of services, often developed by many different teams, including general services for storage, database management, authentication, and directories, as well as application-specific services, such as email spam detection, address book management, and advertising. These services are often accessed using RPC's, which we broadly define as a mechanism in which a client sends a request to a server and the server sends back a reply. This definition includes many types of client-server interactions, such as the interactions in CORBA, COM, REST, SOAP, etc.

In enterprise environments, application developers are not malicious though some level of isolation is desirable so that a problem in one application or service does not affect others.

3 Design

We now explain the design of RPC chains, starting with the basic mechanism for chaining RPC's in Section 3.1. The code that chains successive RPC's is stored in a repository, explained in Section 3.2. In Section 3.3, we cover the state that is needed during the chain execution. We then discuss composition of chains in Section 3.4, legacy servers in Section 3.5, isolation in Section 3.6, debugging in Section 3.7, exceptions in Section 3.8, failures in Section 3.9, and chain splitting in Section 3.10.

3.1 Main mechanism

Servers provide services in the form of *service functions*, which is the general term we use for remote procedures, remote methods, or any other processing units at servers. An RPC chain calls a sequence of service functions, possibly at different servers. Service functions are connected together via *chaining functions*, which specify the next service function to execute in a chain (see Figure 2 top). Chaining functions are provided by the client and executed at the server. They can be arbitrary C# methods with the restriction that they be *stand-alone* code, that is, code which does not refer to non-local variables and functions, so that they can be compiled by themselves.

We chose this general form of chaining for two reasons. First, we want to allow the chain to unfold dynamically, so that the choice of next hop depends on what happens earlier in the chain. For example, an error at a service function could shorten a chain. Second, we wanted to support server modularity, so that services and client applications can be developed independently. Thus, a server may not produce output that is immediately ready for another server, in the way intended by the client's application. One may need to convert formats, reorder parameters, combine them, or even combine the outputs from several servers in the chain. For example, an NFS

```
// service function
object sf(object parmlist)
  // parmlist:  parameter list
// chaining function
nexthop cf(object state, object result)
  // state:  from client or earlier parts of chain
  // result:  from last preceding service function
  // returns next chain hop:
  //        (server, sf_name, parmlist,
  //        cf_name, state)
```

```
chain_id start_chain(machine_t server,
        string sf_name, object parmlist,
        string cf_name, object state)
```

Figure 2: **(Top)** Signature of a service function (sf) and chaining function (cf). **(Bottom)** Signature of function that launches an RPC chain.

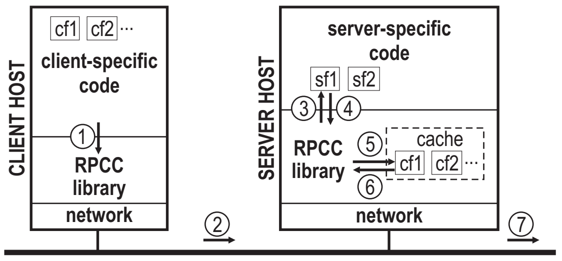


Figure 3: Execution of an RPC chain (see explanatory text in Section 3.1). RPCC stands for RPC chain.

server does not output data in the format expected by a SQL server: one needs glue that will convert the output, choose the tables, and add the appropriate SQL wrapper, according to application needs. Chaining functions provide this glue. We initially considered a simpler alternative to chaining functions, in which a client just provides a static list of servers to call, but this design does not address the issues above. We also note that it is easy to translate a static server list into the appropriate chaining functions (one could even write a programmer tool that automatically does that), so our design includes static lists as a special case.

Figure 3 shows how an RPC chain executes. (1) A client calls our RPCC (RPC chain) library, specifying a server, a reference to a service function sf_1 at that server, its parameters, and a chaining function cf_1. (2) This information is then sent to the chosen server. (3) The server executes service function sf_1, which (4) returns a result. (5) This result is passed to the chaining function cf_1, which then (6) returns the next server, service function, and chaining function, and (7) the chain continues.

For example, suppose client A wants to call service functions sf_B, sf_C, sf_D at servers B, C, and D, in this order. To do so, the client specifies a reference to sf_B and a chaining function cf_1. cf_1 causes a call to sf_C at server C with a chaining function cf_2, which in turn causes a call to sf_D at server D with a chaining function cf_3, which causes the final result to be returned to the client A.

3.2 Chaining function repository

Chaining functions are provided by clients but executed at servers. To save bandwidth, in our implementation the client does not send the actual code to the server. Rather, the client uploads the code to a repository, and sends a reference to the server; the server downloads the code from the repository and caches it for subsequent use. The repository stores chaining functions in source code format, and servers compile the code at runtime using the reflection capabilities of .NET/C# (Java has similar capabilities).

We store source code because it introduces fewer dependencies, is more robust (binary formats change more frequently), and simplifies debugging. Because the cost of runtime compilation can be significant ($\approx$50 ms, see Section 5.2.1), servers cache the compiled code, not the source code, to avoid repeated compilations.

When the chaining function is very small, it could be transmitted by the client with the RPC chain, so that the server does not have to contact the repository. Our implementation presently does not support this option.

3.3 Parameters and state

A chaining function is client logic that may depend on run-time variables, tables, or other state from the client or earlier parts of the chain. This state needs to be passed along the chain, and ideally it should be small, otherwise its transmission cost can outweigh the benefits of an RPC chain (see Section 5.2.2). We represent the state as a set of name-value pairs, which is passed as a parameter to the chaining function (see Figure 2).

The output of each service function is also passed as a parameter to the subsequent chaining function. For example, in our storage copy application (Section 4.1), the first service function reads a file, and the chaining function uses the result as input to the next service function, which writes to a file on a different server. In our email application, a service function reads an email message, and the chaining function adds the message to the state of the next chaining function, so that the message is passed along the chain back to the chain originator (a mail web server).

3.4 Nesting and composition

RPC chains can be nested: a service function in a chain may itself start a sub-chain. For example, the main chain could call a storage service, which then needs to call a

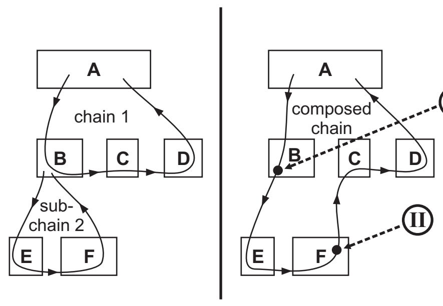


Figure 4: Composition of nested chains. **(Left)** The main chain 1 and a sub-chain 2. **(Right)** Result and manner of composing chains. (I) B starts a sub-chain, causing the RPCC library to push the $B \rightarrow C$ chaining function and its state parameter into a stack. (II) Chaining function at F returns an indication that the chain ended and the result that B is supposed to produce. This causes the RPCC library to pop from the stack, obtaining the $B \rightarrow C$ chaining function and its state parameter. It then calls this chaining function with the result and state. The chain now continues at C.

replica. We implement nesting so that a nested chain can be adjoined to an existing chain, as shown in Figure 4. Note the difference between starting a chain going from B to E, and moving to the next host in a chain going from C to D: the former occurs when the *service function* at B starts a new chain, while the latter occurs when the *chaining function* at C calls the next node in the chain. This distinction is important because the service function at B represents a native procedure at the service, while a chaining function at C represents logic coming from A. At E, the chaining function that calls F represents logic coming from B.

To compose a chain with its sub-chain, the chaining function of the parent chain needs to be invoked when a sub-chain ends (to continue the parent chain). Accordingly, when a host starts a sub-chain, the RPCC library saves the chaining function and its state parameter, and passes them along the sub-chain. The sub-chain ends when its chaining function returns null in `nexthop.server`, and a result in `nexthop.state` (this is the result that the host originating the sub-chain must produce for the parent chain). When that happens, the RPCC library calls the saved chaining function with the saved state and `nexthop.state`. Note that a chain and a sub-chain need not be aware of each other for composition.

To allow multiple levels of nesting, we use a *chain stack* that stores the saved chaining function and its state for each level of composition. The stack is popped as each sub-chain ends.

3.5 Handling legacy RPC services

RPC chains support legacy services that have standard RPC interfaces. For that, we use a simple wrapper module, installed at the legacy RPC server, which includes the RPCC library and exposes the legacy remote procedures as service functions.

Each service function passes requests and responses to and from the corresponding legacy remote procedure. Because the service function calls the legacy remote procedure locally through the RPC's standard network interface (e.g., TCP), the legacy server will see all requests as coming from the local machine, and this can affect network address-based server access control policies. (This is not a problem if access control is based on internal RPC authenticators, such as signatures or tokens, which can be passed on by the wrapper.)

One solution is to re-implement the access control mechanism at the wrapper, but this is application-specific. A better solution is for the wrapper to fake the network address of its requests and capture the remote procedure's output before it is placed on the network.

3.6 Isolation

Chaining functions are pieces of client code running at servers. Even though clients are trustworthy in the environment we consider, they are still prone to buffer overruns, crashes, and other problems. Thus, chaining functions are sandboxed to provide isolation, so that client code cannot crash or otherwise adversely affect the server on which it runs.

We need two types of isolation: (1) restricting access to sensitive functions, such as file and network I/O and privileged operating system calls, and (2) restricting excessive consumption of resources (CPU and memory).

We achieve (1) through direct support by .NET/C# of access restrictions to file I/O, system and environment variables, registry, clipboard, sockets, and other sensitive functions (Java has similar capabilities). This is accomplished by placing descriptive annotations, called *attributes*, in the source code of chaining functions when they are compiled at run-time.

We achieve (2) by monitoring CPU and memory utilization and checking that they are within preset values. The appropriate values are a matter of policy at the server, but for the short-lived type of executions that we target with RPC chains, chaining functions should consume at most a few CPU seconds and hundreds of megabytes of memory, even in the most extreme cases.

If a chaining function violates restrictions on access or resource consumption, an RPC chain exception is thrown according to the mechanism in Section 3.8.

Another way to isolate chaining functions is to use a chaining proxy (Section 7.3).

3.7 Debugging and profiling

A very useful debugging tool for traditional applications is "printf", which allows an application to display messages on the console. We provide an analogous facility for RPC chain applications: a *virtual console*, where nodes in the chain can log debugging information. The contents of the virtual console are sent with the chain, and eventually reach the client, which can then dump the contents to a real console or file. The virtual console can also be used to gather profiling information for each step in the chain and be aggregated at the client.

Even with "printf", debugging RPC chains can be hard, because it involves distributed execution over multiple machines. We can reduce this problem to the simpler problem of debugging RPC-based code by running RPC chains in a special *interactive mode*. The key observation is that chaining functions are *portable code* that can be executed at any machine. In interactive mode, chaining functions always execute at the client instead of the servers. To accomplish this, after each service function returns, the RPCC library sends its result back to the client, which then applies the chaining function to continue the chain from there. A chain executed in interactive mode looks like a series of RPC calls. By running the client in an interactive debugger, the developer can control the execution of the chain and inspect the outputs of service and chaining functions at each step.

3.8 Exceptions

An RPC chain may encounter exceptional conditions while it is executing: (1) the next server in the chain can be down, (2) the chaining function repository can be down, or (3) the state passed to the chaining function can be missing vital information due to a bug. All of these will result in an exception, either at the RPCC library in cases (1) and (2), or at a chaining function in case (3). (Service functions do not throw exceptions; they simply return an error to the caller.)

Who should handle such exceptions? One possibility is to handle them locally, by having the client send exception handling code as part of the chain. Doing this requires sending all the state that the handling code needs, which complicates the application design. Instead, we choose a less efficient but simpler alternative (since exceptions are the rare case). We simply propagate exceptions back to the client that started the chain. The client receives the exception name, its parameters, and the path of hosts that the chain has traversed thus far. (If the client crashes, the exception becomes moot and is ignored.)

In the case of nested chains, the exception propagates first to the host that started the current sub-chain. If that host does not catch the exception, it continues propagating to the host that started the parent chain, until it gets to the client. For example, in Figure 4 right, if E throws an exception (say, because it could not contact F), the exception goes to B, the node that created the sub-chain. This is a natural choice because B understands the logic of the sub-chain that it created, and so it may know how to recover from the exception. If B does not catch the exception, it is propagated to A.

3.9 Broken chains

The crash of a host while it executes an RPC chain results in a *broken chain*. In this section, we describe the broken chain detection and recovery mechanisms.

Detection. We detect a broken chain using a simple end-to-end timeout mechanism at the client called *chain heartbeats*: a chain periodically sends an alive message to the client that created it, say every 3 seconds, and the client uses a conservative timeout of 6 seconds. If there are sub-chains, only the top-level creator gets the heartbeats. Heartbeats carry a unique chain identifier, a pair consisting of the client name and a timestamp, so that the client knows to which chain it refers.

We achieve the periodic sending through a time-to-heartbeat timer, which is sent with the chain, and it is decremented by each node according to its processing time, until it reaches 0, the time to send a heartbeat. Synchronized clocks are not needed to decrement the timer; we only need clocks that run at approximately the same speed as real time. Since we do not know link delays, we assume a conservative value of 200 ms and decrement the time-to-heartbeat timer by this amount for every network hop. This assumption may be violated when if there is congestion and dropped packets, resulting in a premature timeout (false positive). However, the impact of false positives is small because of our recovery mechanism, explained next.

Recovery. To recover from a broken chain, the client simply retransmits the request. Like standard remote procedures, we make chains idempotent by including a chain-id with each chain, and briefly caching the results of service functions and chaining functions at each server. If a server sees the same chain-id, it uses the cached results for the service and chaining functions. The chain can continue in this fashion up to the host where the chain previously broke. At that host, if the "next" host is still down, an exception is thrown. Alternatively, a fail-over mechanism that calls a backup server can be implemented by using logical server names which are mapped to a backup when the primary fails. This is similar to the mechanisms used to fail over standard RPC's.

Upon a second timeout, a client executes the RPC chain in *interactive mode* (as in Section 3.7), to determine exactly at which node the chain stopped, and returns an error to the application.

3.10 Splitting chains

For performance reasons, it may be desirable to split a chain to allow parallel execution. The decision to split a chain should be made with consideration of the added complexity, as concurrent computations are always harder to understand, design, debug, and maintain compared to sequential computations. Although our applications do not use splitting chains, we now explain how such chains can be implemented.

Split. We modify chaining functions so that they can return more than one *nexthop* parameter. The RPCC library calls each nexthop concurrently, resulting in the several split-chains. Each chain has an id comprised of the id of the parent plus a counter. For example, if there is a 3-way split of chain 74, the split-chains will have ids 74.1, 74.2, and 74.3. Each of these split chains can in turn be split again, and result in split-chains with increasingly long ids. For example, if split-chain 74.1 splits into two, the resultant split-chains will have ids 74.1.1 and 74.1.2. We note for future reference that each split-chain knows how many siblings it has (this information is passed on to the split-chains when the chain splits).

Broken split chains. Recall that we use an end-to-end mechanism to handle broken chains (Section 3.9) via a chain heartbeat. When a chain splits, we also split the heartbeats: each split-chain sends its own heartbeat (with the split-chain id) and the client will be content only if it periodically sees the heartbeat from all the split-chains. The heartbeat messages indicate the number of sibling split-chains, so that the client knows how many to expect. If a split-chain is missing, the client starts the chain again (even if other split-chains are still running, this does not cause a problem because of idempotency).

Merge. To merge split-chains, a *merge host* collects the results of each split-chain and invokes a *merge function* to continue the chain. The merge host and function are chosen when the chain splits (they are returned by the chaining function causing the split). The merge host can be any host; a good choice is the next host in the chain. The merge host awaits outcomes from all split-chains before calling the merge function, which takes the vector of results and returns *nexthop*, specifying the next service function and chaining function to call.

After split-chains complete (i.e., reach the merge host), the parent chain will continue and resume its heartbeats. However, split-chains do not necessarily complete at the same time, so there may be a period from when the first split-chain completes until the parent chain resumes. During this period the merge host sends heartbeats on behalf of the completed split-chains, so that the client does not time out.

Crash garbage. When there are crashes in the system, the merge host may end up with the outcome of stale split-chains. This garbage can be discarded after a timeout: as we mentioned, RPC chains are intended for short-lived computations, so we propose a timeout of a minute. Note that if a slow system causes a running chain to be garbage collected, the client will recover after it times out.

4 Applications

To demonstrate RPC chains, we apply and evaluate them in two important enterprise applications: a storage application (Section 4.1) and a web application (Section 4.2).

4.1 Storage applications

Storage services generally provide two basic functions, *read* and *write*, based on keys, file names, object id's, or other identifiers. While this generic interface is suitable for many applications, its low-level nature sometimes forces bad data access patterns on applications. For instance, if a client wants to copy a large object from one storage server to another, the client must read the object from one server and write it to the other, causing all the data to go through the client. If the client is separated from the storage servers by a high latency or low bandwidth connection, this copying could be very slow.

One solution is to modify the storage service on a case-by-case basis for different operations and different settings. For example, the Amazon S3 storage service recently added a new copy operation to its interface [2], so that an end user can efficiently copy her data between data centers in the US and Europe, without having to transfer data through her machine. Although such application-specific interfaces can be beneficial, they are specific to particular operations and do not mitigate adverse communication patterns in other settings.

RPC chains provide a more general solution: they not only enable the direct copying of data from one server to another (through a simple chain that reads and then writes), but also enable broader uses. To demonstrate this idea, we layered RPC chains over a legacy NFS v3 storage server, as explained in Section 3.5. (We could have used other types of storage, such as an object store.) We then implemented a simple chain to copy data without passing through the client.

We also show a more sophisticated application of chains by implementing a primary-backup replication of

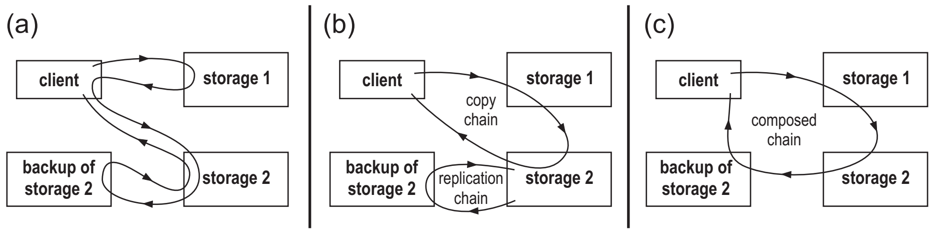


Figure 5: (a) Copying data from storage server 1 to a replicated storage server 2 without RPC chains. The client reads from storage 1 and writes to storage 2; when this happens, storage 2 writes to a backup server. (b) Using a chain to copy data and a chain to replicate data (composition disabled). (c) Composing the chains. The chains are not aware of each other but the RPCC library can combine them.

the storage server: when the primary receives a write request, it creates a chain to apply the request on a backup server. Because replication is done through chains, it can be composed with other chains. This is illustrated in Figure 5(b), which shows a setup with two storage servers, the second of which is replicated, and a user who wants to copy data from the first to the second server. Two chains are created as a result of this request: a chain that the client launches for copying, and another that the second storage server launches for replication. The RPCC library allows these two chains to be composed together, as shown in Figure 5(c). We report on quantitative benefits of our approach in Section 5.3.

4.2 Web mail application

Web applications are generally composed of multiple tiers or services: there are front-end web servers, authentication servers, application servers, and storage and database servers. Some of these tiers, namely the web servers and application servers, play the role of orchestrating other tiers, and they tend to keep very little user state of their own, other than soft session state. This is a propitious setting for RPC chains, because performance gains can be realized by optimizing the communication patterns of the various services. We demonstrate this point with a sample application.

We consider a typical web mail application. There are web servers that handle HTTP requests, authentication servers and address-book servers that are shared with other applications, email storage servers that store the users' mail, and ad servers that are responsible for displaying relevant ads. These services can be located in multiple data centers, for several reasons: (1) no single data center can host them all; (2) a service may have been developed in a particular location and so it is hosted close by; (3) for performance reasons, it may be desirable for some services to be located close to their users (e.g., users created in Asia may have their mailbox stored in Asia), though this is not always achievable (e.g., an Asian user travels to the U.S. and his mailbox is still in Asia); and (4) a service may need high availability or the ability to withstand disasters.

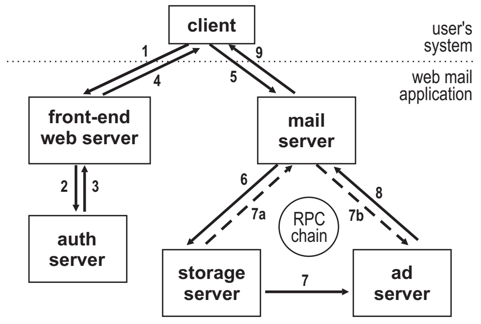


Figure 6: A simplified web mail server that uses RPC chains. The solid line shows the login sequence followed by retrieval of email and ads. The dashed line shows how a system based on standard RPC's would differ. The chain is not used for the web client, since it is outside the system. It is used in the communication between mail, storage, and ad servers.

We implemented a simple web mail service as shown in Figure 6, to study the benefits of RPC chains in such a setting. Our web mail system consists of a front-end server that authenticates users by verifying their logins and passwords. Upon successful authentication, the front-end server returns a cookie to the client along with the name of an email server. The client then uses the cookie to communicate with the email server to send and receive email messages. Upon receiving a client request, the email server first verifies the cookie, then calls the back-end storage server to fetch the appropriate emails for the user. Finally, the mail server sends the message to an ad server so that relevant ads can be added to the messages before they are returned to the client.

Note that the adding of ads to emails imposes a significant overhead on performance. This is of particular concern because one of the primary performance goals of a webmail service is to minimize the response time observed by clients. In addition, emails and ads cannot be fetched in parallel, since relevant ads cannot be selected without knowing the contents of the emails. It is also difficult to pre-compute the relevant ads because the relevance of ads may change over time.

Using RPC chains, we can mitigate some of the ad-related overheads. Even though we can only fetch ads after fetching the emails, we can eliminate one latency hop from the communication path of the web mail application by creating a chain that causes emails to be sent directly from the storage server to the ad server, without having to go through to email server (as shown in step 7 of Figure 6). Once the ad server has appended the appropriate ads to the emails, the emails can be sent to the email server which then returns it to the client. In Section 5.4, we evaluate the benefit of using RPC chains to improve the communication pattern in this fashion.

5 Evaluation

We now evaluate RPC chains. We start with some microbenchmarks, in which we measure the overhead of chaining functions and we compare RPC chains versus standard RPC's. We then evaluate the storage and web applications to demonstrate the performance improvements provided by RPC chains. The general question we address is when are RPC chains advantageous and what are the exact benefits.

5.1 Setup

In this section, we present the evaluation of our storage and multi-tier web application. Our experimental setup consists of ten machines in four geodistributed sites in a corporate network that spans the globe. We had machines in 4 sites: (1) Mountain View, California, USA, (2) Redmond, Washington, USA, (3) Cambridge, United Kingdom, and (4) Beijing, China. The measured latency and throughput of the links between these sites are shown in Figure 7.

5.2 Microbenchmarks

5.2.1 Overhead of chaining functions

In our first experiment, we evaluate the overhead imposed by chaining functions (pieces of client code) at servers. We considered chaining functions of three sizes, 621 bytes, 5 KB, and 50 KB, corresponding to small, medium, and large functions.

We first measured the time it takes to compile a function at run-time. The results are shown in the first two columns of Figure 8, averaged over 10 runs (± refers to standard error). We used a 3 Ghz Intel Core 2 Duo processor running Windows Vista Enterprise SP1. The functions were written in C# and compiled using Microsoft Visual Studio 2008.

(a)

	Redmond	Beijing	Cambridge
Mt.View	32 ms	180 ms	240 ms
Redmond		146 ms	210 ms
Beijing			354 ms

(b)

	Redmond	Beijing	Cambridge
Mt. View	6.3 MB/s	2.1 MB/s	1.4 MB/s
Redmond		8.5 MB/s	8.6 MB/s
Beijing			2.4 MB/s

Figure 7: (a) Ping round-trip times and (b) bandwidth of TCP connections between pair of sites.

Source size (KB)	Compile time (ms)	Compiled size (KB)
0.6	45.7 ± 0.3	0.4
5	47.1 ± 0.4	4.6
50	76.0 ± 0.3	15.9

Figure 8: Overhead for compiling chaining functions and storing compiled code.

We also did a linear regression with a larger set of points (17 sizes, with 10 runs each) and found that the cost of compilation is 44.8 ms plus 1 ms for each 5000 bytes of source code. We see that there is a large initial compilation cost of tens of milliseconds, which we do not want to pay every time we call the server in a chain.

We measured the size of the compiled code, shown in the third column of Figure 8. We see that it is very small (we initially thought it would be large, but this is not the case). This allows the server to cache even tens of thousands of chaining functions in less than 50 MB, which justifies our choice of doing so.

5.2.2 RPC chain versus standard RPC

In our next experiment, we compare the latency of an RPC chain versus standard RPC. We used the smallest non-trivial chain, which goes through two servers (A

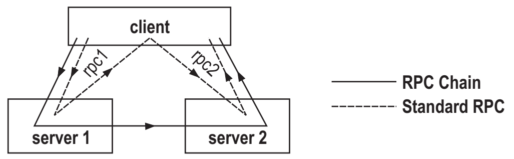


Figure 9: Executions used in the experiment of Section 5.2.2.

chain that goes through only one server is the same as an RPC), and compare it against a pair of consecutive RPC's going to the two servers, as shown in Figure 9. To isolate concerns, the service executed at each server is a no-op.

The figure makes it clear that the RPC chain incurs one fewer hop than the pair of RPC calls. What is not shown is that the RPC chain has potentially two overheads that the pair of RPC calls do not: (1) even if the client needs the response from server 1 but server 2 does not, the data is still relayed through server 2, and (2) the client needs to send state for the chaining function to execute at server 1. The first overhead can be avoided through a simple extension to RPC chains to allow each server in the chain to send some data to the client (Section 7.1).

We now consider the second overhead, and examine the question of how much state the client can send while still allowing the RPC chain to be faster than the pair of RPC calls. We assume that the chaining function is already cached at server 1, which is the common case for frequent chains.

Back-of-the-envelope calculation. We start with a simple calculation. Let S be the size of the state sent by the client for the chaining function at server 1. Then, in terms of total latency, the RPC chain saves one network latency but incurs $S/link\_bandwidth$ to send the state. Thus, the RPC chain fares better as long as $link\_latency > S/link\_bandwidth$, or

$$S < link\_latency \times link\_bandwidth$$

For wide area links, the latency-bandwidth product can easily be in the tens to hundreds of kilobytes or more.

Experiment. We executed the RPC chain and the pair of RPC's. The client was located in Redmond while the servers were in Mountain View. (Because both servers were in the same site, this setup favors the RPC chain by an additional network latency; we later explain the case when the servers are far apart.)

Figure 10 shows the client end-to-end latency as a function of the state size (error bars show standard error). For the standard RPC execution, state size does not affect total latency, since this state simply stays at the client. The total latency was 75±1 ms. For the RPC chain, the latency naturally increases with the state size. The point at which both lines cross is at ≈150 KB. This is a fair amount of state to send in many cases—definitely much more than we needed in either of our applications.

If servers 1 and 2 were far apart, this would shift the RPC chain line up by the corresponding extra latency. For example, if the latency from server 1 to server 2 were 15 ms, the lines would cross at ≈100 KB (assuming the distance from client to server 2 remains the same), which is still a reasonable state size (and much more than we needed in our applications).

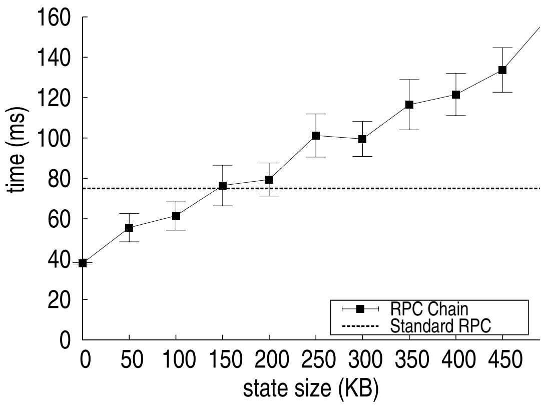


Figure 10: Execution time using an RPC chain versus standard RPC to call 2 servers.

5.3 Storage application

We now evaluate the use of RPC chains for the storage application described in Section 4.1.

5.3.1 Copy performance

In our experiments, we copy data from one storage server to another using two utilities: one that uses RPC chains, called *Chain copy*, and another that uses standard RPC's, called *RPC copy*. Both utilities use pipelining, so that the client has multiple outstanding requests on either server. We also tried using the operating-system provided "copy" program, but it performed much worse than either Chain copy or RPC copy, because it it reads and writes one chunk of data at a time (no pipelining).

In our first experiment, a single client copies a file of variable size (25 KB, 100 KB, 250 KB, and 500 KB) between two servers, and we measure the time it takes. We vary the location of the client (Mt. View, Redmond, Beijing) and fix the location of the servers in Mt. View. In the setting where both the client and the servers were in Mt. View, we placed them in two separate subnets, where the ping latency between the two was 2 ms and TCP bandwidth was 10 MB/s.

Figure 11 shows the results. Each bar represents the median of 40 repetitions of the experiment. As we can see, Chain copy provide considerable benefits in every case, compared to RPC copy. The benefits are greater for larger files and longer distances between client and servers. In a local setting, the copying time is reduced by up to factor of 2, while in the longest-distance setting (Beijing-Mt. View), the reduction is up to a factor of 5.

Another benefit of using Chain copy (not shown) is a reduction by a factor of two in (a) the aggregate network bandwidth consumption, and (b) the client bandwidth consumption. This reduction is important because links

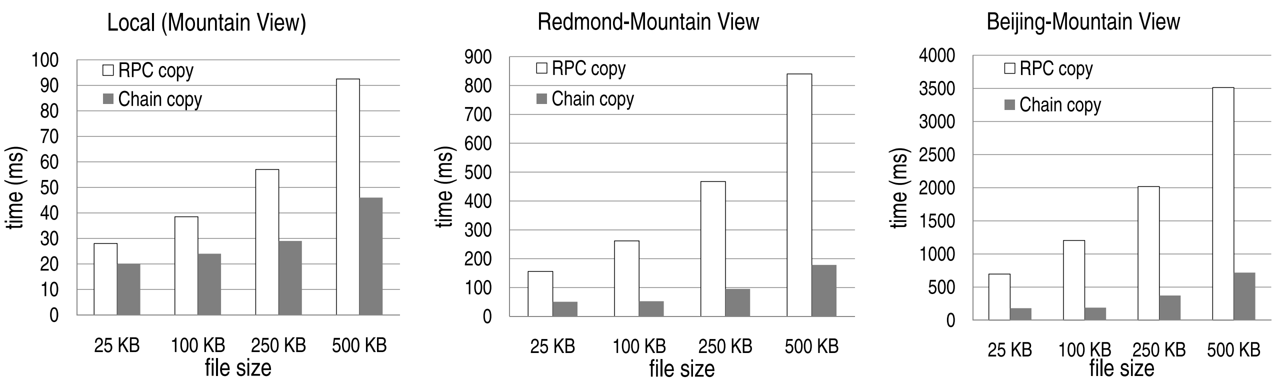


Figure 11: Comparison of RPC copy and Chain copy under various settings. **(Left)** Client and servers are in the same site in Mt. View. **(Center)** Client is in Redmond and servers are in Mt. View. **(Right)** Client is in Beijing and servers are in Mt. View.

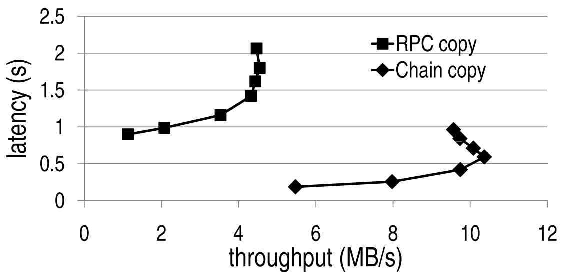


Figure 12: Throughput-latency of RPC copy and Chain copy. Latency is the time to copy a 128 KB file, and throughput is the rate at which files are copied.

connecting data centers have limited bandwidth and/or are priced based on the bandwidth used.

In our next experiment, we vary the number of clients simultaneously copying files from one server to another, and measure the resultant throughput and latency of the system. This allows us to observe the behavior of the system under varying load as well as measure the peak throughput of the system. As before, the client machine was located in Redmond and the servers were located in Mt. View. We ran multiple client instances in parallel on the client machine, each client copying 1000 files in succession, each file measuring 256 KB in size. We measure the time that each client takes to complete copying 1000 files, and compute conservative throughput and latency numbers based on the slowest client.

Figure 12 shows the results of the experiment. For both RPC copy and Chain copy, the average latency decreases as the amount of workload placed on the system increases. Initially, the increase in workload also results in an increase in the aggregate throughput of the system, but once the system becomes saturated, any increase in workload only increases latency without any gain in throughput. Our results show that RPC copy is able to sustain a peak throughput of 4.5 MB/s. This peak throughput occurs when the network link between the client and the servers, which had a bandwidth of 6.3 MB/s, becomes saturated. Since Chain copy does not require that the data blocks of the files being copied actually flow through the client, it was not subject to this limitation and was thus able to achieve a higher peak throughput of 10.4 MB/s. Rather than a network bandwidth limitation, Chain copy's throughput is limited by the servers' ability to keep up with requests.

5.3.2 Benefit of chain composition

In this experiment, we measure the benefit of composing RPC chains. We use two chains: one for copying from one server to another (as above) and the other for primary-backup replication of the second server (as in Figure 5). We compare two systems that use RPC chains; one system uses chain composition to combine the two chains, while the other has composition disabled. In the experiment, one client copies one file of variable size from the non-replicated server to the replicated server. The client is in Cambridge, the source server is in Mt. View, the primary of the destination server is in Mt. View, and the backup of the destination server is in Redmond.

Figure 13 shows the result. As we can see, composing the chain reduces the duration of the copy by 12%-20%, with larger files having a greater reduction. Without composition, the destination server has to handle both requests from the source server as well as the replies from the backup server. Composition reduces the load on the destination server by allowing the backup server to send replies directly to the client. In addition, composition

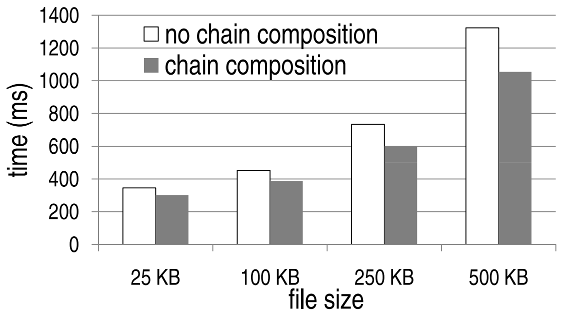


Figure 13: Benefit of chain composition.

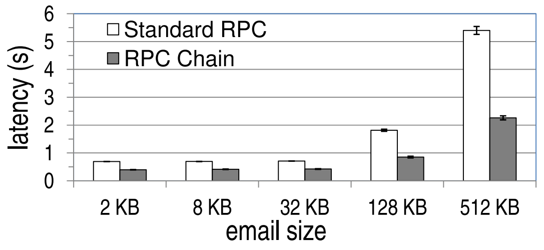


Figure 14: RPC chain in web mail application.

eliminates the unnecessary messages from the backup server to the destination server, reducing the amount of bandwidth consumption. A combination of these factors allow composition to improve the overall performance of the system. As file size increases, the setup cost becomes relatively small compared to the actual cost of executing the chains. This makes the impact of the more efficient chain that resulted from composition more apparent.

5.4 Web mail application

We now describe the evaluation of the web mail application presented in Section 4.2. In our experimental setup, we placed the client in Mountain View, the mail server and the authentication server in Redmond, and all other servers in Beijing. This setup emulates the case where a user from Asia travels to the US and wants to access web mail services that are hosted in Asia. Since the web mail provider may have servers deployed worldwide, the user can be directed to a mail server and an authentication server (Redmond) that is close to his current location (Mountain View). However, user-specific data is stored on servers close to the user's normal location (Beijing), so the mail server has to fetch data from those machines.

Specifically, after receiving a cookie from the client and verifying the client's identity, the mail server must fetch the client's email from the storage server followed by appropriate ads from the ad server, both of which are located in Beijing. A traditional system implemented using RPC's would have the mail server contact the storage server, fetch the user's emails, then contact the ad server to retrieve relevant ads. However, in our setting, where the mail server is located close to the client but far away from the storage server and ad server, traversing the long links between Redmond and Beijing four times would be less than ideal. As described in Section 4.2, RPC chains allow us to eliminate unnecessary network traversals. In this case, our RPC-chain-enabled mail server sends emails directly from the storage server to the ad server before returning the result to the mail server, halving the number of long link traversals.

We measure the client perceived latency of opening an inbox and retrieving one email: the client first contacts the front end authentication server to authenticate herself, then she sends a read request to the mail server to retrieve a single email. We measure the time it takes for the client to receive the email, which is appended with an ad whose size is small relative to the size of the email. We vary the size of the email that is fetched, and for each size, we repeated the experiment 20 times.

As shown in Figure 14, RPC chains consistently reduces the client perceived latency of the web mail application. As the size of the email increases, the latency improvement from using RPC changes also increases. Overall, we found that the use of RPC chains reduced the latency of the web mail application by 40% to 58% when compared to standard RPC's.

We note that the significant performance gains of using RPC chains comes at a very low cost of implementation. For the web mail application, the effort involved in enabling RPC chains was mainly in terms of implementing chaining functions which totaled a mere 48 lines of C# code. In general, a simple way for existing applications to benefit from RPC chains is to identify the critical causal path of RPC requests, and replace that path with an RPC chain. The effort is that of writing a single RPC chain; in the worst case, one can do it from scratch. The harder problem is finding the critical causal path, which has been an active area of research (e.g., [1]).

6 Limitations

We now describe some limitations of RPC chains.

Chaining state cannot always be sent. RPC chains are not appropriate if the chaining state is large or if it cannot be determined when the client starts the chain. For example, suppose that (1) A calls B using an RPC, (2) A gets a reply, and (3) depending on the state of a sensor or some immediate measurement at A, A then calls C or D. It is not possible to use an RPC chain $A \rightarrow B \rightarrow (C \text{ or } D)$, because the choice of going to C versus D must be made at A where the sensor is.

Programming with continuations. To use RPC chains, developers need to make use of continuation-style programming. This can be much harder than programming using sequential code, because continuations must explicitly keep track of all their state. Continuations are notoriously hard to debug, because there is no simple way to track the execution that led to a given state.

We note, however, that programming with continuations is already tolerated in code that uses asynchronous RPC's and callbacks. Moreover, one could perhaps write a tool that automatically produces continuations from sequential code, using techniques from the compiler literature (see, e.g., [3]).

Terminating chains. When an application terminates, it is usually desirable to release its resources and halt all its activities. However, if the application has outstanding RPC chains, it is not easy to terminate them. This problem exists with traditional RPC's as well (there is no easy way to terminate a remote procedure), but it is worse with RPC chains because the remote servers involved may not be known.

RPC chains are designed for relatively short-lived executions, and for these uses, this problem is less of a concern, because a chain soon terminates anyways. The only exception is a buggy chain that runs forever. For such chains, the RPCC library can impose a maximum chain length, say 2000 hops, and throw an exception after that.

7 Extensions

We now discuss some extensions of RPC chains.

7.1 Intermediate chain results

If a client wants to receive some results from intermediate servers of the chain, these results need to be relayed through the chain. If the amount of data is large, it can impose a significant overhead. We can extend RPC chains to address this issue, by allowing each server in the chain to directly return some data to the client. This data is application-specific and is returned by the chaining function. Thus, we add a new field, `client-response`, to the `nexthop` result of a chaining function. The RPCC library sends `client-response` to the client concurrently with continuing the chain.

What happens under chain composition? In this case, the "client" that gets `client-response` is the server that created the sub-chain. The name of these creators, at each level of composed chain, are kept in the chain stack (the chain stack is explained in Section 3.4).

7.2 Dealing with large chaining states

The chaining state is the state that the client sends along the chain to execute the chaining functions. If this state is large, this can incur a significant state overhead. Two optimizations are possible to mitigate this cost.

Fall-back to standard RPC. As explained in Section 3.7, we can execute a chain in interactive mode, which causes the chain to go back to the client at every step. This is effectively a fall-back to standard RPC, causing all chaining functions to execute at the client, which eliminates the overhead of sending the chaining state, at the cost of extra network delays. We explored this trade-off in Section 5.2.2. It is possible to have the RPCC library gauge the size of the chaining state before starting the chain, and if the state is larger than some threshold, execute the chain in interactive mode. The threshold can be chosen dynamically based on previous executions of the same chain, in an adaptive manner. By doing so, an RPC chain will always perform at least as well as standard RPC's, modulo the small computational overhead of executing chaining functions and the time it takes to adapt. However, in the applications we examined in this paper, we did not need this technique because the chaining state was always small.

Hiding latency. In our implementation, servers wait to receive the chaining state before executing the next service function in the chain. This waiting is not necessary, because the service function depends only on its parameters, not on the chaining state (the chaining state is only needed for the chaining function, which executes later). Therefore, a natural optimization is to start the service function even as the chaining state is being received. If the service function takes significant time to complete, (e.g., it involves disk I/O or some lengthy computation), this will mask part or all of the latency of transmitting the chaining state.

7.3 Chaining proxy

As we said, chaining functions are *portable code* that do not have to execute at the server. They can execute at a designated *chaining proxy* machine, to avoid any overhead at the server. Doing so incurs extra communication, but if the chaining proxy is geographically close to the server, this cost is small relative to that of a wide-area hop. To choose the chaining proxy, we can use a simple mapping from servers to nearby proxies configured by an administrator.

8 Related work

RPC chains utilize two well-understood ideas in the context of remote execution: *function shipping*, and *continuations*.

Function shipping is the general technique of sending computation to the data rather than bringing the data to the computation. It is used in some systems where the cost of moving data is large compared to the cost of moving computation. For example, Diamond [10] is a storage architecture in which applications download searchlet code to disk to perform efficient filtering of large data sets locally, thereby improving efficiency. RPC chains use function shipping to send chaining logic.

A continuation [17] refers to the shifting of program control and transfer of current state from one part of a program to another. Extending this to *distributed continuations* is a natural step, allowing a continuation to shift program control from one processor to another. Several works in the parallel programming community give high-level programming continuation constructs and specify their behavior formally, e.g., [12, 11]. Distributed continuations were exploited to enhance the functionality of web servers and overcome the stateless nature of HTTP interaction. By comparison, the RPC chain is a generic mechanism that is independent of the service provided by servers. RPC chains support complex chaining structures, and can be used with a diverse set of servers.

The above mentioned ideas for code mobility, and others, are leveraged in a variety of high-level programming paradigms for distributed execution. Distributed workflows, e.g., [5, 22], can use distributed continuations to distribute a workflow description in a decentralized fashion. MapReduce [6], and Dryad [23] are programming models for data-parallel jobs, such as a data mining calculations, which process large amounts of data in batches. These systems target self-contained jobs that execute for substantial periods, while RPC Chains are intended for short-lived remote executions in an environment with many diverse services that are possibly developed independently of their applications. Mobile agents have been extensively studied in the literature and many systems have been built, including Telescript/Odyssey [19], Aglets [4], D'Agents [8], and others (see e.g., [20, 9]). A mobile agent is a process that can autonomously migrate itself from host to host as it executes; migration involves moving the process's current state to the new host and resuming execution. The motivation for mobile agents include (a) bringing processes closer to the resources they need in a given stage of the computation, and (b) allowing clients to disconnect from the network while an agent executes on their behalf. An RPC Chain can be considered as a mobile agent whose purpose is to execute a series of RPC calls. However, mobile agents are much more general and ambitious than RPC Chains (which possibly contributed to their eventual demise): they have social abilities, being able to adjust their behavior according to the host in which they are currently executing; they can learn about execution environments never envisioned by their creators; and they can persist if the clients that created them disappear. Much of the literature regarding mobile agents is about security (how agents can survive malicious hosts, and how hosts can protect themselves against malicious agents) and language support for code mobility (how to write programs that can transparently move to other machines). For RPC chains, security is a smaller concern in the trusted data center and enterprise environments that we consider, and we are not concerned about transparent mobility.

Some related work includes more targeted uses of mobile code. Work on Active Networks introduced network packets called *capsules*, which carry code that network switches execute to route the packet (see [18] for a survey). This provides a general scheme for extending network protocols beyond the existing deployed base, and allows for more dynamic routing schemes. In contrast, RPC chains are aimed at higher-level applications, and their main purpose is to eliminate communication hops when a client needs to call many services in succession.

Distributed Hash Tables (e.g., Chord [16], CAN [14], Pastry [15], Tapestry [24]) have a *lookup* protocol, for finding the host responsible for a given key. Such protocols generally need to contact several hosts successively, and this can be done in two ways. In an *interactive* lookup protocol, the host that initiates the lookup operation issues RPC's to each host in succession. A *recursive* lookup protocol [7] works like a routing protocol: the host that initiates the operation contacts the first host in the sequence, which in turn contacts the next one, and so forth; when a host finds the key, it contacts the request initiator directly. This protocol is hard-coded into the lookup operation, and it is executed by a set of servers that implement this operation. In contrast, RPC chains provide a generic chaining mechanism that is independent of the operation (service function) executed.

Finally, SOAP [21] is a protocol that supports RPC's using XML over HTTP. It has the notion of intermediaries that can process a SOAP message (RPC) before it reaches the final recipient. However, there is no client logic that routes and transform messages, and the notion of a pre-specified distinguished final recipient is inherent to SOAP. Typical uses for intermediary nodes include blocking messages (firewall), buffering and batching of messages, tracing, and encrypting/decrypting messages as it passes through an untrusted domain.

9 Conclusion

We proposed the RPC chain, a simple but powerful primitive that combines multiple RPC invocations into a chain, in order to optimize the communication pattern of applications that use many composite services, possibly developed independently of each other. With RPC chains, client can save network hops, resulting in considerably smaller end-to-end latencies in a geodistributed setting. Clients can also save bandwidth because they are not forced to receive data they do not need. We demonstrated the use of RPC chains for a storage and a web application, and we think RPC chains could have many more applications beyond those.

Acknowledgements. We thank Rama Subramanian, our shepherd Michael J. Freedman, and the anonymous reviewers for helpful comments. This research was supported in part by NSF Grant 0424422 and U.S. Dept. of Homeland Security Grant 2006-CS-001-000001-02.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *ACM Symposium on Operating Systems Principles* (Oct. 2003), pp. 74–89.

[2] AMAZON.COM, INC. Amazon simple storage service: Copy proposal. `http://doc.s3.amazonaws.com/proposals/copy.html`.

[3] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[4] ARIDOR, Y., AND OSHIMA, M. Infrastructure for mobile agents: Requirements and design. In *Workshop on Mobile Agents* (Sept. 1998), pp. 38–49.

[5] BARBARÁ, D., MEHROTRA, S., AND RUSINKIEWICZ, M. INCAs: Managing dynamic workflows in distributed environments. *Journal of Database Management, Special Issues on Multidatabases 7*, 1 (Winter 1996), 5–15.

[6] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *ACM Symposium on Operating System Design and Implementation* (Dec. 2004), pp. 137–150.

[7] FREEDMAN, M. J., LAKSHMINARAYANAN, K., RHEA, S., AND STOICA, I. Non-transitive connectivity and DHTs. In *Conference on Real, Large Distributed Systems* (Dec. 2005), pp. 55–60.

[8] GRAY, R., KOTZ, D., NOG, S., RUS, D., AND CYBENKO, G. Mobile agents: The next generation in distributed computing. In *Aizu International Symposium on Parallel Algorithms and Architectures Synthesis* (Mar. 1997), pp. 8–24.

[9] HARRISON, C. G., CHESS, D. M., AND KERSHENBAUM, A. Mobile Agents: Are they a good idea? In *International Workshop on Mobile Object Systems* (July 1996), pp. 25–47.

[10] HUSTON, L., SUKTHANKAR, R., WICKREMESINGHE, R., SATYANARAYANAN, M., GANGER, G. R., RIEDEL, E., AND AILAMAKI, A. Diamond: A Storage Architecture for Early Discard in Interactive Search. In *USENIX Conference on File and Storage Technologies* (Mar. 2004), pp. 73–86.

[11] JAGANNATHAN, S. Continuation-based transformations for coordination languages. *Theoretical Computer Science 240*, 1 (June 2000), 117–146.

[12] MOREAU, L. The PCKS-machine: An abstract machine for sound evaluation of parallel functional programs with first-class continuations. In *European Symposium on Programming* (Apr. 1994), pp. 424–438.

[13] NIELSEN, J. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Indianapolis, 1999.

[14] RATNASAMY, S., FRANCIS, P., HANDLEY, M., KARP, R. M., AND SHENKER, S. A scalable content-addressable network. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Aug. 2001), pp. 161–172.

[15] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *International Conference on Distributed Systems Platforms* (Nov. 2001), pp. 329–350.

[16] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (Aug. 2001), pp. 149–160.

[17] STRACHEY, C., AND WADSWORTH, C. P. Continuations: A mathematical semantics for handling full jumps. *Higher-Order and Symbolic Computation 13*, 1-2 (Apr. 2000), 135–152.

[18] TENNENHOUSE, D. L., SMITH, J. M., SINCOSKIE, W. D., WETHERALL, D. J., AND MINDEN, G. J. A survey of active network research. *IEEE Communications Magazine 35*, 1 (Jan. 1997), 80–86.

[19] WHITE, J. Telescript technology: The foundation for the electronic marketplace, 1994. Unpublished manuscript. White paper, General Magic, Inc.

[20] WHITE, J. Mobile agents white paper, 1996. Unpublished manuscript. Available at `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.7931`.

[21] WORLD WIDE WEB CONSORTIUM. SOAP version 1.2. `http://www.w3.org`.

[22] YU, W., AND YANG, J. Continuation-passing enactment of distributed recoverable workflows. In *ACM Symposium on Applied Computing* (Mar. 2007), pp. 475–481.

[23] YU, Y., ISARD, M., FETTERLY, D., BUDIU, M., ERLINGSSON, Ú., GUNDA, P. K., AND CURREY, J. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *ACM Symposium on Operating System Design and Implementation* (Dec. 2008), pp. 1–14.

[24] ZHAO, B. Y., HUANG, L., STRIBLING, J., RHEA, S. C., JOSEPH, A. D., AND KUBIATOWICZ, J. D. Tapestry: A resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications 22*, 1 (Jan. 2004), 41–53.

Studying Spamming Botnets Using Botlab

John P. John Alexander Moshchuk Steven D. Gribble Arvind Krishnamurthy

Department of Computer Science & Engineering
University of Washington

Abstract

In this paper we present Botlab, a platform that continually monitors and analyzes the behavior of spam-oriented botnets. Botlab gathers multiple real-time streams of information about botnets taken from distinct perspectives. By combining and analyzing these streams, Botlab can produce accurate, timely, and comprehensive data about spam botnet behavior. Our prototype system integrates information about spam arriving at the University of Washington, outgoing spam generated by captive botnet nodes, and information gleaned from DNS about URLs found within these spam messages.

We describe the design and implementation of Botlab, including the challenges we had to overcome, such as preventing captive nodes from causing harm or thwarting virtual machine detection. Next, we present the results of a detailed measurement study of the behavior of the most active spam botnets. We find that six botnets are responsible for 79% of spam messages arriving at the UW campus. Finally, we present defensive tools that take advantage of the Botlab platform to improve spam filtering and protect users from harmful web sites advertised within botnet-generated spam.

1 Introduction

Spamming botnets are a blight on the Internet. By some estimates, they transmit approximately 85% of the 100+ billion spam messages sent per day [14, 21]. Botnet-generated spam is a nuisance to users, but worse, it can cause significant harm when used to propagate phishing campaigns that steal identities, or to distribute malware to compromise more hosts.

These concerns have prompted academia and industry to analyze spam and spamming botnets. Previous studies have examined spam received by sinkholes and popular web-based mail services to derive spam signatures, determine properties of spam campaigns, and characterize scam hosting infrastructure [1, 39, 40]. This analysis of "incoming" spam feeds provides valuable information on aggregate botnet behavior, but it does not separate activities of individual botnets or provide information on the spammers' latest techniques. Other efforts reverse engineered and infiltrated individual spamming botnets, including Storm [20] and Rustock [5]. However, these techniques are specific to these botnets and their communication methods, and their analysis only considers characteristics of the "outgoing" spam these botnets generate. Passive honeynets [13, 27, 41] are becoming less applicable to this problem over time, as botnets are increasingly propagating via social engineering and web-based drive-by download attacks that honeynets will not observe. Overall, there is still opportunity to design defensive tools to filter botnet spam, identify and block botnet-hosted malicious sites, and pinpoint which hosts are currently participating in a spamming botnet.

In this paper we turn the tables on spam botnets by using the vast quantities of spam that they generate to monitor and analyze their behavior. To do this, we designed and implemented *Botlab*, a continuously operating botnet monitoring platform that provides real-time information regarding botnet activity. Botlab consumes a feed of all incoming spam arriving at the University of Washington, allowing it to find fresh botnet binaries propagated through spam links. It then executes multiple captive, sandboxed nodes from various botnets, allowing it to observe the precise outgoing spam feeds from these nodes. It scours the spam feeds for URLs, gathers information on scams, and identifies exploit links. Finally, it correlates the incoming and outgoing spam feeds to identify the most active botnets and the set of compromised hosts comprising each botnet.

A key insight behind Botlab is that the combination of *both* incoming and outgoing spam sources is essential for enabling a comprehensive, accurate, and timely analysis of botnet behavior. Incoming spam bootstraps the process of identifying spamming bots, outgoing spam enables us to track the ebbs and flows of botnets' ongoing spam campaigns and establish the ground truth regarding spam templates, and correlation of the two feeds can classify incoming spam according to botnet that is sourcing it, determine the number of hosts active within each botnet, and identify many of these botnet-infected hosts.

1.1 Contributions

Our work offers four novel contributions. First, we tackle many of the challenges involved in building a real-time botnet monitoring platform, including identifying and incorporating new bot variants, and preventing Botlab hosts from being blacklisted by botnet operators.

Second, we have designed network sandboxing mechanisms that prevent captive bot nodes from causing harm, while still enabling our research to be effective. As well, we discuss the long-term tension between effectiveness and safety in botnet research given botnets' trends, and we present thought experiments that suggest that a determined adversary could make it extremely difficult to conduct future botnet research in a safe manner.

Third, we present interesting behavioral characteristics of spamming botnets derived from our multi-perspective analysis. For example, we show that just a handful of botnets are responsible for most spam received by UW, and attribute incoming spam to specific botnets. As well, we show that the bots we analyze use simple methods for locating their command and control (C&C) servers; if these servers were efficiently located and shut down, much of today's spam flow would be disrupted. As another example, in contrast to earlier findings [40], we observe that some spam campaigns utilize multiple botnets.

Fourth, we have implemented several prototype defensive tools that take advantage of the real-time information provided by the Botlab platform. We have constructed a Firefox plugin that protects users from scam and phishing web sites propagated by spam botnets. The plug-in blocked 40,270 malicious links emanating from one botnet monitored by Botlab; in contrast, two blacklist-based defenses failed to detect any of these links. As well, we have designed and implemented a Thunderbird plugin that filters botnet-generated spam. For one user, the plugin reduced the amount of spam that bypassed his SpamAssassin filters by 76%.

The rest of this paper is organized as follows. Section 2 provides background material on the botnet threat. Section 3 discusses the design and implementation of Botlab. We evaluate Botlab in Section 4 and describe applications we have built using it in Section 5. We discuss our thoughts on the long-term viability of safe botnet research in Section 6. We present related work in Section 7 and conclude in Section 8.

2 Background on the Botnet Threat

A botnet is a large-scale, coordinated network of computers, each of which executes specific bot software. Botnet operators recruit new nodes by commandeering victim hosts and surreptitiously installing bot code onto them; the resulting army of "zombie" computers is typically controlled by one or more command-and-control (C&C) servers. Botnet operators employ their botnets to send spam, scan for new victims, steal confidential information from users, perform DDoS attacks, host web servers and phishing content, and propagate updates to the botnet software itself.

Botnets originated as simple extensions to existing Internet Relay Chat (IRC) softbots. Efforts to combat botnets have grown, but so has the demand for their services. In response, botnets have become more sophisticated and complex in how they recruit new victims and mask their presence from detection systems:

Propagation: Malware authors are increasingly relying on social engineering to find and compromise victims, such as by spamming users with personal greeting card ads or false upgrade notices that entice them to install malware. As propagation techniques move up the protocol stacks, the weakest link in the botnet defense chain becomes the human user. As well, systems such as passive honeynets become less effective at detecting new botnet software, instead requiring active steps to gather and classify potential malware.

Customized C&C protocols: While many of the older botnet designs used IRC to communicate with C&C servers, newer botnets use encrypted and customized protocols for disseminating commands and directing bots [7, 9, 33, 36]. For example, some botnets communicate via HTTP requests and responses carrying encrypted C&C data. Manual reverse-engineering of bot behavior has thus become time-consuming if not impossible.

Rapid evolution: To evade detection from trackers and anti-malware software, some newer botnets morph rapidly. For instance, most malware binaries are often packed using polymorphic packers that generate different looking binaries even though the underlying code base has not changed [29]. Also, botnet operators are moving away from relying on a single web server to host their scams, and instead are using *fast flux DNS* [12]. In this scheme, attackers rapidly rebind the server DNS name to different botnet IP addresses, in order to defend against IP blacklisting or manual server take-down. Finally, botnets also make updates to their C&C protocols, by incorporating new forms of encryption and command distribution.

Moving forward, analysis and defense systems must contend with the increasing sophistication of botnets. Monitoring systems must be pro-active in collecting and executing botnet samples, as botnets and their behavior change rapidly. As well, botnet analysis systems will increasingly have to rely on external observations of botnet behavior, rather than necessarily being able to crack and reverse engineer botnet control traffic.

3 The Botlab Monitoring Platform

The Botlab platform produces fresh information about spam-oriented botnets, including their current campaigns, constituent bots, and C&C servers. Botlab partially automates many aspects of botnet monitoring, reducing but not eliminating the manual effort required of a human operator to analyze new bot binaries and incorporate them into Botlab platform.

Botlab's design was motivated by four requirements:

1. *Attribution:* Botlab must identify the spam botnets that are responsible for campaigns and the hosts that

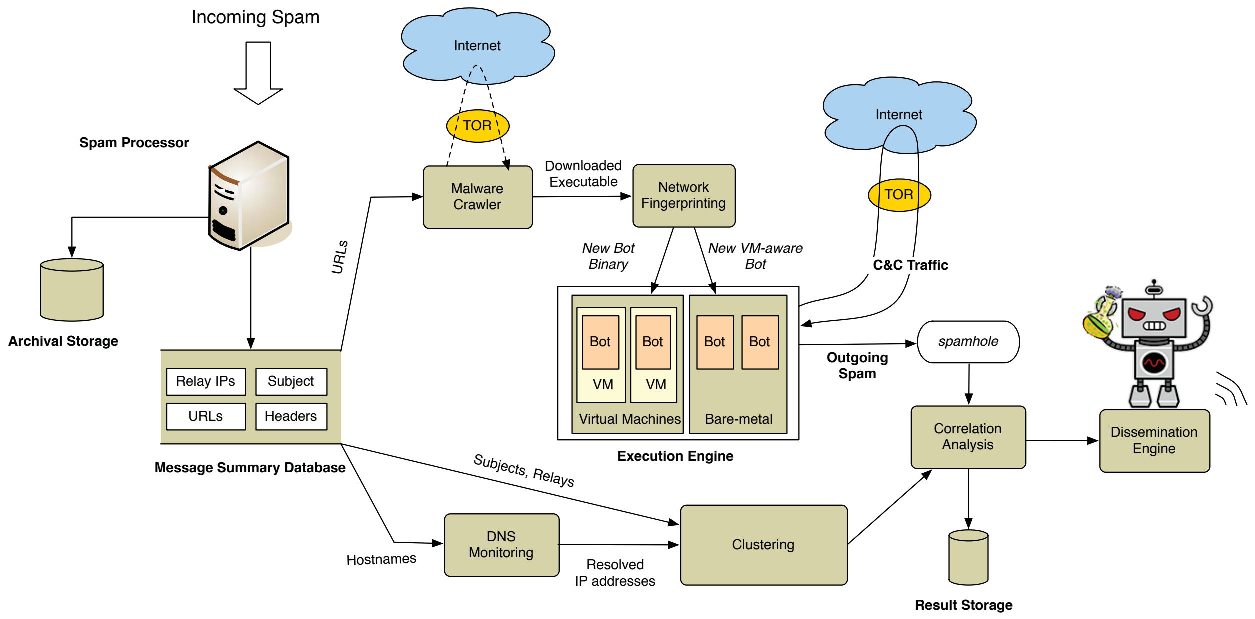


Figure 1: **Botlab Architecture.** Botlab coordinates and monitors multiple source of data about spam botnets, including incoming spam from the University of Washington, and outgoing spam generated by captive bot nodes.

belong to those botnets.

2. *Adaptation:* Botlab must track changes in the botnets' behavior over time.

3. *Immediacy:* Because the value of information about botnet behavior degrades quickly, Botlab must produce information on-the-fly.

4. *Safety:* Botlab must not cause harm.

There is a key tension in our work between safety and effectiveness, similar to tradeoff between safety and fidelity identified in the Potemkin honeyfarm [34]. In Section 6, we discuss this tension in more detail and comment on the long-term viability of safe botnet research.

Figure 1 shows the Botlab architecture. We now describe Botlab's main components and techniques.

3.1 Incoming Spam

Botlab monitors a live feed of spam received by approximately 200,000 University of Washington e-mail addresses. On average, UW receives 2.5 million e-mail messages each day, over 90% of which is classified as spam. We use this spam feed to collect new malware binaries, described next, and within Botlab's correlation engine, described in Section 3.5.

3.2 Malware Collection

Running captive bot nodes requires up-to-date bot binaries. Botlab obtains these in two ways. First, many botnets spread by emailing malicious links to victims; accordingly, Botlab crawls URLs found in its incoming spam feed. We typically find approximately 100,000 unique URLs per day in our spam feed, 1% of which point to malicious executables or drive-by downloads. Second, Botlab periodically crawls binaries or URLs contained in public malware repositories [3, 25] or collected by the MWCollect Alliance honeypots [22].

Given these binaries, a human operator then uses Botlab's automated tools for malware analysis and fingerprinting to find bot binaries that actively send spam, as discussed next in Section 3.3. Our experience to date has yielded two interesting observations. First, though the honeypots produced about 2,000 unique binaries over a two month period, none of these binaries were spamming bots. A significant fraction of the honeypot binaries were traditional IRC-based bots, whereas the spamming binaries we identified from other sources all used non-IRC protocols. This suggests that spamming bots propagate through social engineering techniques, rather than the automated compromise of remote hosts.

Second, many of the malicious URLs seen in spam point to legitimate web servers that have been hacked to provide malware hosting. Since malicious pages are typically not linked from the legitimate pages on these web servers, an ordinary web crawl will not find them. This undermines the effectiveness of identifying malicious pages using exhaustive crawls, an hypothesis that is supported by our measurements in Section 5.

3.3 Identifying Spamming Bots

Botlab executes spamming bots within sandboxes to monitor botnet behavior. However, we must first prune the binaries obtained by Botlab to identify those that correspond to spamming bots and to discard any duplicate binaries already being monitored by Botlab.

Simple hashing is insufficient to find *all* duplicates, as malware authors frequently repack binaries or release slightly modified versions to circumvent signature-based security tools. Relying on anti-virus software is also impractical, as these tools do not detect many new malware variants.

To obtain a more reliable behavioral signature, Botlab produces a *network fingerprint* for each binary it considers. A network fingerprint captures information about the network connections initiated by a binary. To obtain it, we execute each binary in a safe sandbox and log all outbound network connection attempts. A network fingerprint will then consist of a set of flow records of the form `<protocol, IP address, DNS address, port>`. Note that the DNS address field might be blank if a bot communicates with an IP directly, instead of doing a DNS lookup.

Once network activity is logged, we extract the flow records. We execute each binary two times and take the network fingerprint to be the set of flow records which are common across both executions. This eliminates any random connections which do not constitute stable behavioral attributes. For example, some binaries harvest e-mail addresses and spam subjects by searching `google.com` for random search terms, and following links to the highest-ranked search results; repeated execution identifies and discards these essentially random connection attempts.

Given the network fingerprints N_1 and N_2, of two binaries B_1 and B_2 respectively, we define the similarity coefficient of the binaries, $S(B_1, B_2)$, to be:

$$S(B_1, B_2) = \frac{|N_1 \cap N_2|}{|N_1 \cup N_2|}$$

If the similarity coefficient of two binaries is sufficiently high (we use 0.5 as the threshold), we consider the binaries to be behavioral duplicates. As well, binaries which attempt to send e-mail are classified as spamming bots.

We took a step to validate our duplicate elimination procedure. Unfortunately, given a pair of arbitrary binaries, determining that they are behavioral duplicates is undecidable, so we must rely on an approximation. For this, we used five commercial anti-virus tools and a set of 500 malicious binaries which made network connections. All five anti-virus tools had signatures for only 192 of the 500 binaries, and we used only these 192 binaries in our validation. We considered a pair of binaries to be duplicates if their anti-virus tags matched in the majority of five tools. Note that we do not expect the tags to be identical across different anti-virus tools. Network fingerprinting matched this tag-based classification 98% of the time, giving us reasonable confidence in its ability to detect duplicates. Also, we observed a false-positive rate of 0.62%, where the anti-virus tags did not match, but network fingerprinting labeled the files as duplicates. Note again that anti-virus tools lack signatures for many new binaries our crawler analyzes, making them unfit to use as our main duplicate suppression method.

3.3.1 Safely generating fingerprints

The tension between safety and effectiveness is particularly evident when constructing signatures of newly gathered binaries. A safe approach would log emitted network packets, but drop them instead of transmitting them externally; unfortunately, this approach is ineffective, since many binaries must first communicate with a C&C server or successfully transmit probe email messages before fully activating. An effective approach would allow a binary unfettered access to the Internet; unfortunately, this would be unsafe, as malicious binaries may perform DoS attacks, probe or exploit remote vulnerabilities, transmit spam, or relay botnet control traffic.

Botlab attempts to walk the tightrope between safety and effectiveness. We provide a human operator with tools that act as a safety net: traffic destined to privileged ports, or ports associated with known vulnerabilities, is automatically dropped, and limits are enforced on connections rates, data transmission, and the total window of time in which we allow a binary to execute. As well, Botlab provides operators with the ability to redirect outgoing SMTP traffic to *spamhole*, an emulated SMTP server that traps messages while fooling the sender into believing the message was sent successfully.

We are confident that our research to date has been safe. However, the transmission of any network traffic poses some degree of risk of causing harm to the receiver, particularly when the traffic originates from an untrusted binary downloaded from the Internet. In Section 6, we present our thoughts on the long-term viability of safely conducting this research.

3.3.2 Experience classifying bots

We have found that certain bots detect when they are being run in a virtual machine and disable themselves. To identify VMM detection, Botlab generates two network fingerprints for each binary: we execute the binary in a VMware virtual machine and also on a bare-metal machine containing a fresh Windows installation. By comparing the resulting two network fingerprints, we can infer whether the binary is performing any VM detection.

Some of the spamming binaries we analyzed made initial SMTP connections, but subsequently refused to send spam. For example, one spam bot connected to spamhole, but never sent any spam messages after receiving the initial greeting string from the SMTP server. We deduced that this bot was checking that the greeting string included the domain name to which the bot was connecting, and we modified spamhole to return appropriate domain names in the string.

We also observed that some spam bots perform more sophisticated SMTP verification before they send spam. For example, when the MegaD bot begins executing, it transmits a test e-mail to a special MegaD mail server, verifying each header it receives during the SMTP handshake. MegaD's mail server returns a message ID string after sending the message, which the bot then sends to its C&C server. The C&C server verifies that the message with this ID was actually delivered to the MegaD mail server before giving any further instructions to the bot. Accordingly, to generate a signature for MegaD, and later, to continuously execute a captured MegaD node, the human operator had to indicate to Botlab to deflect SMTP messages destined for MegaD's mail server from the spamhole to the live Internet.

Some bots do not send spam through SMTP, but instead use HTTP-based web services. For example, a Rustock variant rotates through valid `hotmail.com` accounts to transmit spam. To safely intercept this spam, we had to construct infrastructure that spoofs Hotmail's login and mail transmission process, including using fake SSL certificates during login. Fortunately, this variant does not check the SSL certificates for validity. However, if the variant evolves and validates the certificate, we would not be able to safely analyze it.

3.4 Execution Engine

Botlab executes spamming bot binaries within its execution engine. The engine runs each bot within a VM or on a dedicated bare-metal box, depending on whether the bot binary performs VMM detection. In either case, Botlab sandboxes network traffic to prevent harm to external hosts. We re-use the network safeguards described in the previous section in the execution engine sandbox: our sandbox redirects outgoing e-mail to spamhole, permits only traffic patterns previously identified as safe by a human operator to be transmitted to the Internet, and drops all other packets. Traffic permitted on the Internet is also subject to the same rate limiting policies we previously described.

Though we have analyzed thousands of malware binaries to date, only a surprisingly small fraction correspond to unique spamming botnets. In fact, we have so far found just seven spamming bots: Grum, Kraken, MegaD, Pushdo, Rustock, Srizbi, and Storm. (Botnet names are derived according to tags with which antivirus tools classify the corresponding binaries.) We believe these are the most prominent spam botnets existing today, and our results suggest that they are responsible for sending most of the world's spam. Thus, it appears the spam botnet landscape consists of just a handful of key players.

3.4.1 Avoiding blacklisting

If the botnet owners learn about Botlab's existence, they might attempt to blacklist IP addresses belonging to the University of Washington. The C&C servers would then refuse connections to Botlab's captive bots, rendering Botlab ineffective. To prevent this, Botlab routes any bot traffic permitted onto the Internet, including C&C traffic, through the anonymizing Tor network [6]. Our malware crawler is also routed through Tor. While Tor provides a certain degree of anonymity, a long-term solution to avoid blacklisting would be to install monitoring agents at geographically diverse and secret locations, with the hosting provided by organizations that desire to combat the botnet threat.

Some bots track and report the percentage of e-mail messages successfully sent and e-mail addresses for which sending failed. These lists can be used by botnet owners to filter out invalid or outdated addresses. To avoid detection, we had to ensure that our bots did not report 100% delivery rates, as these are unlikely to happen in the real world. Doing so was easy; our bots experience many send errors because of failed DNS lookups for mail servers. Thus, we simply rely on DNS to provide us with a source of randomness in bot-reported statistics. Should bot masters begin to perform a more complicated statistics analysis, more controlled techniques for introducing random failures in spamhole might become necessary.

3.4.2 Multiple C&C servers

Some botnets partition their bots across several C&C servers. For example, in Srizbi, different C&C servers are responsible for sending different classes of spam. These spam classes differ in subject line, content, embedded URLs, and even languages. If we were to run only a single Srizbi bot binary, it would connect to one C&C server, and therefore we would only have a partial view of the overall botnet activity.

To rectify this, we take advantage of a C&C redundancy mechanism built into many bots, including Srizbi: if the primary C&C server goes down, an alternate C&C server is selected either via hardcoded IP addresses or programmatic DNS lookups. Botlab can thus block the primary C&C server(s) and learn additional C&C addresses. Botlab can then run multiple instances of the same bot, each routed to a different C&C server.

3.5 Correlating incoming and outgoing spam

Botlab's correlation analyzer combines our different sources of botnet information to provide a more complete view into overall botnet activity. For example, armed with a real-time *outgoing* spam feed, we can classify spam received by our *incoming* spam feed according to the botnet that is responsible for sending it. We will describe how we derived our classification algorithm and evaluate its accuracy in Section 4.3.1.

For spam that cannot be attributed to a particular botnet using our correlation analysis, we use *clustering* analysis to identify sets of relays used in the same spam campaign. In Section 4.2, we evaluate various ways in which this clustering can be performed. If there is a significant overlap between a campaign's relay cluster and known members of a particular botnet (where botnet membership information is derived from the earlier correlation analysis), then we can merge the two sets of relays to derive a more complete view of botnet membership.

3.6 Summary

We have outlined an architecture for Botlab, a real-time spam botnet monitoring system. Some elements of Botlab have been proposed elsewhere; our principal contribution is to assemble these ideas into an end-to-end system that can safely identify malicious binaries, remove duplicates, and execute them without being blacklisted. By correlating the activity of captured bots with the aggregate incoming spam feed, the system has the potential to provide more comprehensive information on spamming botnets and also enable effective defenses against them. We discuss these issues in the remainder of the paper.

4 Analysis

We now present an analysis of botnets that is enabled by our monitoring infrastructure. First, we examine the actions of the bots being run in Botlab, characterize their behavior, and analyze the properties of the outgoing spam feed they produce. Second, we analyze our incoming spam feed to extract coarse-grained, aggregate information regarding the perpetrators of malicious activity. Finally, we present analysis that is made possible by studying both the outgoing and incoming spam feeds. Our study reveals several interesting aspects of spamming botnets.

4.1 The Spam Botnets

In our analysis, we focus on seven spam botnets: Grum, Kraken, MegaD, Pushdo, Rustock, Srizbi, and Storm. Although our malware crawler analyzed thousands of potential executables, after network fingerprinting and pruning described earlier, we found that only variants of these seven bots actively send spam. Next, we summarize various characteristics of these botnets and our experience running them.

4.1.1 Behavioral Characteristics

Table 1 summarizes various characteristics of our botnets, which we have monitored during the past six months. The second column depicts the number of days on which we have observed a botnet actively sending spam. We found that keeping all botnets active simultaneously is difficult. First, locating a working binary for each botnet required vastly different amounts of time, depending on the timings of botnet propagation campaigns. For example, we have only recently discovered Grum, a new spamming botnet which has only been active for 8 days, whereas Rustock has been running for more than 5 months. Second, many bots frequently go offline for several days, as C&C servers are taken down by law enforcement, forcing the bot herders to re-establish new C&C servers. Sometimes this breaks the bot binary, causing a period of inactivity until a newer, working version is found.

The amount of outgoing spam an individual bot can generate is vastly different across botnets. MegaD and Srizbi bots are the most egregious: they can send out more than 1,500 messages per minute, using as many as 80 parallel connections at a time, and appear to be limited only by the client's bandwidth. On the other hand, Rustock and Storm are "polite" to the victim – they send messages at a slow and constant rate and are unlikely to saturate the victim's network connection. Big variability in send rates suggests these rates might be useful in fingerprinting and distinguishing various botnets.

Bots use various methods to locate and communicate with their C&C servers. We found that many botnets use very simple schemes. Rustock, Srizbi, and Pushdo simply hardcode the C&C's IP address in the bot binary, and MegaD hardcodes a DNS name. Kraken uses a proprietary algorithm to generate a sequence of dynamic DNS names, which it then attempts to resolve until it finds a working name. An attacker registers the C&C server at one of these names and can freely move the C&C to another name in the event of a compromise. In all of these cases, Botlab can efficiently pinpoint the IP addresses of the active botnet C&C servers; if these servers could be efficiently located and shut down, the amount of worldwide spam generated would be substantially reduced.

Although recent analysis suggests that botnet control is shifting to complicated decentralized protocols as exemplified by Storm [20, 33], we found the majority of our other spam bots use HTTP to communicate with their C&C server. Using HTTP is simple but effective, since bot traffic is difficult to filter from legitimate web traffic. HTTP also yields a simple pull-based model for botnet

Botnet	# days active in trace	total spam messages	spam send rate (messages/min)	C&C protocol	C&C servers contacted over lifetime	C&C discovery
Grum	8 days	864,316	344	encrypted HTTP, port 80	1	static IP (206.51.231.192)
Kraken	25 days	5,046,803	331	encrypted HTTP, port 80	41	algorithmic DNS lookups
Pushdo	59 days	4,932,340	289	encrypted HTTP, port 80	96	set of static IPs
Rustock	164 days	7,174,084	33	encrypted HTTP, port 80	1	static IP (208.72.169.54)
MegaD	113 days	198,799,848	1638	encrypted custom protocol, ports 80 and 443	21	static DNS name (majzufaiuq.info)
Srizbi	51 days	86,003,889	1848	unencrypted HTTP, port 4099	20	set of static IPs
Storm	50 days	961,086	20	compressed TCP	N/A	p2p (Overnet)

Table 1: **The botnets monitored in Botlab.** Table gives characteristics of representative bots participating in the seven botnets. Some bots use all available bandwidth to send more than a thousand messages per minute, while others are rate-limited. Most botnets use HTTP for C&C communication. Some do not ever change the C&C server address yet stay functional for a long time.

operators: a new bot makes an HTTP request for work and receives an HTTP response that defines the next task. Upon completing the task, the bot makes another request to relay statistics, such as valid and invalid destination addresses, to the bot master. All of our HTTP bots follow this pattern, which is easier to use and appears just as sustainable as a decentralized C&C protocol such as Storm's protocol.

We checked whether botnets frequently change their C&C server to evade detection or reestablish a compromised server. The column "C&C servers contacted" of Table 1 shows how many times a C&C server used by a bot was changed. Surprisingly, many bots change C&C servers very infrequently; for example, the various copies of Rustock and Srizbi bots have used the same C&C IP address for 164 and 51 days, respectively, and experienced no downtime during these periods. Some bots are distributed as a set of binaries, each with different hardcoded C&C information. For example, we found 20 variants of Srizbi, each using one hardcoded C&C IP address. The C&C changes are often confined to a particular subnet; the 10 most active /16 subnets contributed 103 (57%) of all C&C botnet servers we've seen. As well, although none of the botnets shared a C&C server, we found multiple overlaps in the corresponding subnets; one subnet (208.72.*.*) provided C&C servers for Srizbi, Rustock, Pushdo, and MegaD, suggesting infrastructural ties across different botnets.

As it turns out, these botnets had many of their C&C servers hosted by McColo, a US based hosting provider. On November 11, McColo was taken offline by its ISPs, and as a result, the amount of spam reaching the University of Washington dropped by almost 60%. As of February 2009, the amount of spam reaching us has steadily increased to around 80% of the pre-shutdown levels as some of the botnet operators have been able to redirect their bots to new C&C servers, and in addition, new botnets have sprung up to replace the old ones.

4.1.2 Outgoing Spam Feeds

The spam generated by our botnets is a rich source of information regarding their malicious activities. The content of the spam emails can be used to identify the scams perpetrated by the botnets (as discussed in Section 4.3) and help develop application-level defenses for end-hosts (see Section 5). In this section, we analyze the characteristics of the spam mailing lists, discuss the reach of various botnets, and examine whether spam subjects could be used as fingerprints for the botnets.

Size of mailing lists: We first use the outgoing spam feeds to estimate the size of the botnets' recipient lists. We assume the following model of botnet behavior:

- A bot periodically obtains a new chunk of recipients from the master and sends spam to this recipient list. Let c be the chunk size.
- On each such request, the chunk of recipients is selected uniformly at random from the spam list.
- The chunk of recipients received by a bot is much smaller than the spam list size N.

Assuming these are true, the probability of a particular email address from the spamlist appearing in k chunks of recipients obtained by a bot is $1 - (1 - c/N)^k$. As the second term decays with k, the spam feed will expose the entire recipient list in an asymptotic manner, and eventually most newly-picked addresses will be duplicates of previous picks. Further, if we recorded the first m recipient addresses from a spam trace, the expected number of repetitions of these addresses within the next k chunks is $m[1 - (1 - c/N)^k]$.

We have observed that MegaD, Rustock, Kraken, and Storm follow this model. We fit the rates at which they see duplicates in their recipient lists into the model above to obtain their approximate spam list sizes. We present the size estimates at a confidence level of 95%. We estimate MegaD's spam list size to be *850 million addresses* (±0.2%), Rustock's to be *1.2 billion* (±3%), Kraken's to be *350 million* (±0.3%), and Storm's *110 million* (±6%).

Srizbi and Pushdo partition their spam lists in a way that precludes the above analysis. We have not yet collected enough data for Grum to reliably estimate its spam list size – our bot has not sent enough emails to see duplicate recipient email addresses.

	MegaD	Kraken	Rustock
Kraken	28%	N/A	7%
MegaD	N/A	8%	9%
Pushdo	0%	0%	0%
Rustock	15%	6%	N/A
Srizbi	21%	10%	8%
Storm	24%	11%	7%

Table 2: **Overlap between recipient spam lists.** The table shows the fraction of each botnet's recipient list that is shared with MegaD, Kraken, and Rustock's recipient lists. For example, Kraken shares 28% of its recipient list with MegaD.

Overlap in mailing lists: We also examined whether botnets systematically share parts of their spam lists. To do this, we have measured address overlap in outgoing spam feeds collected thus far and combined it with modeling similar to that in the previous section (more details are available in [16]). We found that overlaps are surprisingly small: the highest overlap is between Kraken and MegaD, which share 28% of their mailing lists. It appears different botnets cover different partitions of the global email list. Thus, spammers can benefit from using multiple botnets to get wider reach, a behavior that we in fact observe and discuss in Section 4.3.

Spam subjects: Botnets carefully design and hand-tune custom spam subjects to defeat spam filters and attract attention. We have found that between any two spam botnets, there is *no* overlap in subjects sent within a given day, and an average overlap of 0.3% during the length of our study. This suggests that subjects are useful for classifying spam messages as being sent by a particular botnet. To apply subject-based classification, we remove any overlapping subjects, leaving, on average, 489 subjects per botnet on a given day. As well, a small number of subjects include usernames or random message IDs. We remove these elements and replace them with regexps using an algorithm similar to AutoRE [39]. We will evaluate and validate this classification scheme using our

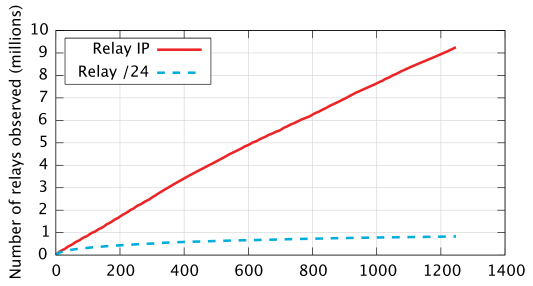


Figure 2: **Number of distinct relay IPs and the /24s containing them.**

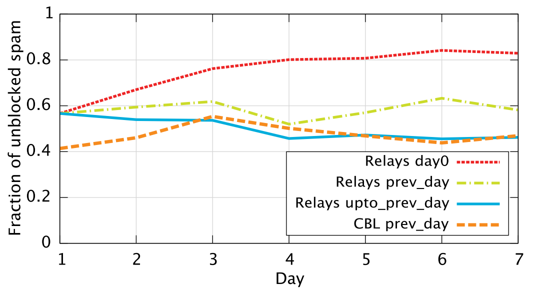


Figure 3: **Fraction of spam that is captured by using IP-based blacklists.** We find that using relays seen locally so far works as well as a commercial blacklist, and can block almost 60% of the spam.

incoming spam in Section 4.3.1.

4.2 Analysis of Incoming Spam

We analyze 46 million spam messages obtained from a 50-day trace of spam from University of Washington and use it to characterize the hosts sending the spam, the scam campaigns propagated using spam, and the web hosting infrastructure for the scams. To do this, each spam message is analyzed to extract the set of relays through which the purported sender forwarded the email, the subject, the recipient address, other SMTP headers present in the email, and the various URLs embedded inside the spam body.

We found that on average, 89.2% of the incoming mail at UW is classified as spam by UW's filtering systems. Around 0.5% of spam contain viruses as attachments. Around 95% of the spam messages contain HTTP links, and 1% contain links to executables.

4.2.1 Spam sources

Figure 2 plots the total number of distinct last-hop relays seen in spam messages over time. We consider only the IP of the last relay used before a message reaches UW's mail servers, as senders can spoof other relays. The number of distinct relay IPs increases steadily over time and reaches 9.5 million after 7 weeks worth of spam messages. Two factors could be responsible for keeping this

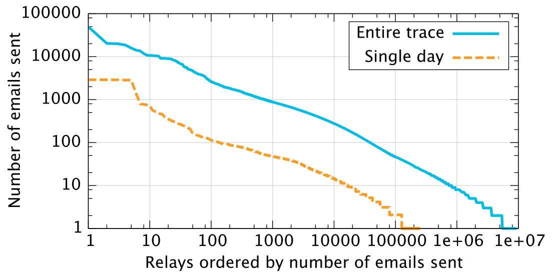


Figure 4: **Number of messages sourced by distinct relay IPs, over a single day and the entire trace.**

growth linear. One is a constant balance between the influx of newly-infected bots and the disappearance of disinfected hosts. Another is the use of dynamic IP (DHCP) leases for end hosts, which causes the same physical machine to manifest itself under different IPs. To place a lower bound on the number of distinct spam *hosts* given the DHCP effect, Figure 2 also shows the number of distinct /24's corresponding to spam relays, assuming that the IPs assigned by DHCP to a particular host stay in the /24 range.

The constantly changing list of IPs relaying spam does suggest that simple IP-based blacklists, such as the Spamhaus blacklist [31], will not be very effective at identifying spam. To understand the extent to which this churn impacts the effectiveness of IP-based blacklists, we analyze four strategies for generating blacklists and measure their ability to filter spam. First, we consider a blacklist comprising of the IP addresses of the relays which sent us spam a week ago. Next, we have a blacklist that is made up of the IP addresses of the relays which sent us spam the previous day. Third, we consider a blacklist that contains the IP addresses of the relays which sent us spam at any point in the past. Finally, we look at a commonly used blacklist such as the Composite Blocking List (CBL) [4]. Figure 3 shows the comparison. The first line shows how quickly the effectiveness of a blacklist drops with time, with a week-old blacklist blocking only 20% of the spam. Using the relay IPs from the previous day blocks around 40% of the spam, and using the entire week's relay IPs can decrease the volume of spam by $50-60\%$. Finally, we see that a commercial blacklist performs roughly as well as the local blacklist which uses a weeks' worth of information. We view these as preliminary results since a rigorous evaluation of the effectiveness of blacklists is possible only if we can also quantify the false positive rates. We defer such analysis to future work.

We next analyze the distribution of the number of messages sent by each spam relay. Figure 4 graphs the number of messages each distinct relay has sent during our trace. We also show the number of messages sent by each relay on a particular day, where DHCP effects are

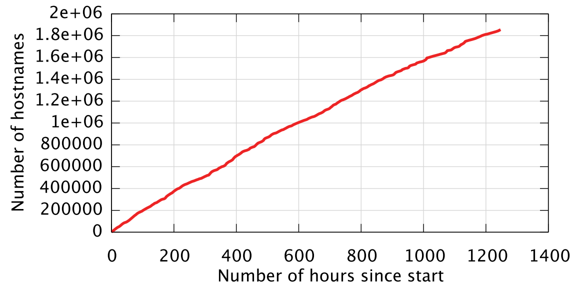


Figure 5: **Number of distinct hostnames in URLs conveyed by spam.** Spammers constantly register new DNS names.

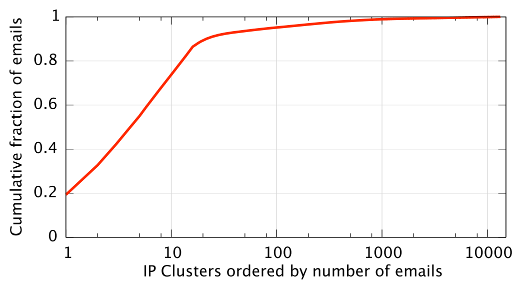


Figure 6: **Clustering spam messages by the IP of URLs contained within them.** Links in 80% of spam point to only 15 distinct IP clusters.

less likely to be manifested. On any given day, only a few tens of relays send more than 1,000 spam messages, with the bulk of the spam conveyed by the long tail. In fact, the relays that sent over 100 messages account for only 10% of the spam, and the median number of spam messages per relay is 6. One could classify the heavy hitters as either well-known open mail relays or heavily provisioned spam pumps operated by miscreants. We conjecture that most of the long tail corresponds to compromised machines running various kinds of bots.

4.2.2 Spam campaigns and Web hosting

We next examine whether we can identify and characterize individual spam campaigns based on our incoming spam. Ideally, we would cluster messages based on similar content; however, this is difficult as spammers use sophisticated content obfuscation to evade spam detection. Fortunately, more than 95% of spam in our feed contains links. We thus cluster spam based on the following attributes: 1) the domain names appearing in the URLs found in spam, 2) the content of Web pages linked to by the URLs, and 3) the resolved IP addresses of the machines hosting this content. We find that the second attribute is the most useful for characterizing campaigns.

Clustering with URL domain names revealed that for any particular day, 10% of the observed domain names account for 90% of the spam. By plotting the number of distinct domain names observed in our spam feed over time (shown in Figure 5), we found that the number of distinct hostnames is large and increases steadily, as spammers typically use newly-registered domains. (In fact, on average, domain names appearing in our spam are only two weeks old based on `whois` data.) Consequently, domain-based clustering is too fine-grained to reveal the true extent of botnet infections.

Our content clustering is performed by fetching the Web page content of all links seen in our incoming spam. We found that nearly 80% of spam pointed to just 11 distinct Web pages, and the content of these pages did not change during our study. We conclude that while spammers try to obfuscate the content of messages they send out, the Web pages being advertised are static. Although this clustering can identify distinct campaigns, it cannot accurately attribute them to specific botnets. We revisit this clustering method in Section 4.3.2, where we add information about our botnets' outgoing spam.

For IP-based clustering, we analyzed spam messages collected during the last week of our trace. We extracted hostnames from all spam URLs and performed DNS lookups on them. We then collected sets of resolved IPs from each lookup, merging any sets sharing a common IP. Finally, we grouped spam messages based on these IP clusters; Figure 6 shows the result. We found that 80% of the spam corresponds to the top 15 IP clusters (containing a total of 57 IPs). In some cases, the same Web server varied content based on the domain name that was used to access it. For example, a single server in Korea hosted 20 different portals, with demultiplexing performed using the domain name. We conjecture that such Web hosting services are simultaneously supporting a number of different spam campaigns. As a consequence, web-server-based clustering is too coarse-grained to disambiguate individual botnets.

4.3 Correlation Analyses

We now bring together two of our data sources, our outgoing and incoming spam feeds, and perform various kinds of correlation analyses, including: 1) classifying spam according to which botnet sourced it, 2) identifying spam campaigns and analyzing botnet partitioning, 3) classifying and analyzing spam used for recruiting new victims, and 4) estimating botnet sizes and producing botnet membership lists. Note that we exclude Grum from these analyses, because we have not yet monitored this recently discovered bot for a sufficiently long time.

4.3.1 Spam classification

To classify each spam message received by University of Washington as coming from a particular botnet, we use subject-based signatures we derived in Section 4.1.2. Each signature is *dynamic* — it changes whenever botnets change their outgoing spam. We have applied these

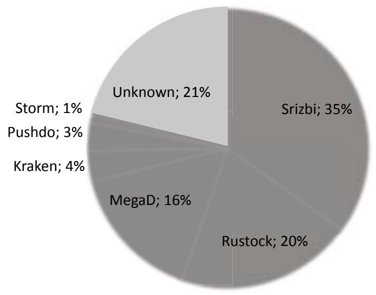


Figure 7: **Average contributions of each botnet to incoming spam received at University of Washington.** 79% of spam comes from six spam botnets, and 35% comes from just one botnet, Srizbi.

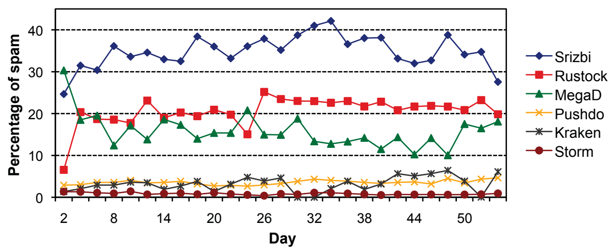


Figure 8: **Breakdown of spam e-mails by botnet over time.** Most botnets contribute approximately the same fraction of spam to our feed over our study period, with Srizbi, Rustock, and MegaD being the top contributors. Kraken shows gaps in activity on days 28-32 and 52. Day 1 corresponds to March 13, 2008.

signatures to a 50-day trace of incoming spam messages received at University of Washington in March and April 2008. Figure 7 shows how much each botnet contributed to UW spam on average, and Figure 8 shows how the breakdown behaves over time. We find that on average, our six botnets were responsible for 79% of UW's incoming spam. This is a key observation: it appears that for spam botnets, only a handful of major botnets produce most of today's spam. In fact, 35% of all spam is produced by just *one* botnet, Srizbi. This result might seem to contradict the lower bound provided by Xie et al. [39], who estimated that 16 – 18% of the spam in their dataset came from botnets. However, their dataset excludes spam sent from blacklisted IPs, and a large fraction of botnet IPs are present in various blacklists (as shown in Sections 4.2 and [28]).

We took a few steps to verify our correlation. First, we devised an alternate classification based on certain unique characteristics of the "Message ID" SMTP header for Srizbi and MegaD bots, and verified that the classification does not change using that scheme. Second, we extracted last-hop relays from each classified message and checked overlaps between sets of botnet relays. The

	Kraken	MegaD	Pushdo	Rustock	Srizbi	Storm
Canadian Healthcare	0%	0%	0.01%	22%	3%	0%
Canadian Pharmacy	16%	28%	10%	0%	9%	6%
Diamond Watches	22%	0.1%	0%	0%	13%	0%
Downloadable Software	0%	0%	25%	0%	0%	0%
Freedom From Debt Forever!	19%	0%	0%	0%	0%	1%
Golden Gate Casino	0%	32%	0%	0%	0%	0%
KING REPLICA	0%	4%	3%	0%	15%	0%
LNHSolutions	0%	6%	0%	0%	0%	0%
MaxGain+ ... No.1 for PE	0%	0%	3%	78%	0%	0%
Prestige Replicas	7%	0%	0.3%	0%	31%	0%
VPXL - PE Made Easy	20%	8%	6%	0%	24%	55%
Unavailable	3%	22%	38%	0%	0%	24%
Other	13%	0.1%	15%	0%	5%	14%

Table 3: **Clustering incoming spam by the title of the web page pointed to by spam URLs.** The columns show how frequently each botnet was sending each campaign on April 30, 2008. Many botnets carry out multiple campaigns simultaneously.

	Kraken	MegaD	Pushdo	Rustock	Srizbi	Storm
Kraken	N/A	32%	16%	10%	13%	28%
MegaD	32%	N/A	20%	8%	21%	40%
Pushdo	16%	20%	N/A	3%	14%	19%
Rustock	10%	8%	3%	N/A	7%	6%
Srizbi	13%	21%	14%	7%	N/A	15%
Storm	28%	40%	19%	6%	15%	N/A

Table 4: **Overlap in hosting infrastructure of the web pages pointed to by spam URLs.** The table shows what fraction of spam sent by different botnets on April 30, 2008 contain URLs pointing to the same webservers.

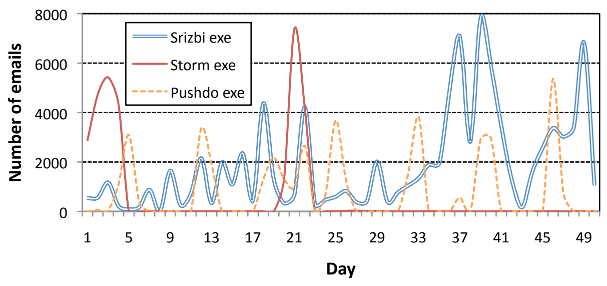


Figure 9: **Propagation campaigns.** The graph shows the number of e-mails with links that infected victims with either Srizbi, Storm, or Pushdo.

overlaps are small; it is never the case that many of the relays belonging to botnet X are also in the set of relays for botnet Y. The biggest overlap was 3.3% between Kraken and MegaD, which we interpret as 3.3% of Kraken's relays also being infected with the MegaD bot.

4.3.2 Spam campaigns

To gain insight into kinds of information spammers disseminate, we classified our incoming spam according to spam campaigns. We differentiate each campaign by the contents of the web pages pointed to by links in spam messages. Using data from Section 4.3.1, we classify our incoming spam according to botnets, and then break down each botnet's messages into campaign topics, defined by titles of campaign web pages. Table 3 shows these results for a single day of our trace. For example, Rustock participated in two campaigns – 22% of its messages advertised "Canadian Healthcare", while 78% advertised "MaxGain+". We could not fetch some links because of failed DNS resolution or inaccessible web servers; we marked these as "Unavailable". The table only shows the most prevalent campaigns; a group of less common campaigns is shown in row marked "Other".

All of our botnets simultaneously participate in multiple campaigns. For example, Kraken and Pushdo participate in at least 5 and 7, respectively. The percentages give insight into how the botnet divides its bots across various campaigns. For example, Kraken might have four customers who each pay to use approximately 20% of the botnet to send spam for "Canadian Pharmacy", "Diamond Watches", "Freedom from Debt", and "VPXL". Multiple botnets often simultaneously participate in a single campaign, contrary to an assumption made by prior research [40]. For example, "Canadian Pharmacy" is distributed by Kraken, MegaD, Pushdo, Srizbi, and Storm. This suggests the most prominent spammers utilize the services of multiple botnets.

Botnets use different methods to assign their bots to campaigns. For example, Botlab monitors 20 variants of Srizbi, each using a distinct C&C server. Each C&C server manages a set of campaigns, but these sets often differ across C&C servers. For example, bots using C&C server X and Y might send out "Canadian Pharmacy" (with messages in different languages), whereas server Z divides bots across "Prestige Replicas" and "Diamond Watches". Thus, Srizbi bots are partitioned *statically* across 20 C&C servers, and then *dynamically* within each server. In contrast, all of our variants of Rustock contact the same C&C server, which dynamically schedules bots to cover each Rustock campaign with a certain fraction of the botnet's overall processing power, behaving much like a lottery scheduler [35].

In Section 4.2.2, we saw that most of the Web servers support multiple spam campaigns. Now, we examine whether the Web hosting is tied to particular botnets, i.e., whether all the spam campaigns hosted on a server are sent by the same botnet. In Table 4, we see that this is not the case – every pair of botnets shares some hosting infrastructure. This suggests that scam hosting is more of a 3^{rd} party service that is used by multiple (potentially competing) botnets.

Botnet	Kraken	MegaD	Pushdo	Rustock	Srizbi	Storm
# unique relays seen	20,275	57,402	27,266	83,836	119,604	7,814

Table 5: **The number of unique relays belonging to each botnet.** These numbers provide a lower bound on the size of each botnet, as seen on September 3, 2008. More accurate estimates are possible by accounting for relays not seen in our spam feed.

4.3.3 Recruiting campaigns

Using our correlation tools, we were able to identify incoming spam messages containing links to executables infecting victims with the Storm, Pushdo, or Srizbi bot. Figure 9 shows this activity over our incoming spam trace. The peaks represent campaigns launched by botnets to recruit new victims. We have observed two such campaigns for Storm – one for March 13-16 and another centered on April 1, corresponding to Storm launching an April Fool's day campaign, which received wide coverage in the news [23]. Srizbi appears to have a steady ongoing recruiting campaign, with peaks around April 15-20, 2008. Pushdo infects its victims in bursts, with a new set of recruiting messages being sent out once a week.

We expected the spikes to translate to an increase in number of messages sent by either Srizbi, Storm, or Pushdo, but surprisingly this was not the case, as seen by matching Figures 9 and 8. This suggests that bot operators do not assign all available bots to send spam at maximum possible rates, but rather limit the overall spam volume sent out by the whole botnet.

4.3.4 Botnet membership lists and sizes

A botnet's power and prowess is frequently measured by the botnet's size, which we define as the number of active hosts under the botnet's control. A list of individual nodes comprising a botnet is also useful for notifying and disinfecting the victims. We next show how Botlab can be used to obtain information on both botnet size and membership.

As before, we classify our incoming spam into sourcing botnets and extract the last-hop relays from all successfully classified messages. After removing duplicates, these relay lists identify hosts belonging to each botnet. Table 5 shows the number of unique, classified relays for a particular day of our trace. Since botnet membership is highly dynamic, we perform our calculations for a single day, where churn can be assumed to be negligible. As well, we assume DHCP does not affect the set of unique relays on a timescale of just a single day. These numbers of relays for each botnet are effectively the lower bound on the botnet sizes. The actual botnet sizes are higher, since there are relays that did not send spam to University of Washington, and thus were not seen in our trace. We next estimate the percentage of total relays that we do see, and use it to better estimate botnet sizes.

Let us assume again that a bot sends spam to email addresses chosen at random. Further, let p be the probability with which a spam message with a randomly chosen email address is received by our spam monitoring system at University of Washington. If n is the number of messages that a bot sends out per day, then the probability that at least one of the messages generated by the bot is received by our spam monitors is $[1 - (1 - p)^n]$. For large values of n, such as when $n \sim 1/p$, the probability of seeing one of the bot's messages can be expressed as $[1 - e^{-np}]$.

We derive p using the number of spam messages received by our spam monitor and an estimate of the global number of spam messages. With our current setup, the former is approximately 2.4 million daily messages, while various sources estimate the latter at 100-120 billion messages (we use 110 billion) [14, 21, 32]. This gives $p = 2400000/110$ billion $= 2.2 \cdot 10^{-5}$.

For the next step, we will describe the logic using one of the botnets, Rustock, and later generalize to other botnets. From Section 4.1, we know that Rustock sends spam messages at a constant rate of $47.5K$ messages per day and that this rate is independent of the access link capacity of the host. Note that Rustock's sending rate translates to a modest rate of 1 spam message per two seconds, or about 0.35 KB/s given that the average Rustock message size is 700 bytes – a rate that can be supported by almost all end-hosts [15]. The probability that we see the IP of a Rustock spamming bot node in our spam monitors on a given day is $[1 - e^{-47500 \cdot 2.2 \cdot 10^{-5}}] = 0.65$. This implies that the 83,836 Rustock IPs we saw on September 3rd represent about 65% of all Rustock's relays; thus, the total number of active Rustock bots on that day was about $83,836/0.65 = 128,978$. Similarly, we estimate the active botnet size of Storm to be 16,750. We would like to point that these estimates conservatively assume a bot stays active 24 hours per day. Because some bots are powered off during the night, these botnet sizes are likely to be higher.

These estimates rely on the fact that both Rustock and Storm send messages at a slow, constant rate that is unlikely to saturate most clients' bandwidth. Our other bots send spam at higher rates, with the bot adapting to the host's available bandwidth. Although this makes it more likely that a spamming relay is detected in our incoming spam, it is also more difficult to estimate the number of messages a given bot sends. In future work, we plan to study the rate adaptation behavior of these bots and combine it with known bandwidth profiles [15]. Meanwhile, Table 5 gives conservative size estimates.

5 Applications enabled by Botlab

Botlab provides various kinds of real-time botnet information, which can be used by end hosts wishing for protection against botnets, or by ISPs and activists for law enforcement. Next, we discuss applications enabled by our monitoring infrastructure.

5.1 Safer web browsing

Spam botnets propagate many harmful links, such as links to phishing sites or to web pages installing malware. For example, on September 24, 2008, we observed the Srizbi botnet distribute 40,270 distinct links to pages exploiting Flash to install the Srizbi bot. Although the current spam filtering tools are expected to filter out spam messages containing these links, we found that this is often not the case. For example, we have forwarded a representative sample of each of Srizbi's outgoing spam campaigns to a newly-created Gmail account controlled by us, where we have used Gmail's default spam filtering rules, and found that 79% of spam was *not* filtered out. Worse, Gmail filters are not "improving" quickly enough, as forwarding the same e-mails two days later resulted in only a 5% improvement in detection. Users are thus exposed to many messages containing dangerous links and social engineering traps enticing users to click on them.

Botlab can protect users from such attacks using its real-time database of malicious links seen in outgoing, botnet-generated spam. For example, we have developed a Firefox extension, which checks the links a user visits against this database before navigating to them. In this way, the extension easily prevented users from browsing to any of the malicious links Srizbi sent on September 24.

Some existing defenses use blacklisting to prevent browsers from following malicious links. We have checked two such blacklists, the Google Safe Browser API and the Malware Domain List, six days after the links were sent out, and found that *none* of the 40,270 links appeared on either list. These lists suffer from the same problem: they are *reactive*, as they rely on crawling and user reports to find malicious links *after* they are disseminated. These methods fail to quickly and exhaustively find "zero-day" botnet links, which point to malware hosted on recently compromised web servers, as well as malware hosted on individual bots via fast-flux DNS and a continuous flow of freshly-registered domain names. In contrast, Botlab can keep up with spam botnets because it uses *real-time blacklists*, which are updated with links at the instant they are disseminated by botnets.

5.2 Spam Filtering

Spam continuously pollutes email inboxes of many millions of Internet users. Most email users use spam filtering software such as SpamAssassin [30], which uses heuristics-based filters to determine whether a given message is spam. The filters usually have a threshold that a user varies to catch most spam while minimizing the number of false positives — legitimate email messages misclassified as spam. Often, this still leaves some spam sneaking through.

Botlab's real-time information can be used to build better spam filters. Specifically, using Botlab, we can determine whether a message is spam by checking it against the outgoing spam feeds for the botnets we monitor. This is a powerful mechanism: we simply rely on botnets themselves to tell us which messages are spam.

We implemented this idea in an extension for the Thunderbird email client. The extension checks messages arriving to the user's inbox against Botlab's live feeds using a simple, proof-of-concept correlation algorithm: an incoming message comes from a botnet if 1) there is an exact match on the set of URLs contained in the message body, or 2) if the message headers are in a format specific to that used by a particular botnet. For example, all of Srizbi's messages follow the same unique message ID and date format, distinct from all other legitimate and spam email. Although the second check is prone to future circumvention, this algorithm gives us an opportunity to pre-evaluate the potential of this technique. Recent work has proposed more robust algorithms, such as automatic regular-expression generation for spammed URLs in AutoRE [39], and we envision adopting these algorithms to use Botlab data to filter spam more effectively in real-time settings.

Although we have not yet thoroughly evaluated our extension, we performed a short experiment to estimate its effectiveness. One author used the extension for a week, and found that it reduced the amount of spam bypassing his departmental SpamAssassin filters by 156 messages, or 76%, while having a 0% false positive rate. Thus, we believe that Botlab can indeed significantly improve today's spam filtering tools.

5.3 Availability of Botlab Data

To make Botlab's data publicly available, we have set up a web page, `http://botlab.cs.washington.edu/`, which publishes data and statistics we obtained from Botlab. The information we provide currently includes activity reports for each spam botnet we monitor, ongoing scams, and a database of rogue links disseminated in spam. We also publish lists of current C&C addresses and members of individual botnets. We hope this information will further aid security researchers and activists in the continuing fight against the botnet threat.

6 Safety

We have implemented safeguards to ensure that Botlab never harms remote hosts, networks, or users. In this section, we discuss the impact of these safeguards on the effectiveness of Botlab, and our concerns over the long-term viability of safely conducting bot research.

6.1 Impact on effectiveness

Initially, we hoped to construct a fully automatic platform that required no manual analysis on the part of an operator to find and analyze new botnet binaries. We quickly concluded this would be infeasible to do safely, as human judgment and analysis is needed to determine whether previously uncharacterized traffic is safe to transmit.

Even with a human in the loop, safety concerns caused us to make choices that limit the effectiveness of our research. Our network sandbox mechanisms likely prevented some binaries from successfully communicating with C&C servers and activating, causing us to fail to recognize some binaries as spambots, and therefore to underestimate the diversity and extent of spamming botnets. Similarly, it is possible that some of our captive bot nodes periodically perform an end-to-end check of e-mail reachability, and that our spamhole blocking mechanism causes these nodes to disable themselves or behave differently than they would in the wild.

6.2 The long-term viability of safe botnet research

The only provably safe way for Botlab to execute untrusted code is to block all network traffic, but this would render Botlab ineffective. To date, our safeguards have let us analyze bot binaries while being confident that we have not caused harm. However, botnet trends and thought experiments have diminished our confidence that we can continue to conduct our research safely.

Botnets are trending towards the use of proprietary encrypted protocols to defeat analysis, polymorphism to evade detection, and automatically upgrading to new variants to incorporate new mechanisms. It is hard to understand the impact of allowing an encrypted packet to be transmitted, or to ensure that traffic patterns that were benign do not become harmful after a binary evolves. Accordingly, the risk of letting any network traffic out of a captured bot node seems to be growing.

Simple thought experiments show that it is possible for an adversary to construct a bot binary for which there is no safe and effective Botlab sandboxing policy. As an extreme example, consider a hypothetical botnet whose C&C protocol consists of different attack packets. If a message is sent to an existing member of the botnet, the message will be intercepted and interpreted by the bot. However, if a message is sent to a non-botnet host, the message could exploit a vulnerability on that host. If such a protocol were adopted, Botlab could not transmit any messages safely, since Botlab would not know whether a destination IP address is an existing bot node. Other adversarial strategies are possible, such as embedding a time bomb within a bot node, or causing a bot node to send benign traffic that, when aggregated across thousands of nodes, results in a DDoS attack. Moreover, even transmitting a "benign" C&C message could cause other, non-Botlab bot nodes to transmit harmful traffic.

Given these concerns, we have disabled the crawling and network fingerprinting aspects of Botlab, and therefore are no longer analyzing or incorporating new binaries. As well, the only network traffic we are letting out of our existing botnet binaries are packets destined for the current, single C&C server IP address associated with each binary. Since Storm communicates with many peers over random ports, we have stopped analyzing Storm. Furthermore, once the C&C servers for the other botnets move, we will no longer allow outgoing network packets from their bot binaries. Consequently, the Botlab web site will no longer be updated with bots that we have to disable. It will, however, still provide access to all the data we have collected so far.

Our future research must therefore focus on deriving analysis techniques that do not require bot nodes to interact with Internet hosts, and determining if it is possible to construct additional safeguards that will sufficiently increase our confidence in the safety of transmitting specific packets. Unfortunately, our instinct is that a motivated adversary can make it impossible to conduct effective botnet research in a safe manner.

7 Related Work

Most related work can be classified into four categories: malware collection, malware analysis, botnet tracking systems, and spam measurement studies. We now discuss how our work relates to representative efforts in each of these categories.

Malware collection: Honeypots (such as Honeynet [13] and Potemkin [34]) have been a rich source of new malware samples. However, we found them less relevant for our work, as they failed to find any spam bots. The likely cause is that spam botnets have shifted to social-engineering-based propagation, relying less on service exploits and self-propagating worms. Other projects, such as HoneyMonkey [38], have used automated web crawling to discover and analyze malware; automated web patrol is now part of Google's infrastructure [24]. However, our results show that Google's database did not contain many malicious links seen in our outgoing spam feed, indicating that a blind crawl will not find malware from spam-distributed links.

Malware analysis: Botlab does not perform any static analyses of malware binaries. Instead, it generates network fingerprints by executing the binaries and observing network accesses. [2] and [37] perform similar dynamic analysis of binaries by executing them in virtual environments and tracking changes in the system, such as the creation of files, processes, registry entries, etc. [2] uses this information to group malware into broad categories, and [37] generates a detailed report of the malware's actions. Since these techniques require running the binary in an instrumented setting (such as a debugger), they would not be able to analyze malware which performs VMM or debugger detection. More similar to our approach is [27], which generates network fingerprints and uses them to detect IRC bots.

Botnet tracking: Closely related to our work is the use of virtualized execution environments to track IRC botnets [27, 41]. By executing a large number of IRC bot samples, these efforts first identify the IRC servers and then infiltrate the corresponding IRC channels to snoop on the botnets. In our experience, botnets move away from plaintext IRC protocols to encrypted HTTP-based or p2p protocols, requiring more elaborate mechanisms as well as human involvement for a successful infiltration – a point of view that is increasingly voiced in the research community [18].For example, Ramachandran et al. [28] infiltrated the Bobax botnet by hijacking the authoritative DNS for the domain running the C&C server for the botnet. They were then able to obtain packet traces from the bots which attempted to connect to their C&C server. More recently, Kanich et al. [17] infiltrated the command and control infrastructure of the Storm botnet, and modified the spam being sent in order to measure the conversion rates.

Less related to our work is the research on developing generic tools that can be deployed at the network layer to automatically detect the presence of bots [19]. Rishi [8] is a tool that detects the use of IRC commands and uncommon server ports in order to identify compromised hosts. BotSniffer [11] and BotHunter [10] are other network-based anomaly detection tools that work by simply sniffing on the network. Our work provides a different perspective on bot detection: a single large institution, such as University of Washington, can detect most of the spamming bots operating at a given point in time by simply examining its incoming spam feed and correlating it with the outgoing spam of known bots.

Spam measurement studies: Recently, a number of studies have examined incoming spam feeds to understand botnet behavior and the scam hosting infrastructure [1, 40, 39]. In [26], the authors use a novel approach to collecting spam – by advertising open mail forwarding relays, and then collecting the spam that is sent through them. Botlab differs from these efforts in its use of *both* incoming and outgoing spam feeds. In addition to enabling application-level defenses that are proactive as opposed to reactive, our approach yields a more comprehensive view of spamming botnets that contradicts some assumptions and observations from prior work. For instance, a recent study [40] analyzes about 5 million messages and proposes novel clustering techniques to identify spam messages sent by the same botnet, but this is done under the assumption that each spam campaign is sourced by a single botnet; we observe the contrary to be true. Also, analysis of only the incoming spam feed might result in too fine-grained a view (at the level of short-term spam campaigns as in [39]) and cannot track the longitudinal behavior of botnets. Our work enables such analysis due to its use of live bots, and in that respect, we share some commonality with the recent study of live Storm bots and their spamming behavior [20].

8 Conclusion

In this work, we have described Botlab, a real-time botnet monitoring system. Botlab's key aspect is a multi-perspective design that combines a feed of incoming spam from the University of Washington with a feed of outgoing spam collected by running live bot binaries. By correlating these feeds, Botlab can perform a more comprehensive, accurate, and timely analysis of spam botnets.

We have used Botlab to discover and analyze today's most prominent spam botnets. We found that just six botnets are responsible for 79% of our university's spam. While domain names associated with the scams change frequently, the locations of C&C servers, web hosts, and even the content of web pages pointed to by scams remain static for long periods of time. A spam botnet typically engages in multiple spam campaigns simultaneously, and the same campaign is often purveyed by multiple botnets. We have also prototyped tools that use Botlab's real-time information to enable safer browsing and better spam filtering. Overall, we feel Botlab advances our understanding of botnets and enables promising research in anti-botnet defenses.

Acknowledgments

We gratefully acknowledge Farnam Jahanian for guiding us through the shepherding process. We also would like to thank the anonymous NSDI reviewers for their valuable feedback on the submitted version. We are grateful to James Morris for his help with spam feeds, and Dave Dittrich for his comments and feedback on the paper. This research was partially supported by the National Science Foundation under Grants CNS-0831540, CNS-0627367, and CNS-0519696.

References

[1] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker. Spamscatter: Characterizing Internet Scam Hosting Infrastructure. In *USENIX Security*, 2007.

[2] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *RAID*, pages 178–197, 2007.

[3] Castle Cops. `http://www.castlecops.com`.

[4] Composite Block List. `http://cbl.abuseat.org/`.

[5] K. Chiang and L. Lloyd. A Case Study of the Rustock Rootkit and Spam Bot. In *Proc. of the First Workshop on Hot Topics in Understanding Botnets*, 2007.

[6] R. Dingledine, N. Mathewson, and P. Syverson. Tor: the second-generation onion router. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, page 21, Berkeley, CA, USA, 2004. USENIX Association.

[7] D. Dittrich and S. Dietrich. Command and control structures in malware: From Handler/Agent to P2P. *USENIX ;login:*, 2007.

[8] J. Goebel and T. Holz. Rishi: identify bot contaminated hosts by IRC nickname evaluation. In *Proc. of the First Workshop on Hot Topics in Understanding Botnets*, 2007.

[9] J. B. Grizzard, V. Sharma, C. Nunnery, B. B. Kang, and D. Dagon. Peer-to-Peer Botnets: Overview and Case Study. In *Proc. of the First Workshop on Hot Topics in Understanding Botnets*, 2007.

[10] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee. BotHunter: Detecting malware infection through ids-driven dialog correlation. In *USENIX Security*, August 2007.

[11] G. Gu, J. Zhang, and W. Lee. BotSniffer: Detecting Botnet Command and Control Channels in Network Traffic. In *NDSS*, 2008.

[12] T. Holz, C. Gorecki, K. Rieck, and F. Freiling. Measuring and Detecting Fast-Flux Service Networks. In *NDSS*, 2008.

[13] Honeynet Project. *Know your enemy*. Addison-Wesley Professional, 2004.

[14] Ironport. 2008 Internet Security Trends. `http://www.ironport.com/securitytrends/`, 2008.

[15] T. Isdal, M. Piatek, A. Krishnamurthy, and T. Anderson. Leveraging bittorrent for end host measurements. In *PAM*, 2007.

[16] J. P. John, A. Moshchuk, S. D. Gribble, and A. Krishnamurthy. Studying Spamming Botnets Using Botlab. Technical Report TR-2008-10-01, University of Washington, Computer Science and Engineering, October 2008.

[17] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. M. Voelker, V. Paxson, and S. Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 3–14, New York, NY, USA, 2008. ACM.

[18] C. Kanich, K. Levchenko, B. Enright, G. M. Voelker, and S. Savage. The Heisenbot Uncertainty Problem: Challenges in Separating Bots from Chaff. In *LEET*, 2008.

[19] A. Karasaridis, B. Rexroad, and D. Hoeflin. Wide-Scale Botnet Detection and Characterization. In *Proc. of the First Workshop on Hot Topics in Understanding Botnets*, 2007.

[20] C. Kreibich, C. Kanich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. On the Spam Campaign Trail. In *LEET*, 2008.

[21] Marshal Press Release: Srizbi now leads the spam pack. `http://www.marshal.com/trace/traceitem.asp?article=567`.

[22] MWCollect. `http://www.mwcollect.org`.

[23] J. Nazario. April Storms Day Campaign. `http://asert.arbornetworks.com/2008/03/april-storms-day-campaign/`, 2008.

[24] Niels Provos and Panayiotis Mavrommatis and Moheeb Rajab and Fabian Monrose. All Your iFrames Point to Us. In *USENIX Security Symposium*, 2008.

[25] Offensive Computing. `http://www.offensivecomputing.net`.

[26] A. Pathak, Y. C. Hu, and Z. M. Mao. Peeking into spammer behavior from a unique vantage point. In F. Monrose, editor, *LEET*. USENIX Association, 2008.

[27] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A Multifaceted Approach to Understanding the Botnet Phenomenon. In *IMC*, 2006.

[28] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *SIGCOMM*, 2006.

[29] Shadow Server. `http://www.shadowserver.org`.

[30] SpamAssassin. `http://spamassassin.apache.org`.

[31] Spamhaus. `http://www.spamhaus.org/`.

[32] Spamunit. Spam Statistics. `http://spamunit.com/spam-statistics/`.

[33] S. Stover, D. Dittrich, J. Hernandez, and S. Dietrich. Analysis of the Storm and Nugache Trojans: P2P is here. *USENIX ;login:*, 2007.

[34] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *SOSP*, 2005.

[35] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Operating Systems Design and Implementation*, pages 1–11, 1994.

[36] P. Wang, S. Sparks, and C. C. Zou. An Advanced Hybrid Peer-to-Peer Botnet. In *Proc. of the First Workshop on Hot Topics in Understanding Botnets*, 2007.

[37] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[38] Yi-Min Wang and Doug Beck and Xuxian Jiang and Roussi Roussev and Chad Verbowski and Shuo Chen and Sam King. Automated Web Patrol with Strider HoneyMonkeys. In *NDSS*, 2006.

[39] Yinglian Xie and Fang Yu and Kannan Achan and Rina Panigrahy and Geoff Hulten and Ivan Osipkov. Spamming Botnets: Signatures and Characteristics. In *SIGCOMM*, 2008.

[40] L. Zhuang, J. Dunagan, D. R. Simon, H. J. Wang, I. Osipkov, G. Hulten, and J. D. Tygar. Characterizing Botnets from Email Spam Records. In *Proc. of Workshop on Large-scale Exploits and Emergent Threats*, 2008.

[41] J. Zhuge, T. Holz, X. Han, J. Guo, and W. Zou. Characterizing the irc-based botnet phenonmenon. Technical Report TR-2007-010, Reihe Informatik, 2007.

Not-a-Bot: Improving Service Availability in the Face of Botnet Attacks

Ramakrishna Gummadi*, Hari Balakrishnan*, Petros Maniatis†, Sylvia Ratnasamy†
*MIT CSAIL, †Intel Research Berkeley

Abstract

A large fraction of email spam, distributed denial-of-service (DDoS) attacks, and click-fraud on web advertisements are caused by traffic sent from compromised machines that form botnets. This paper posits that by identifying human-generated traffic as such, one can service it with improved reliability or higher priority, mitigating the effects of botnet attacks.

The key challenge is to identify human-generated traffic in the absence of strong unique identities. We develop NAB ("Not-A-Bot"), a system to approximately identify and certify human-generated activity. NAB uses a small trusted software component called an attester, which runs on the client machine with an untrusted OS and applications. The attester tags each request with an attestation if the request is made within a small amount of time of legitimate keyboard or mouse activity. The remote entity serving the request sends the request and attestation to a verifier, which checks the attestation and implements an application-specific policy for attested requests.

Our implementation of the attester is within the Xen hypervisor. By analyzing traces of keyboard and mouse activity from 328 users at Intel, together with adversarial traces of spam, DDoS, and click-fraud activity, we estimate that NAB reduces the amount of spam that currently passes through a tuned spam filter by more than 92%, while not flagging any legitimate email as spam. NAB delivers similar benefits to legitimate requests under DDoS and click-fraud attacks.

1 Introduction

Botnets comprising compromised machines are the major originators of email spam, distributed denial-of-service (DDoS) attacks, and click-fraud on advertisement-based web sites today. By one measure, the current top six botnets alone are responsible for more than 85% of all spam mail [23], amounting to more than 120 billion messages per day that infest more than 95% of all inboxes [14, 24]. Botnet-generated DDoS attacks account for about five percent of all web traffic [9], occurring at a rate of more than 4000 distinct attacks per week on average [17]. A problem of a more recent vintage, click-fraud, is a growing threat to companies that draw revenue from web ad placements [26]; bots are said to generate 14–20% of all ad clicks today [8].

As a result, if it were possible to tag email or web requests as "human-generated," and therefore not "bot-generated," the problems of spam, DDoS, and click-fraud could be significantly mitigated. This observation is not new, but there is currently no good way to obtain such tags *automatically* without explicit human input. As explained in §4, requiring human input (say in the form of answering CAPTCHAs [30]) is either untenable (persuading users to answer a CAPTCHA before clicking on a web ad or link is unlikely to work well), or ineffective (e.g., because today the task of solving CAPTCHAs can be delegated to other machines and humans, and not inextricably linked to the request it is intended to validate).

The problem with obtaining this evidence automatically is that the client machine may have been compromised, so one cannot readily trust any information provided by software running on the compromised machine. To solve this problem, we observe that almost all commodity PCs hitting the market today are equipped with a Trusted Platform Module (TPM) [28]. We use this facility to build a trusted path between physical input devices (the keyboard and mouse, extensible in the future to devices like the microphone) and a *human activity attester*, which is a small piece of trusted software that runs isolated from the (untrusted) operating system.

The key challenge for the attester is to certify human-generated traffic without relying on strong unique identities. This paper describes NAB, a system that implements a general-purpose human activity attester (§4), and then shows how to use this attester for email to control spam, and for web requests to mitigate DDoS attacks and click fraud. Attestations are signed statements by the trusted attester, and are attached to application requests such as emails. Attestations are verified by a *verifier* module running at the server of an entity interested in knowing whether the incoming request traffic was sent as a result of human activity. If the attestation is valid (i.e., it is not forged or used before), that server can take suitable application-specific action—improving the "spam score" for an attested email message, increasing the priority of an attested web request, etc. NAB requires minor modifications to client and server applications to use attestations, and no application protocols such as SMTP or

HTTP need be modified.

NAB's philosophy is to do no harm to users who do not deploy NAB, while benefiting users who do. For example, email senders who use the attester decrease the likelihood that their emails are flagged as spam (that is, decrease the *false positives* of spam detectors), and email receivers that use the verifier see reduced spam in their inboxes. These improvements are preserved even under adversarial workloads. Further, since NAB does not use identity-based blacklisting or filtering, legitimate email from an infected machine can still be delivered with valid attestations.

The NAB approach can run on any platform that provides for the attested execution of trusted code, either directly or via a secure booting mechanism such as those supported by Intel's TXT and AMD's Pacifica architectures. We have constructed our prototype attester as host kernel model running under a trusted hypervisor. Other implementations, such as building the attester within the trusted hardware, or running it in software without virtualization (e.g., via Flicker [16]) are also possible.

Our prototype extends the Xen hypervisor [3], thus isolating itself from malicious code running within untrusted guest operating systems in a virtual machine. We stripped the host kernel and Xen Virtual Machine Monitor (VMM) down to fewer than 30,000 source lines, including the necessary device drivers, and built the attester as a 500-line kernel module. This code, together with the TPM and input devices forms the trusted computing base (TCB). Generating an attestation on a standard PC takes fewer than 10^7 CPU cycles, or less than 10 ms on a 2 GHz processor, making NAB practical for handling fine-grained attestation requests, such as individual web clicks or email messages.

We evaluate whether NAB can be applied to spam control, DDoS defense, and click-fraud detection, using a combination of datasets containing normal user activity and malicious bot activity. We used traces of keyboard and mouse activity from 328 PCs of volunteering users at Intel gathered over a one-month period in 2007 [11], packet-level traces of bot activity that we gathered from a small number of "honeypot" computers infected by malware at the same site, as well as publicly available traces of email spam and DDoS activity. On top of those traces, we constructed an adversarial workload that maximizes the attacker's benefit obtained under the constraints imposed by NAB. Our experimental study shows that:

1. With regards to spam mitigation, we reduced the volume of spam messages that evaded a traditional spam filter (what are called *false negatives* for the spam filter) by 92%. We reduced the volume of legitimate, non-spam messages that were misclassified by the spam filter (false positives) to 0.
2. With regards to web DDoS mitigation, we deprioritized 89% of bot-originated web activity without impacting human-generated web requests.
3. With regards to click-fraud mitigation, we detected bot-originating click-fraud activity with higher than 87% accuracy, without losing any human-generated web clicks.

Although our specific results correspond only to our particular traces, choice of applications, and threat model (e.g., NAB does nothing to mitigate the volume of evil traffic created manually by an evil human), we argue that they apply to a large class of on-line applications affected by bot traffic today. Those include games, brokerage, and single sign-on services. This suggests that a human activity attestation module might be a worthwhile addition to the TCB of commodity systems for the long term.

2 Threat Model and Goal

Threat model and assumptions. We assume that the OS and applications of a host cannot be trusted, and are susceptible to compromise. A host is equipped with a TPM, which boots the attester stack—this includes all software on which the attester implementation depends, such as the host kernel and VMM in our implementation (§4.4). This trust in the correct boot-up of the attester can be remotely verified, which is the standard practice for TPM-assisted secure booting today. We assume that the users of subverted hosts may be lax, but not malicious enough to mount hardware attacks against their own machine's hardware (such as shaving the protective coating off their TPM chip or building custom input hardware). We assume correct hardware, including the correct operation and protection of the TPM chip from software attacks, as per its specification [28]. We make no assumptions about what spammers do with their own hardware. Finally, we assume that the cryptographic primitives we use are secure, and that their implementations are correct.

Goal. NAB consists of an attester and a verifier. Our primary goal is to distinguish between bot and human-generated traffic at the verifier, so that the verifier can implement application-specific remedies, such as prioritizing or improving the delivery of human traffic over botnet traffic. We would like to do so without requiring any user input or imposing any cognitive burden on the user.

We aim to bound the final botnet traffic that manages to bypass any measures put up against it (spam and DDoS filters, click fraud detectors, etc.). We will consider our approach successful if we can reduce this botnet traffic that evades our best approaches today to a small fraction of its current levels ($\approx 10\%$), even in the worst case for NAB (i.e., with adaptive bots that modulate their behavior to gain the maximum benefit allowable by our mechanism), while still identifying all valid human-generated traffic correctly. We set this goal because we do not believe that purely technical approaches such as NAB will completely suppress attack traffic such as spam, since spam also relies on social engineering. We demonstrate that NAB achieves this goal with our realistic workloads and adaptive bots (§6).

3 NAB Architecture

We now present the requirements and constraints that drive the NAB architecture.

3.1 Requirements and Constraints

Requirements. There are four main requirements. First, attestations must be generated in response to human requests automatically. Second, such attestations must not be transferable from the client on which they are generated to attest traffic originating from another client. Third, NAB must benefit users that deploy it without hurting those that do not. Fourth, NAB must preserve the existing privacy and anonymity semantics of applications while delivering these benefits.

Constraints. NAB has two main constraints. First, the host's OS or applications cannot be trusted. In particular, a compromised machine can actively try to subvert the attester functionality. Second, the size of the attester TCB should be small, because it is a trusted component; the smaller a component is, the easier it is to validate it operates correctly, which makes it easier to trust.

Challenge. The key challenge is to meet these requirements without assuming the existence of globally unique identities. Even assuming a public-key infrastructure (PKI), deploying and managing large-scale identity systems that map certificates to users is a daunting problem [4].

Without such identities, the requirements are hard to meet, and, in some cases, even seemingly in conflict with each other. For example, generating attestations automatically without trusting the OS and applications is challenging. Further, there is tension between the requirement that NAB should benefit its users without hurting other users, and the requirement that NAB should preserve the existing anonymity and privacy semantics. NAB's attestations are anonymously signed certificates of requests, and the membership size of the signing keys is several million. We describe how NAB uses such attestations to overcome the absence of globally unique identities in §4.4.

TPM background. The TPM is a small chip specified by the Trusted Computing Group to strengthen the security of computer systems in general. A TPM provides many security services, among which the ability to measure and attest to the integrity of trusted software running on the computer at boot time. Since a TPM is too slow to be used routinely for cryptographic operations such as signing human activity, we use the TPM only for its secure bootstrap facilities, to load an *attester*, a small trusted software module that runs on the host processor and generates attestations (i.e., messages asserting human activity).

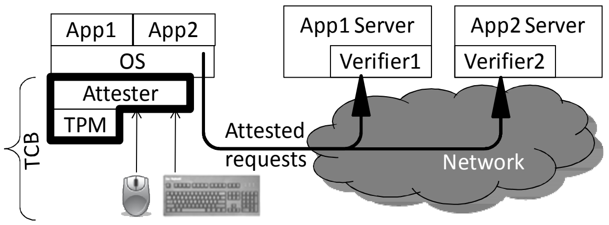


Figure 1: NAB architecture. The thick black line encloses the TCB.

The attester relies on two key primitives provided by TPMs. The first is called *direct anonymous attestation* (DAA), which allows the attester to sign messages anonymously. Each TPM has an *attestation identity key* (AIK), which is an anonymous key used to derive the attester's signing key. The second primitive is called *sealed storage*, which provides a secure location to store the attester's signing key until the attester is measured and launched correctly.

3.2 Architecture

NAB consists of an attester that runs locally at a host and generates attestations, as well as an external verifier that validates these attestations (running at a server expected to handle spam and DDoS requests, or checking for click fraud). The attester code hashes to a well-known SHA-1 value, which the TPM measures at launch. The attester then listens on the keyboard and mouse ports for human activity clicks, and decides whether an attestation should be granted to an application when the application requests one. If the attester decides to grant an attestation, the application can submit the attestation along with the application request to the verifier for human activity validation. The verifier can confirm human activity as long as it trusts the attestation TCB, which consists of the attester, the TPM, and input device hardware and drivers. This architecture is shown in Figure 1.

Attestations are signed messages with two key properties that enable the verifier to validate them correctly:

1. **Non-transferability.** An attestation generated on a machine is authenticated by a chain of signing keys that pass through that machine's TPM. Hence, a valid

attestation cannot be forged to appear as if it were issued by an attester other than its creator, and no valid attestation can be generated without the involvement of a valid attester and TPM chip.

2. **Binding to the content of a request.** An attestation contains the hash digest of the content of the request it is attesting to. Since an attester generates an attestation only in response to human activity, this binding ensures that the attestation corresponds to the content used to generate it. Binding thus allows a request to be tied as closely as practical to the user's intent to generate that request, greatly reducing opportunities for using human activity to justify unrelated requests.

4 Attester Design and Implementation

Our attester design assumes no special hardware support other than the availability of a TPM device. However, it is flexible enough to exploit the recent processor extensions for trusted computing such as AMD's Secure Virtual Machine (SVM) or Intel's Trusted Execution Technology (TXT) to provide additional features such as late launch (i.e., non boot-time launch), integration into the TCB of an OS, etc., in the future.

The attester's sole function is to generate an attestation when an application requests one. An attestation request contains only the application-specific content to attest to (e.g., the email message to send out). The attester may provide the attestation or refuse to provide an attestation at all. We discuss two important decisions: when to grant an attestation and what to attest.

4.1 When To Grant An Attestation

The key question in designing the attester is deciding under what conditions a valid attestation must be granted. The goal is to simultaneously ensure that human-generated traffic is attested, while all bot-generated traffic is denied attestation.

The attester's decision is one of guessing the human's presence and intent: was there a human operating the computer, and did she really intend to send the particular email for which the application is requesting an attestation? Since the attester lacks a direct link to the human's intentions, it must guess based on the trusted inputs available: the keyboard and mouse. We considered three key design points for such a guessing module.

The best-quality guess is not a guess at all: the attester could momentarily take over the keyboard, mouse, and display device, and prompt the user with a specific question to attest or not attest to a particular email. Since the OS and other applications are displaced in the process, only the human user can answer the question. From the interaction point of view, this approach is similar to the User Account Control (UAC) tool in Microsoft Windows Vista, in which the OS prompts the user for explicit approval before performing certain operations, although in our context it would be the much smaller and simpler attester that performs that function. While technically feasible to implement, users have traditionally found explicit prompts annoying in practice, as revealed by the negative feedback on UAC [29]. What is worse, user fatigue inevitably leads to an always-click-OK user behavior [32], which defeats the purpose of attestation.

So, we only consider guesses made automatically. In particular, we use implicit guessing of human intent, using *timing* as a good heuristic: how recently before a particular attestation request was the last keyboard or mouse activity observed? We call this a "$t - \delta$" attester, if δ_m denotes the time since the last mouse activity and δ_k denotes the time since the last keyboard activity. For example, the email application requests an attestation specifying that a keyboard or mouse click should have occurred within the last Δ_k or Δ_m milliseconds respectively, where the $\Delta_{\{k,m\}}$ represents the application-specified upper-bound. The attester generates attestations that indicate this time lag, or refuses if that lag is longer than $\Delta_{\{k,m\}}$ milliseconds.

This method is simpler and cheaper in terms of required resources than an alternative we carefully considered and eventually discarded. Using keyboard activity traces, we found that good-quality guesses can be extracted by trying to *support* the content of an attestation request using specific recent keyboard and mouse activity. For example, the attester can observe and remember a short sequential history of keystrokes and mouse clicks in order of observation. When a particular attestation request comes in, the attester searches for the longest subsequence of keyclicks that matches the content to attest. An attestation could be issued containing the quality of match (e.g., a percentage of content matched), only raising an explicit alarm and potential user prompting if that match is lower than a configurable threshold (say 60%). This design point would not attest to bot requests unless they contained significant content overlap with legitimate user traffic. Nevertheless this method raised great implementation complexity, given the typical multitasking behavior of modern users (switching between windows, interleaving keyboard and mouse activity, inserting, deleting, selecting and overwriting text, etc.). So, we ultimately discarded it in favor of the simpler $t - \delta$ attester, which allowed a simple implementation with a small TCB size.

One drawback of the $t-\delta$ attester is that it allows a bot to generate attestations for its own traffic by "harvesting" existing user activity. So, NAB could allow illegitimate traffic to receive attestations, though only at the rate of human activity.

NAB mitigates this situation in two ways. First, NAB ensures that two attestations are separated by at least the application-specific Δ milliseconds. For email, we find from the traces (§6) that $\Delta = 1$ second works well. Since key clicks cannot be captured or stored, we throttle a bot significantly in practice. Today's bots send several tens of thousands of spam within a few hours [14], so even an adaptive bot is constrained by this rate limit.

Second, if legitimate traffic fails to receive an attestation (e.g., because bot code attestation requests absorbed all recent user activity before the user's application had a chance to do so), a NAB-aware application alerts the user that it has not been able to acquire an attestation, possibly alerting the user that unwholesomeness is afoot at her computer. We note that this technique is not perfect, because a bot can hijack such prompts. In practice, we found that such feedback is useful, although we evaluate NAB assuming adversarial bots.

4.2 What To Attest

The second attester design decision is what to attest, i.e., how much to link a particular attestation to the issuer, the verifier, and the content.

Traditional human activity solutions such as CAPTCHAs [30] do not link to the actual request being satisfied. A CAPTCHA is a challenge that only a human is supposed to be able to respond to. A correct response to a CAPTCHA attests to the fact that a human was likely involved in answering the question, but it does not say where the human was or whether the answer came from the user of the service making the request. The problem is that human activity can be trafficked, as evidenced by spammers who route human activity challenges meant for account creation to sketchy web sites to have them solved by those sites' visitors in exchange for free content [25], or to sweatshops with dedicated CAPTCHA solvers. Thus, *a* human was involved in providing the activity, but not necessarily the human intended by the issuer of the challenge.

In contrast, NAB generates responder-specific, content-specific, and, where appropriate, challenger-specific attestations. Attestations are certificates of human activity that contain a signature over the entire request content. For example, an email attestation contains the signature over the entire email, including the "From:" address (i.e., the responder), the email body (i.e., the content), and the "To:" address (i.e., the challenger). Similarly, a web request attestation contains the URL, which provides both responder-specific and content-specific attestations.

Content-specific attestation is more subtle. Whereas CAPTCHAs are used today for coarse-grained actions such as email account creation, they are considered too intrusive to be used for finer granularity requests such as sending email or retrieving web URLs. So, in practice, the challenge response is "amortized" over multiple requests (i.e., all email sent from the CAPTCHA-created mail account). Even if an actual human created the account, nothing prevents the bots in that human's desktop from sending email indiscriminately using that account.

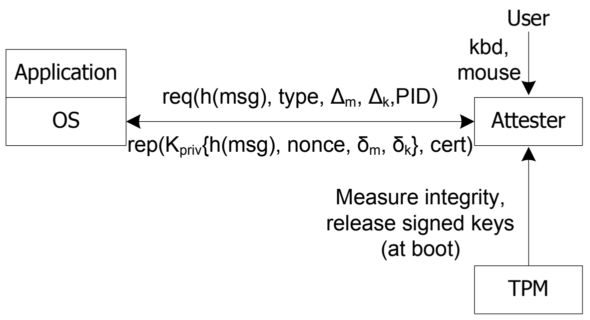

User
kbd, mouse
Application
OS
req(h(msg), type, Δ_m, Δ_k,PID)
Attester
rep(K_{priv}{h(msg), nonce, δ_m, δ_k}, cert)
Measure integrity, release signed keys (at boot)
TPM

Figure 2: Attester interfaces.

Finally, challenger-specific attestation helps in ensuring that unwitting, honest humans do not furnish attestations for bad purposes. A verifier expecting an attestation from human A's attester will reject an attestation from human B that might be provided instead. In the spam example, this is tantamount to explicit sender authentication.

Attestations with these three properties, together with application-specific verifier policies described in §5.2, meet our second and third requirements (§3.1).

4.3 Attester API

Figure 2 shows the relationship between the attester and other entities. The API is simple: there is only a single request/reply pair of calls between the OS and the attester. An application's attestation request contains the hash of the message to be attested (i.e., the contents of an email message or the URL of a browser click), the type of attestation requested, and the process id (PID) of the requesting process.

If the attester verifies that the type of attestation being requested is consistent with user activity seen on the keyboard/mouse channels, it signs the attestation and, depending on the attestation type, includes δ_m and δ_k, which indicate how long ago a mouse click and a keyboard click respectively were last seen. The attestation is an offline computation , and is thus an instance of a non-interactive proof of human activity.

The same API is used for all applications. The only customization allowed is whether to include the values of the δ_m or δ_k, depending on the attestation type. The attester uses a group signature scheme for anonymous attestations, extending the Direct Anonymous Attestation (DAA) service [7] provided by recent TPMs. Anonymous attestations preserve the current privacy semantics of web and email, thereby meeting our fourth and final requirement (§3.1).

We have currently defined and implemented two attestation types. Type 0 is for interactive applications such as all types of web requests. Type 1 is for delay-tolerant applications such as email. Type 0 attestations are generated only when there is either a mouse or keyboard click in the last one second, and do not include the δ_m or δ_k values. Type 0 attestations are offered as a privacy enhancement, to prevent verifiers from tracking at a fine temporal granularity a human user's activity or a particular request's specific source machine. We chose one second as the lag for Type 0 attestations since it is sufficient for local interactive applications; for example, this is ample time between a key or mouse click and a local action such as generating email or transmitting an HTTP GET request. Type 1 attestations can be used with all applications we have examined, when this finer privacy concern is unwarranted. To put the two types in perspective, a Type 0 attestation is roughly equivalent to a Type 1 attestation requested with $\Delta_m = \Delta_k = 1sec$ and in which the attested δ_m/δ_k values have been hidden.

Attestation structure. An attestation has the form $\langle d, n, \delta_m, \delta_k, \sigma, C\rangle$. It contains a cryptographic content digest d (e.g., a SHA-1 hash) of the application-specific payload being attested to; a nonce n used to maintain the freshness of the attestations and to disallow improper attestation reuse; the $\delta_{\{k,m\}}$ values (for type 1 attestations); the attestation signature $\sigma = sign(K_{priv}, \langle d, n, \delta_m, \delta_k\rangle)$; and a certificate C from the TPM guaranteeing the attester's integrity, the version of the attester being used, the attestation identity key of the TPM that measured the attester integrity, and the signed attester's public key K_{pub} (Figure 2). The certificate C is generated during booting of the attester and is stored and reused until reboot.

The mechanism for attesting to web requests is simple: when a user clicks on a URL that is either a normal link or an ad, the browser requests an attestation on the entire page URL. After the browser fetches the page content, it uses the same attestation to retrieve any included objects within the page. As explained in §5.2, the verifier accepts the attestation for all included objects.

The mechanism for sending email in the common case is also straightforward: the entire email message, including headers and attachments, constitutes the request. Interestingly, the same basic mechanism is extensible to other email usage scenarios, such as text or web-based email, email-over-ssh, batched and offline email, and script-generated email.

Email usage scenarios (mailing lists; remote, batched, offline, scripted or web mail). To send email to mailing lists, the attester attests to the email normally, except that the email destination address is the name of the target mailing list. Every recipient's verifier then checks that the recipient is subscribed to the mailing list, as described in §5.2. Also, a text-based email application running remotely over ssh can obtain attestations from the local machine with the help of the ssh client program executing locally. This procedure is similar to authentication credential forwarding implemented in ssh. Similarly, a graphical email client can obtain and store an attestation as soon as the "send" button is clicked, regardless of whether it has a working network connection, or if the email client is in an offline mode, or if the client uses an outbox to batch email instead of sending it immediately. In case of web mail, a browser can obtain an attestation on behalf of the web application.

Script-generated email is more complex. The PID argument in the attestation request (Figure 2) is used for deferred attestations, which are attestations approved ahead of time by the user. Such forms of attestation are not required normally, and are useful primarily for applications such as email-generating scripts, cron-jobs, etc. When an application requests a deferred attestation, the user approves the attestation explicitly through a reserved click sequence (currently "Ctl-Alt-F4", followed by number of deferred attestations). These attestations are stored in a simple PID-table in the attester, and released to the application in the future. Since the content of a deferred attestation is not typically known until later (such as when the body of an email is dynamically generated), it is dangerous to release an unbound attestation to the untrusted OS. Instead, the attester stores the deferred attestations in its own memory, and releases only bound attestations. Although the attester ensures that unbound attestations are not released to the untrusted OS, thereby limiting damage, there is no way to ensure that these attestations are not stolen by a bot faking the legitimate script's PID. However, the user is able to reliably learn about the missing attestations after this occurrence, which is helpful during troubleshooting.

4.4 Attester Implementation

The attester is a small module, currently at fewer than 500 source code lines. It requires a TPM chip conforming to any revision of the TPM v1.2 specification [28].

Attester installation. The attester is installed by binding its hash value to an internal TPM register called a Platform Configuration Register (PCR). We use PCR_{18}. Initially, the register value is -1. We extend it[1] with the attester through the TPM operation:

$$PCRExtend(18, H(ATT))$$

where $H(ATT)$ is the attester's hash. If the attester needs to be updated for some reason (which should be a rare event), PCR_{18} is reinitialized and extended with the new code value.

Key generation. At install time, the attester generates an anonymous signing key pair: $\{K_{pub}, K_{priv}\}$. This key pair is derived from the attestation identity key AIK of the TPM, and is an offline operation. K_{priv} allows the attester to sign requests anonymously. The attester then seals the private key K_{priv} to the TPM using the TPM's private storage root key K_{root}.

Assume that the system BIOS, which boots before the attester, extends PCR_{17}. Thus, the sealing operation renders K_{priv} inaccessible to everyone but the attester by executing the TPM call:

$$Seal((17, 18), K_{priv})$$

which returns the encrypted value C of K_{priv}. The TPM unseals and releases the key only to the attester, after the attester is booted correctly.

Until the TPM releases K_{priv} and the accompanying certificate to the attester, there is thus no way for the host to prove to an external verifier that a request is accompanied by human activity. Conversely, if the attester has a valid private key, the external verifier is assured that the attester is not tampered with.

Attester booting. The attester uses a static chain of trust rooted at the TPM and established at boot-time. It is booted as part of the secure boot loading operation before the untrusted OS itself is booted. After the BIOS is booted, it measures and launches the attester. After the attester is launched, it unseals the previously sealed K_{priv} by executing:

$$Unseal(C, MAC_{K_{root}}((17, PCR_{17}), (18, PCR_{18})))$$

The $Unseal$ operation releases K_{priv} only if the PCR registers 17 and 18 after reboot contain the same hash values as the registers at the time of sealing K_{priv}. If the PCR values match, the TPM decrypts C and returns K_{priv} to the attester.

Thus, by sealing the anonymous signing key K_{priv} to the TPM and using secure boot loading to release the key to the attester, NAB meets the challenge of generating attestations without globally unique identities.

Attester execution. The attester waits passively for attestation requests from an application routed through the untrusted OS. A small untrusted stub is loaded into the OS in order to interact with the attester on behalf of the application.

With our current attester design and implementation, applications need to be modified in order to obtain attestations. We find the modifications to be fairly small and localized (§6). The only change as far as applications are concerned is to first obtain appropriate attestations and then include them as part of the requests they submit today. Protocols such as SMTP (mail) or HTTP (web) need not be modified in order to include this functionality. SMTP allows extensible message headers, while HTTP can include the attestation as part of the "user agent" browser string or as an extended header.

5 Verifier Design and Implementation

We now describe how verifiers use attestations to implement attack-specific countermeasures for spam, DDoS and click-fraud.

5.1 Verifier Design

The verifier is co-located with the server processing requests. We describe how the server invokes the verifier for each application in §5.2. When invoked, the verifier is passed both the attestation and the request. The attestation and request contain all the necessary information to validate the request.

The verifier first checks the validity of the attester public key used for signing the request, by traversing the public-key chain in the certificate C (Figure 2). If valid, it then recomputes the hash of the request's content and verifies whether the signed hash value in the attestation matches the request's contents. Further, for attestations that include the $\delta_{\{k,m\}}$ values, the verifier also checks whether $\delta_{\{k,m\}}$ are less than the application-specified $\Delta_{\{k,m\}}$. The verifier then checks to ensure that the attestation is not being double-spent, as described in § 5.3.

A bot running in an untrusted domain cannot masquerade as a trusted attester to the verifier because a TPM will not release the signed K_{pub} (Figure 2) to the bot without the correct code hash. Further, it derives no benefit from tampering with the δ values it specifies in its requests, because the verifier enforces the application-specified upper-limit on $\delta_{\{k,m\}}$.

The verifier then implements an application-specific policy as described next.

5.2 Application-specific Policies

Verifiers implement application-specific policies to deal with bot traffic. Spam can be more aggressively filtered using information in the attestations, legitimate email with attestations can be correctly classified, DDoS can be handled more effectively by prioritizing requests with attestations over traffic without attestations, and click-fraud can be reduced by only serving requests with valid attestations and ignoring other requests.

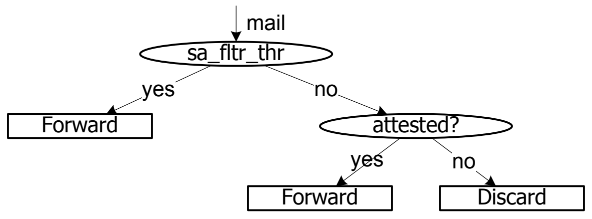


Figure 3: Sender ISP's verifier algorithm.

We now describe how the verifier implements such application-specific policies. Note that these are only example policies that we constructed for our three case studies, and many others are possible.

5.2.1 Spam policy

The biggest problem with Bayesian spam filters such as spamassassin today is that they either flag too much legitimate email as spam, or flag too little spam as such.

When all legitimate requests are expected to carry attestations, the verifier can set spam filters aggressively to flag questionable unattested messages as spam, but use positive evidence of human activity to "whitelist" questionable attested messages.

Sender ISP's email server. The verifier sits on the sender ISP's server alongside a Bayesian spam filter like spamassassin. The filter is configured at an aggressive, low threshold (e.g., -2 instead of the default 5 for spamassassin), because the ISP can force its users to send email with attestations, in exchange for relaying email through its own servers.

This low spamassassin "required score" threshold (or `sa_fltr_thr` in Figure 3) tags most unattested spam as unwanted. However, in the process, it might also tag some valid email as spam. In order to correct this mistake, the verifier "salvages" messages with a high spam filter score that carry a valid attestation, and relays them; high-score, unattested email is discarded as spam. This step ensures that legitimate human-generated email is forwarded unconditionally, even if the sender's machine is compromised. Thus, NAB guarantees that human-generated email from even a compromised machine is forwarded correctly (for example, in our trace study in §6, we did not find a single legitimate email that was ultimately rejected). Finally, while spam that steals attestations will also be forwarded, in our trace-based study this spam volume is 92% less than the spam forwarded today (§6). This reduction is because the attester limits the bot to acquiring attestations only when there is human activity, and even then at a rate limit of at most one attestation per Δ (one second for type 0 attestations).

Recipient's inbox. A second form of deploying the verifier is at the email recipient. This form can coexist with the verifier on the sender's side.

We observe that any email server classifying email as spam or not can ensure that a legitimate email is not misclassified by improving the spam score for email messages with attestations by a small number (=3, §6). This number should be high enough that all legitimate email is classified correctly, while spam with or without attestations is still caught.

The verifier improves the score for all attested emails by 3, thereby vastly improving the delivery of legitimate email. Additionally, in this deployment, the verifier also checks that the 'To:" or "Cc:" headers contain the recipient's email address or the address of a subscribed mailing list. If not (for example, in the case of "Bcc:"), it does not improve the spam score by 3 points.

Incentives. Email senders have an incentive to deploy NAB because it prevents their email from being misclassified as spam. Verifiers can be deployed either for reducing spam forwarded through mail relays or for ensuring that all legitimate email is classified and delivered correctly. Home ISPs, which see significant amount of compromised hosts on their networks, can benefit from the first deployment scenario, because, unlike other methods of content or IP-based filtering, attestations still allow all legitimate email from compromised hosts, while reducing spam significantly (§6). Also, web-based mail servers such as gmail have an incentive to deploy NAB so that they can avoid being blacklisted by other email relays by reducing the spam they forward today. Finally, email recipients have an incentive to deploy NAB because they will receive all legitimate email correctly, unlike today (§6).

5.2.2 DDoS policy

We consider scenarios where DDoS is effected by overloading servers, and not by flooding networks. The verifier resides in a firewall or load balancer, and observes the response time of the web server to determine whether the server is overloaded [31]. Here, unlike in spam, the verifier does not drop requests with invalid or missing attestations. Instead, it prioritizes requests with valid attestations over those that lack them. Prioritizing, rather than dropping, makes sense because some valid requests may actually be generated automatically by machines (for example, automatic page refreshes on news sites like `cnn.com`).

The verifier processes the web request in the following application-specific manner. If the request is for a page URL, the verifier treats it as a fresh request. It keeps a set of all valid attestations it has seen in the past 10 minutes, and adds the attestation and the requested page URL to the list. If the request is for an embedded object within a page URL, the verifier searches the attestation list to see if the attestation is present in the list. If the attestation is present in the list, and if the requested object belongs to the page URL recorded in the list for the attestation, the verifier treats the attestation as valid. Otherwise, it lowers the priority of the request. The verifier ages the stored attestation list every minute.

The priority policy serves all outstanding attested requests first, and uses any remaining capacity to serve all unattested requests in order.

Incentives. Overloaded web sites have a natural incentive to deploy verifiers. While users have an incentive to deploy attesters to receive priority treatment, the attester deployment barrier can be still high. However, since our attester is not application-specific, it is possible for the web browser to leverage the attester deployed for email or click-fraud.

5.2.3 Click-fraud Policy

Click-fraud occurs whenever an automated request is generated for a click, without any interest in the click target. For example, a botmaster puts up a web site to show ads from companies such as Google, and causes his bots to fetch ads served by Google through his web site. This action causes Google to pay money to the botmaster. Similarly, an ad target's competitor might generate invalid clicks in order to run up ad costs and bankrupt the ad purchaser. Further, the competitor might be able to purchase ad words for a smaller price, because the victim might no longer bid for the same ad word. Finally, companies like Google have a natural incentive to prove to their advertisers that ads displayed together with search results are clicked not by bots but by humans.

With NAB, a verifier such as Google can implement the verifier within its web servers, configured as a simple policy of not serving unattested requests. Also, it can log all attested requests to prove to the advertiser that the clicks Google is charging for are, in fact, human-generated.

Incentives. Companies like Google, Yahoo and Microsoft that profit from ad revenue have a good incentive to deploy verifiers internally. They also have an incentive to distribute the attester as part of browser toolbars. Such toolbars are either factory installed with new PCs, or the user can explicitly grant permission to install the attester. While the user may not benefit directly in this case, she benefits from spam and DDoS reduction, and from being made aware of potential problems when a bot steals key clicks.

5.3 Security guarantees

NAB provides two important security guarantees. First, it ensures that attestations cannot be double-spent. Second, it ensures that a bot cannot steal key clicks and accumulate attestations beyond a fixed time window, which reduces the aggregate volume and burstiness of bot traffic.

The verifier uses the nonce in the attestation (Figure 2) for these two guarantees. The verifier stores the nonces for a short period (10 minutes for web requests, one month for email). We find this nonce overhead to be small in practice (§6.3). If a bot recycles an attestation after one month, and the spam filter at the verifier flags the email as spam based on content analysis, the verifier uses the "Date:" field in the attested email to safely discard the request because the message is old.

The combination of application-specific verifier policy and content-bound attestations can also be used to mitigate bursty attacks. For example, a web URL can include an identifier that encodes the link freshness. Since attestations include the identifier, the verifier can discard out-of-date requests, even if they have valid signatures.

6 Evaluation

In this section, we evaluate NAB's two main components: a) our current attester prototype with respect to metrics such as TCB size, CPU requirements, and application changes; and b) our verifier prototype with respect to metrics such as the extent to which it mitigates attack-specific traffic such as spam, DDoS and click-fraud, and the rate at which it can verify attestations.

Our main experiments and their conclusions are shown in Table 1. We elaborate on each of them in turn.

6.1 Attester Evaluation

TCB size. We implemented the attester as a kernel module within Xen. Xen is well-suited because it provides a virtual machine environment with sufficient isolation between the attester and the untrusted OS. However, the chief difficulty was keeping the total TCB size small. Striving for a small TCB allows the attester to handle untrusted OSes with a higher assurance. While the Xen VM itself is small (about 30 times smaller than the Linux kernel), we have to factor the size of a privileged domain such as Domain-0 into the TCB code base. Unfortunately, this increases the size of the TCB to more than 5 million source lines of code (SLOC), the majority of which is device driver code.

Instead, we started with a minimal kernel that only includes the necessary drivers for our platform. We included the Xen VMM and built untrusted guest OSes us-

Experiment	Conclusion
TCB size	500 source lines of code (SLOC) for attester, 30K SLOC total
Attester CPU cost	$< 10^7$ instructions/attestation
Application changes	<250 SLOC for simple applications
Worst-case spam mitigation	$> 92\%$ spam suppressed; no human-sent email missed
Worst-case DDoS mitigation	$> 89\%$ non-human requests identified; no human requests demoted
Worst-case click-fraud mitigation	$> 87\%$ automated clicks denied; no human request denied
Verifier throughput	$> 1,000$ req/s. Scalable to withstand 100,000-bot DDoS

Table 1: Summary of key experiments and their results.

ing the mini-OS [19] domain building facility included in the Xen distribution. Mini-OS allows the user-space applications and libraries of the host VM to be untrusted, leaving us with a total codebase of around 30,000 source lines of code (SLOC) for the trusted kernel, VMM and attester. Our attester was less than 500 SLOC. While this approach produced a TCB that can be considered reasonably small, especially compared to the status quo, we are examining alternatives such as using Xen's driver domain facility that allows device drivers to run in unprivileged domains. We are also working on using the IOMMUs found on the newer Intel platforms, which enable drivers for devices other than keyboard and mouse to run in the untrusted OS, while ensuring that the attester cannot be corrupted due to malicious DMA requests. Such an approach makes the attester portable to any x86 platform.

Attester CPU cost. The attester uses RSA signatures with a 1024-bit modulus, enabling it to generate and return an attestation to the application with a worst-case latency of 10 ms on a 2 GHz Core 2 processor. This latency is usually negligible for email, ad click, or fetching web pages from a server under DDoS. Establishing an outgoing TCP connection to a remote server usually takes more than this time, and attestation generation is interleaved with connection establishment.

Application changes. We modified two command-line email and web programs to request and submit attestations: NET::SMTP, a Perl-based SMTP client, and cURL, an HTTP client written in C. Both modifications required changes or additions of less than 250 SLOC.

6.2 Verifier Evaluation

We used a trace study of detailed keyboard and mouse activity of 328 volunteering users at Intel to confirm the mitigation efficacy of our application-specific verifier policies. We find the following four main benefits with our approach:

1. If the sender's mail relay or the receiver's inbox uses NAB and checks for attestations, the amount of spam that passes through tuned spam filters (i.e., false negatives) reduces by more than 92%, while not flagging any legitimate email as spam (i.e., no false positives). The spam reduction occurs by setting the "scoring thresholds" aggressively; the presence of concomitant human activity greatly reduces the number of legitimate emails flagged as spam.
2. In addition to reduced spam users see in their inboxes, NAB also reduces the peak processing load seen at mail servers, because the amount of attested spam that can be sent even by an adaptive botnet is bounded by the number of human clicks that generate attestations. Hence, mail servers can prioritize attested requests potentially dropping low-priority ones, which improves the fraction of human-generated email processed during high-load periods.
3. NAB can filter out more than 89% of bot-mounted DDoS activity without misclassifying human-generated requests.
4. NAB can identify click-fraud activity generated by adware with more than 87% accuracy, without losing any human-generated web clicks.

Methodology. We use the keyboard and mouse click traces collected by Giroire et al. [11]; activity was recorded on participants' laptops at one-second granularity at all times, both at work and at home. Each user's trace is a sequence of records with the following relevant information: timestamp; number of keyboard clicks within the last second; number of mouse clicks within the last second; the foreground application that is receiving these clicks (such as "Firefox", "Outlook", etc.); and the user's network activity (i.e., the TCP flow records that were initiated in the last one second). Nearly 400 users participated in the trace study, but we use data from 328 users because some users left the study early. These 328 users provide traces continuously over a one-month period between Jan–Feb 2007, as long as their machines were powered on. While the user population size is moderate, the users and the workloads were diverse. For example, there were instances of significant input device activity corresponding to gaming activity outside regular work. So, we believe the traces are sufficiently representative of real-world activity.

Separately, we also collected malware traces from a honeypot. The malware whose traces we gathered included: a) the Storm Worm [13], which was until recently the largest botnet, generating several tens of billions of spam messages per day; and b) three adware bots called 180solutions, Nbcsearch and eXactSearch, which are known to perpetrate click-fraud against Yahoo/Overture. For spam, we also used a large spam corpus containing more than 100,000 spam messages and 50,000 valid messages [1]. Each message in the corpus is hand-classified as spam or non-spam, providing us with ground-truth. For DDoS, we use traffic traces from the Internet Traffic Archive [27], which contain flash-crowd scenarios. We assume that these flash crowds represent DDoS requests, because, as far as an overloaded server is concerned, the two scenarios are indistinguishable.

We overlay the user activity traces with the malware and DDoS traces for each user, and compare the results experienced by the user at the output of the verifier with and without attestations. We consider two strategies for overlaying requests: a normal bot and an adaptive bot. The adaptive bot represents the worst-case scenario for the verifier, because it monitors human activity and modulates its transmissions to collect attestations and masquerade as a user at the verifier.

We consider an adaptive adversary that buffers its requests until it sees valid human activity, and simulate the amount of benefit NAB can provide under such adversarial workloads.

Spam mitigation. The verifier can be used in two ways (§5.2). First, mail relays such as gmail or the SMTP server at the user's ISP can require attestations for outgoing email. In this case, the main benefit comes from filtering out all unattested spam and catching most attested spam, while allowing all legitimate email. So, the main metric here is how much attested spam is suppressed. Second, the inbox at the receiver can boost the "spam score" for all attested email, thereby improving the probability that a legitimate email is not misclassified. So, the main metric here is how much attested human-generated email is misclassified as spam.

Figure 4 shows the amount of spam, attested or not, that managed to sneak through spamassassin's Bayesian filter for a given spam threshold setting. By setting a spam threshold of -2 for an incoming message , and admitting messages that still cleared this threshold and carried valid attestations, we cut down the amount of spam forwarded by mail relays by more than 92% compared to the amount of spam forwarded currently.

From our traces, we also found that no attested human-generated email is misclassified as spam for a spam threshold setting of 5, as long as the spam score of attested messages is boosted by 3 points. On the other hand, spamassassin uses a threshold of 5 by default because, without attestations, a lot of valid email would be missed if it were to use a spam score of -2. Even so, about 0.08% of human-generated email is still misclassified as spam, which is a significant improvement of legitimate email reception.

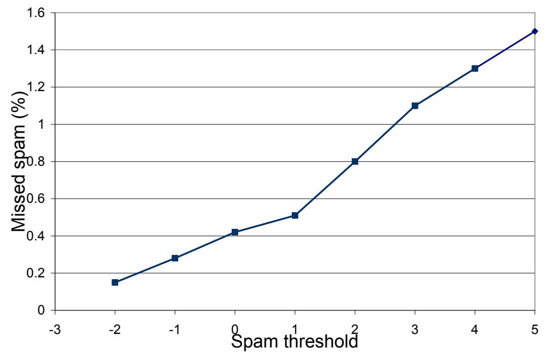


Figure 4: Missed spam percentage vs. spam threshold with attestations. By setting spam threshold to -2, spam cleared by spamassassin and received in inboxes today is reduced more than 92% even in worst case (i.e., adaptive bots), without missing any legitimate email.

There is another benefit that the verifier can derive by using attestations. It comes in the form of reduced peak load observed while processing spam. Today's email servers are taxed by ever-increasing spam requests [23]. At peak times, the mail server can prioritize messages carrying attestations over those that do not, and process the lower-priority messages later.

Figure 5 shows the CDF of the percentage of spam requests that the verifier must still service at a high priority because of stolen attestations. NAB demotes spam traffic without attestations by more than 91% in the worst case (equivalently, less than 7.5% of spam traffic is served at the high priority). At the same time, no human-generated requests are demoted. The mean of the admitted spam traffic is 2.7%, and the standard deviation is 1.3%. Thus, NAB reduces peak server load by more than 10×.

DDoS mitigation. The verifier uses the DDoS policy described in §5.2, by giving lower priority to requests without attestations. Figure 6 shows the CDF of the percentage of DDoS requests that the verifier still serves at a high priority because of stolen attestations. NAB demotes DDoS traffic by more than 89% in the worst case (equivalently, only 11% of DDoS traffic is served at the high priority). At the same time, no human-generated requests are demoted. The mean of the admitted DDoS traffic is 5.8%, and the standard deviation is 2.2%.

Click-fraud mitigation The verifier uses the Click-fraud policy described in §5.2. Figure 7 shows the amount of click-fraud requests that the verifier satisfies due to

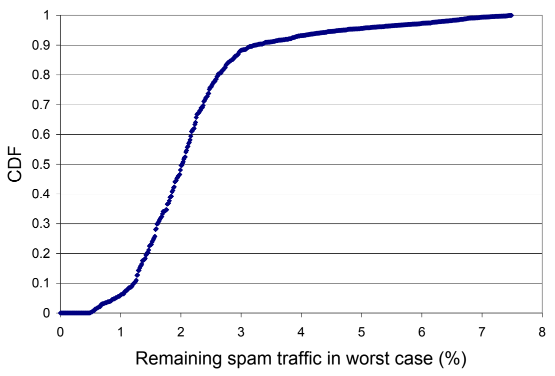


Figure 5: CDF of percentage of bots' spam requests serviced by an email server in the worst case. The mail server's peak spam processing load is reduced to less than 7.5% of its current levels.

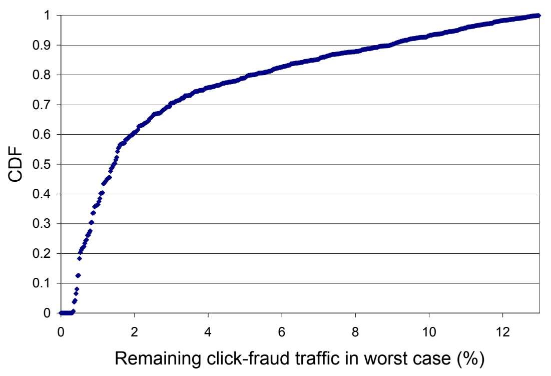


Figure 7: CDF of percentage of bots' click-fraud requests serviced in the worst case. Serviced click-fraud requests are restricted to less than 13% of original levels.

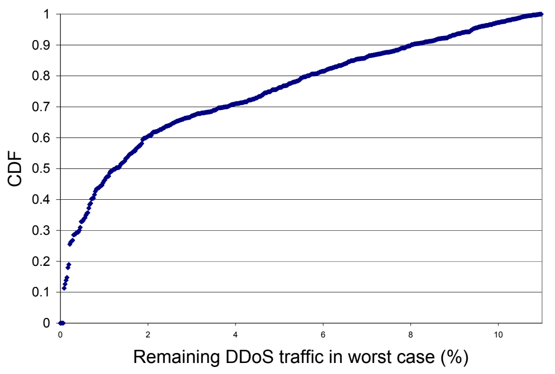


Figure 6: CDF of percentage of bots' DDoS requests serviced in the worst case. Allowed DDoS traffic is restricted to less than 11% of original levels.

valid attestations. NAB denies more than 87% of all in the worst case (equivalently, only 13% of all click-fraud requests is serviced). At the same time, no human-generated requests are denied service. The mean of the serviced click-fraud traffic is 7.1%, and the standard deviation is 3.1%.

6.3 Verifier Throughput

The verifier processes attestations, which are signed RSA messages, at a rate of more than 10,000 attestations per second on a 2 GHz Core 2 processor. It benefits from the fact that RSA verification is several times faster than signing. The verifier processes an attestation by consulting the data base of previously seen nonces within an application-specific period. The longest is email, with a duration of one month, while nonces of web requests are stored for 10 minutes, and fit in main memory. Even in the worst-case scenario of a verifier at an ISP's busy SMTP relay, the storage and lookup costs for the nonces are modest—for a server serving a million clients, each of which sends a thousand emails per day, the nonce storage overhead is around 600 GB, which can fit on a single disk and incur one lookup overhead. This overhead is modest compared to the processing and storage costs incurred for reliable email delivery.

Another concern is that the verifier is itself susceptible to a DDoS attack. To understand how well our verifier can withstand DDoS attacks, we ran experiments on a cluster of 10 Emulab machines configured as distributed email verifiers. We launched a DDoS from bots with fake attestations. Each DDoS bot sent 1 req/s to one of the ten verifiers at random, in order to mimic the behavior of distributed low-rate bots forming a DDoS botnet. Our goal was to determine whether a botnet of 100,000 nodes (which is comparable to the median botnet size) can overwhelm this verifier infrastructure or not. Our bot implementation used 100 clients to simulate 1000 bots each, and attack the ten verifier machines. We assume network bandwidth is not a bottleneck, and that the bots are targeting the potential verification overhead bottleneck. A verifier queues incoming requests until it can attend to it, and has sufficient request buffers.

Figure 8 shows the latency increase (in ms) experienced by a normal client request. Normally, a user takes about 1 ms to get her attestation verified. With DDoS, we find that even a 100,000-node botnet degrades the performance of a normal request only by an additional 1.2 ms at most. Hence, normal request processing is not affected significantly. Thus, a cluster of 10 verifiers can withstand a 100,000-node botnet using fake attestations.

7 Related Work

We classify prior work into three main categories.

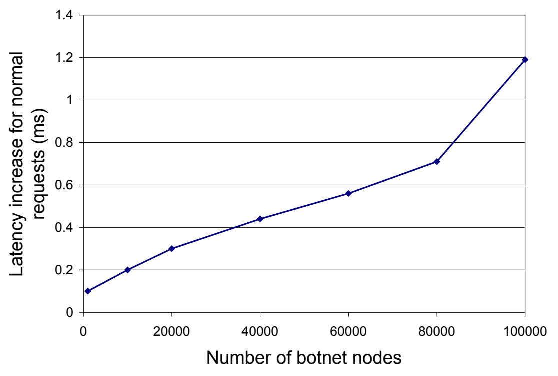


Figure 8: Request processing latency at the verifier.

Human activity detection. CAPTCHAs [30**?**] are the currently popular mechanism for proving human presence to remote verifiers. However, as described in §4, they suffer from four major drawbacks that render them less attractive for mitigating botnet attacks. First, CAPTCHAs as they are used today are transferable and not bound to the content they attest, and are hence vulnerable to man-in-the-middle attacks, although one could imagine designs to improve this shortcoming; second, they are semantically independent of the application (i.e., unbound to the user's intent), are hence exposed to human solver attacks; third, they are obtrusive, which restricts their use for fine-grained attestations (by definition, CAPTCHAs require manual human input), and hence cannot be automated, unlike NAB. Also, we are witnessing continued successes in breaking the CAPTCHA implementations of several sites such as Google, Yahoo, and MSN [12], leading some to question even their long-term viability [34], at least in their current form. By contrast, NAB's security relies on cryptographic protocols such as RSA that have been studied and used longer.

The recent work on the Nexus operating system [33] has developed support for application properties to be securely expressed using a trusted reference monitor mechanism. The Nexus reference monitor is more expressive than a TPM implementing a hash-based trusted boot. So, it allows policies restricting outgoing email only from registered email applications. In contrast, we assume commodity untrusted OS and applications.

The approach of using hardware to enable human activity detection has been described before in the context of on-line games, using untrusted hardware manageability engines (such as Intel's AMT features) [21].

Mitigating spam, DDoS and click-fraud. There is extensive literature related to mitigation techniques for Spam [2], DDoS [20, 35] and click-fraud [26]. There are still no satisfactory solutions, so application-specific defenses are continuously proposed. For example, Occam [10], SPF (Sender Policy Framework), DKIM (DomainKeys Identified Mail) and "bonded sender" [6] have been put forth recently as enhancements. Similarly, DDoS and click-fraud mitigation have each seen several radically different attack-specific proposals recently. These proposals include using bandwidth-as-payment [31], path validation [35], and computational proofs of work [20] for DDoS; and using syndicators, premium clicks, and clickable CAPTCHAs for click-fraud [26].

While all these proposals certainly have several merits, we propose that it is possible to mitigate a variety of botnet attacks using a uniform mechanism such as NAB's attestation-based human activity verification. Such a uniform attack mitigation mechanism amortizes its cost of deployment. Moreover, unlike some proposals, NAB does not rely on IP-address blacklisting, which is unlikely to work well because even legitimate requests from a blacklisted host are denied. Also, NAB can be implemented purely at the end hosts, and does not require Internet infrastructure modification.

Secure execution environments. The TPM specifications [28] defined by the Trusted Computing Group are aimed at providing primitives that can be used to provide security guarantees to commodity OSes. TPM-like services have been extended to OSes that cannot have exclusive access to a physical TPM device of their own, as with legacy and virtual machines. For example, Pioneer [22] provides an externally verifiable code execution environment for legacy devices similar to that provided by a hardware TPM, and vTPM [5] provides full TPM services to multiple virtualized OSes. NAB assumes a single OS and a hardware TPM, but can leverage this research in future.

XOM [15] and Flicker [16] provide trusted execution support even when physical devices such as DMA or, with XOM, even main memory are corrupted, while SpyProxy [18] blocks suspicious web content by executing the content in a virtual machine first. In contrast, NAB assumes compromised machines' hardware is functioning correctly, that the bot may generate diverse traffic such as spam and DDoS, and that owners do not mount hardware attacks against their own machines, which is realistic for botted machines.

8 Conclusions

This paper presented NAB, a system for mitigating network attacks by using automatically obtained evidence of human activity. NAB uses a simple mechanism centered around TPM-backed attestations of keyboard and mouse clicks. Such attestations are responder- and content-specific, and certify human activity even in the absence of globally unique identities. Application-specific verifiers use these attestations to implement various policies. Our implementation shows that it is feasible to provide such attestations at low TCB size and runtime cost. By evaluating NAB using trace analysis, we estimate that NAB can reduce the amount of spam evading tuned spam filters by more than 92% even with worst-case adversarial bots, while ensuring that no legitimate email is misclassified as spam. We realize similar benefits for DDoS and click-fraud. Our results suggest that the application-independent abstraction provided by NAB enables a range of verifier policies for applications that would like to separate human-generated requests from bot traffic.

Acknowledgments. We thank Nina Taft and Jaideep Chandrashekar for the click traces used in this paper, our shepherd Geoff Voelker, Bryan Parno, Frans Kaashoek and the anonymous reviewers for their helpful comments.

References

[1] 2005 TREC public spam corpus, `http://plg.uwaterloo.ca/~gvcormac/treccorpus/`.

[2] A plan for spam, `http://www.paulgraham.com/spam.html`.

[3] P. Barham, B. Dragovic et al. Xen and the art of virtualization. In *SOSP'03*.

[4] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *CRYPTO'93*.

[5] S. Berger, R. Cáceres et al. vTPM: Virtualizing the Trusted Platform Module. In *USENIX-SS'06: Proceedings of the 15th conference on USENIX Security Symposium*.

[6] Bonded sender program, `http://www.bondedsender.com`.

[7] E. Brickell, J. Camenisch, and L. Chen. Direct anonymous attestation. In *CCS'04*.

[8] Click fraud rate rises to 14.1%, `http://redmondmag.com/columns/print.asp?EditorialsID=1456`.

[9] Five percent of Web traffic caused by DDoS attacks, `http://www.builderau.com.au/news/soa/Five-percent-of-Web-traffic-caused-by-DDoS-attacks/0,339028227,339287902,00.htm`.

[10] C. Fleizach, G. Voelker, and S. Savage. Slicing spam with occam's razor. In *CEAS'07*.

[11] F. Giroire, J. Chandrashekar et al. The Cubicle Vs. The Coffee Shop: Behavioral Modes in Enterprise End-Users. In *PAM'08*.

[12] Gmail CAPTCHA cracked, `http://securitylabs.websense.com/content/Blogs/2919.aspx`.

[13] T. Holz, M. Steiner, F. Dahl, E. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on Storm worm. In *Leet'08*.

[14] C. Kanich, C. Kreibich et al. Spamalytics: An empirical analysis of spam marketing conversion. In *CCS'08*.

[15] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP'03*.

[16] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *EuroSys'08*.

[17] D. Moore, C. Shannon, D. J. Brown, G. M. Voelker, and S. Savage. Inferring Internet denial-of-service activity. *ACM Trans. Comput. Syst.*, 24(2), 2006.

[18] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. SpyProxy: Execution-based detection of malicious web content. In *USENIX'07*.

[19] D. G. Murray, G. Milos, and S. Hand. Improving xen security through disaggregation. In *VEE'08*.

[20] B. Parno, D. Wendlandt et al. Portcullis: Protecting connection setup from denial-of-capability attacks. In *SIGCOMM'07*.

[21] T. Schluessler, S. Goglin, and E. Johnson. Is a bot at the controls?: Detecting input data attacks. In *SIGCOMM workshop on Network and system support for games*, 2007.

[22] A. Seshadri, M. Luk et al. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *SOSP'05*.

[23] Six botnets churning out 85% of all spam, `http://arstechnica.com/news.ars/post/20080305-six-botnets-churning-out-85-percent-of-all-spam.html`.

[24] Spam reaches all-time high of 95% of all email, `http://www.net-security.org/secworld.php?id=5545`.

[25] Spammers using porn to break CAPTCHAs, `http://www.schneier.com/blog/archives/2007/11/spammers_using.html`.

[26] The first AdFraud workshop, `http://crypto.stanford.edu/adfraud/`.

[27] Traces in the Internet Traffic Archive, `http://ita.ee.lbl.gov/html/traces.html`.

[28] Trusted Platform Module (TPM) specifications, `https://www.trustedcomputinggroup.org/specs/TPM/`.

[29] Vista's UAC security prompt was designed to annoy you, `http://arstechnica.com/news.ars/post/20080411-vistas-uac-security-prompt-was-designed-to-annoy-you.html`.

[30] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using Hard AI Problems for Security. In *Eurocrypt'03*.

[31] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS Defense by Offense. In *SIGCOMM'06*.

[32] A. Whitten and J. D. Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security'99*.

[33] D. Williams, P. Reynolds, K. Walsh, E. G. Sirer, and F. B. Schneider. Device driver safety through a reference validation mechanism. In *OSDI'08*.

[34] Windows Live Hotmail CAPTCHA cracked, exploited, `http://arstechnica.com/news.ars/post/20080415-gone-in-60-seconds-spambot-cracks-livehotmail-captcha.html`.

[35] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM'05*.

Notes

[1]The TPM terminology uses the term *register extension* to imply appending a new value to the hash chain maintained by that register.

BotGraph: Large Scale Spamming Botnet Detection

Yao Zhao† [*], Yinglian Xie, Fang Yu, Qifa Ke, Yuan Yu, Yan Chen†, and Eliot Gillum‡
†Northwestern University
Microsoft Research Silicon Valley
‡Microsoft Corporation

Abstract

Network security applications often require analyzing huge volumes of data to identify abnormal patterns or activities. The emergence of cloud-computing models opens up new opportunities to address this challenge by leveraging the power of parallel computing.

In this paper, we design and implement a novel system called *BotGraph* to detect a new type of botnet spamming attacks targeting major Web email providers. BotGraph uncovers the correlations among botnet activities by constructing large user-user graphs and looking for tightly connected subgraph components. This enables us to identify stealthy botnet users that are hard to detect when viewed in isolation. To deal with the huge data volume, we implement BotGraph as a distributed application on a computer cluster, and explore a number of performance optimization techniques. Applying it to two months of Hotmail log containing over 500 million users, BotGraph successfully identified over 26 million botnet-created user accounts with a low false positive rate. The running time of constructing and analyzing a 220GB Hotmail log is around 1.5 hours with 240 machines. We believe both our graph-based approach and our implementations are generally applicable to a wide class of security applications for analyzing large datasets.

1 Introduction

Despite a significant breadth of research into botnet detection and defense (e.g., [8, 9]), botnet attacks remain a serious problem in the Internet today and the phenomenon is evolving rapidly ([4, 5, 9, 20]): attackers constantly craft new types of attacks with an increased level of sophistication to hide each individual bot identities.

One recent such attack is the *Web-account abuse attack* [25]. Its large scale and severe impact have repeatedly caught public media's attention. In this attack, spammers use botnet hosts to sign up millions of user accounts (denoted as *bot-users* or *bot-accounts*) from major free Web email service providers such as AOL, Gmail, Hotmail, and Yahoo!Email. The numerous abused bot-accounts were used to send out billions of spam emails across the world.

Existing detection and defense mechanisms are ineffective against this new attack: The widely used mail server reputation-based approach is not applicable because bot-users send spam emails through only legitimate Web email providers. Furthermore, it is difficult to differentiate a bot-user from a legitimate user individually, as both users may share a common computer and that each bot-user sends only a few spam emails [1].

While detecting bot-users individually is difficult, detecting them as an aggregate holds the promise. The rational is that since bot-users are often configured similarly and controlled by a small number of botnet commanders, they tend to share common features and correlate each other in their behavior such as active time, spam contents, or email sending strategies [24, 27]. Although this approach is appealing, realizing it to enable detection at a large scale has two key challenges:

- The first is the algorithmic challenge in finding subtle correlations among bot-user activities and distinguishing them from normal user behavior.
- The second challenge is how to efficiently analyze a large volume of data to unveil the correlations among hundreds of millions of users. This requires processing hundreds of gigabytes or terabytes of user activity logs.

Recent advancement in distributed programming models, such as MapReduce [6], Hadoop [2], and Dryad/DryadLINQ [10, 29], has made programming and computation on a large distributed cluster much easier. This provides us with opportunities to leverage the parallel computing power to process data in a scalable fashion. However, there still exist many system design and implementation choices.

In this paper, we design and implement a system called *BotGraph* to detect the Web-account abuse attack at a large scale. We make two important contributions.

Our first contribution is to propose a novel graph-based approach to detect the new Web-account abuse attack. This approach exposes the underlying correlations among user-login activities by constructing a *large user-user graph*. Our approach is based on the observation that bot-users *share IP addresses* when they log in and send emails. BotGraph detects the abnormal sharing of IP addresses among bot-users by leveraging the random graph theory. Applying BotGraph to two months of Hotmail log of total 450GB data, BotGraph successfully identified over 26 million bot-accounts with a low false positive rate of 0.44%. To our knowledge, we are the first to provide a

[*]The work was done while Yao was an intern at Microsoft Research Silicon Valley.

[1]Recent anecdotal evidence suggests that bot-users have also been programmed to receive emails and read them to make them look more legitimate.

systematic solution that can successfully detect this new large-scale attack.

Our second contribution is an efficient implementation using the new distributed programming models for constructing and analyzing large graphs. In our application, the graph to construct involves tens of millions of nodes and hundreds of billions of edges. It is challenging to efficiently construct such large graphs on a computer cluster as the task requires computing pair-wise correlations between any two users. We present two graph construction methods using different execution plans: the simpler one is based on the MapReduce model [6], and the other performs selective filtering that requires the Join operation provided by Map-Reduce-Merge [28] or DryadLINQ [29]. By further exploring several performance optimization strategies, our implementation can process a one-month dataset (220GB-240GB) to construct a large graph with tens of millions of nodes in 1.5 hours using a 240-machine cluster. The ability to efficiently compute large graphs is critical to perform constant monitoring of user-user graphs for detecting attacks at their earliest stage.

Our ultimate goal, however, is not to just tackle this specific new form of attacks, but also to provide a general framework that can be adapted to other attack scenarios. To this end, the adoption of a graph representation can potentially enable us to model the correlations of a wide class of botnet attacks using various features. Furthermore, since graphs are powerful representations in many tasks such as social network analysis and Web graph mining, we hope our large-scale implementations can serve as an example to benefit a wide class of applications for efficiently constructing and analyzing large graphs.

The rest of the paper is organized as follows. We discuss related work in Section 2, and overview the BotGraph system in Section 3. We then describe in Section 4 the detail algorithms to construct and analyze a large user-user graph for attack detection. We present the system implementation and performance evaluation in Section 5, followed by attack detection results in Section 6. Finally, we discuss attacker countermeasures and system generalizations in Section 7.

2 Background and Related Work

In this section, we first describe the new attack we focus on in our study, and review related work in botnet detection and defense. As we use Dryad/DryadLINQ as our programming model for analyzing large datasets, we also discuss existing approaches for parallel computation on computer clusters, particularly those relate to the recent cloud computing systems.

2.1 Spamming Botnets and Their Detection

The recent Web-account abuse attack was first reported in summer 2007 [25], in which millions of botnet email accounts were created from major Web email service providers in a short duration for sending spam emails. While each user is required to solve a CAPTCHA test to create an account, attackers have found ways to bypass CAPTCHAs, for example, redirecting them to either spammer-controlled Web sites or dedicated cheap labor [2]. The solutions are sent back to the bot hosts for completing the automated account creation. Trojan.Spammer.HotLan is a typical worm for such automated account signup [25]. Today, this attack is one of the major types of large-scale botnet attacks, and many large Web email service providers, such as Hotmail, Yahoo!Mail, and Gmail, are the popular attack targets. To our best knowledge, BotGraph is one of the first solutions to combat this new attack.

The Web-account abuse attack is certainly not the first type of botnet spamming attacks. Botnet has been frequently used as a media for setting up spam email servers. For example, a backdoor rootkit Spam-Mailbot.c can be used to control the compromised bots to send spam emails. Storm botnet, one of the most widespread P2P botnets with millions of hosts, at its peak, was deemed responsible for generating 99% of all spam messages seen by a large service provider [9, 19].

Although our work primarily focuses on detecting the Web-account abuse attack, it can potentially be generalized to detect other botnet spamming attacks. In this general problem space, a number of previous studies have all provided us with insights and valuable understanding towards the different characteristics of botnet spamming activities [1, 11, 23, 26]. Among recent work on detecting botnet membership [20, 22, 24, 27], SpamTracker [24] and AutoRE [27] also aim at identifying correlated spamming activities and are more closely related with our work. In addition to exploiting common features of botnet attacks as SpamTracker and AutoRE do, BotGraph also leverages the connectivity structures of the user-user relationship graph and explores these structures for botnet account detection.

2.2 Distributed and Parallel Computing

There has been decades of research on distributed and parallel computing. Massive parallel processing (MPP) develops special computer systems for parallel computing [15]. Projects such as MPI (Message Passing Interface) [14] and PVM(Parallel Virtual Machine) [21] develop software libraries to support parallel computing. Distributed database is another large category of parallel data processing applications [17].

The emergence of cloud computing models, such as MapReduce [6], Hadoop [2], Dryad/DryadLINQ [10, 29], has enabled us to write simple programs for efficiently analyzing a vast amount of data on a computer cluster. All of them adopt the notion of staged computation, which makes scheduling, load balancing, and failure recovery automatic. This opens up a plethora of opportunities for re-thinking network security—an application

[2]Interestingly, solving CAPTCHAs has ended up being a low-wage industry [3].

that often requires processing huge volumes of logs or trace data. Our work is one of the early attempts in this direction.

While all of these recent parallel computing models offer scalability to distributed applications, they differ in programming interfaces and the built-in operation primitives. In particular, MapReduce and Hadoop provide two simple functions, Map and Reduce, to facilitate data partitioning and aggregation. This abstraction enables applications to run computation on multiple data partitions in parallel, but is difficult to support other common data operations such as database Join. To overcome this shortcoming, Map-Reduce-Merge [28] introduces a Merge phase to facilitate the joining of multiple heterogeneous datasets. More recent scripting languages, such as Pig Latin [16] and Sawzall [18], wrap the low level MapReduce procedures and provide high-level SQL-like query interfaces. Microsoft Dryad/DryadLINQ [10, 29] offers further flexibility. It allows a programmer to write a simple C# and LINQ program to realize a large class of computation that can be represented as a DAG.

Among these choices, we implemented BotGraph using Dryad/DryadLINQ, but we also consider our processing flow design using the more widely used MapReduce model and compare the pros and cons. In contrast to many other data-centric applications such as sorting and histogram computation, it is much more challenging to decompose graph construction for parallel computation in an efficient manner. In this space, BotGraph serves as an example system to achieve this goal using the new distributed computing paradigm.

3 BotGraph System Overview

Our goal is to capture spamming email accounts used by botnets. As shown in Figure 1, BotGraph has two components: aggressive sign-up detection and stealthy bot-user detection. Since service providers such as Hotmail limit the number of emails an account can send in one day, a spammer would try to sign up as many accounts as possible. So the first step of BotGraph is to detect aggressive *signups*. The purpose is to limit the total number of accounts owned by a spammer. As a second step, BotGraph detects the remaining stealthy bot-users based on their *login* activities. With the total number of accounts limited by the first step, spammers have to reuse their accounts, resulting in correlations among account logins. Therefore BotGraph utilizes a graph based approach to identify such correlations. Next, we discuss each component in detail.

3.1 Detection of Aggressive Signups

Our aggressive signup detection is based on the premise that signup events happen infrequently at a single IP address. Even for a proxy, the number of users signed up from it should be roughly consistent over time. A sudden increase of signup activities is suspicious, indicating that the IP address may be associated with a bot. We use

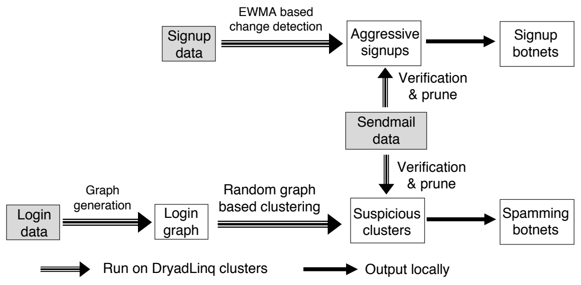


Figure 1: The Architecture of BotGraph.

a simple EWMA (Exponentially Weighted Moving Average) [13] algorithm to detect sudden changes in signup activities. This method can effectively detect over 20 million bot-users in 2 months (see Appendix A for more details on EWMA). We can then apply adaptive throttling to rate limit account-signup activities from the corresponding suspicious IP addresses.

One might think that spammers can gradually build up an aggressive signup history for an IP address to evade EWMA-based detection. In practice, building such a history requires a spammer to have full control of the IP address for a long duration, which is usually infeasible as end-users control the online/offline switch patterns of their (compromised) computers. The other way to evade EWMA-based detection is to be stealthy. In the next section we will introduce a graph based approach to detect stealthy bot-users.

3.2 Detection of Stealthy Bot-accounts

Our second component detects the remaining stealthy bot-accounts. As a spammer usually controls a set of bot-users, defined as a a *bot-user group*, these bot-users work in a collaborative way. They may share similar login or email sending patterns because bot-masters often manage all their bot-users using unified toolkits. We leverage the similarity of bot-user behavior to build a user-user graph. In this graph, each vertex is a user. The weight for an edge between two vertices is determined by the features we use to measure the similarity between the two vertices (users). By selecting the appropriate features for similarity measurement, a bot-user group will reveal itself as a connected component in the graph.

In BotGraph, we use the number of common IP addresses logged in by two users as our similarity feature (i.e., edge weight). This is because the aggressive account-signup detection limits the number of bot-accounts a spammer may obtain. In order to achieve a large spam-email throughout, each bot-account will log in and send emails multiple times at different locations, resulting in the sharing of IP addresses as explained below:

- **The sharing of one IP address:** For each spammer, the number of bot-users is typically much larger than the number of bots. Our data analysis shows that on each day, the average number of bot-users is about

50 times more than the number of bots. So multiple bot-users must log in from a common bot, resulting in the sharing of a common IP address.

- **The sharing of multiple IP addresses:** We found that botnets may have a high churn rate. A bot may be quarantined and leave the botnet, and new bots may be added. An active bot may go offline and it is hard to predict when it will come back online. To maximize the bot-account utilization, each account needs to be assigned to different bots over time. Thus a group of bot-accounts will also share multiple IP addresses with a high probability.

Our BotGraph system leverages the two aforementioned IP sharing patterns to detect bot-user activities.

Note that with dynamic IP addresses and proxies, normal users may share IP addresses too. To exclude such cases, multiple shared IP addresses in the same Autonomous System (AS) are only counted as one shared IP address. In the rest of this paper, we use the number of "shared IP addresses" to denote the the number of ASes of the shared IP addresses. It is very rare to have a group of normal users that always coincidentally use the same set of IP addresses across different domains. Using the AS-number metric, a legitimate user on a compromised bot will not be mistakenly classified as a bot-user because their number of "shared IPs" will be only one [3].

4 Graph-Based Bot-User Detection

In this section we introduce random graph models to analyze the user-user graph. We show that bot-user groups differentiate themselves from normal user groups by forming giant components in the graph. Based on the model, we design a hierarchical algorithm to extract such components formed by bot-users. Our overall algorithm consists of two stages: 1) constructing a large user-user graph, 2) analyzing the constructed graph to identify bot-user groups. Note one philosophy we use is to analyze group properties instead of single account properties. For example, it may be difficult to use email-sending statistics for individual bot-account detection (each bot account may send a few emails only), but it is very effective to use the group statistics to estimate how likely a group of accounts are bot-accounts (e.g., they all sent a similar number of emails).

4.1 Modeling the User-User Graph

The user-user graph formed by bot-users is drastically different from the graph formed by normal users: bot-users have a higher chance of sharing IP addresses and thus more tightly connected in the graph. Specifically, we observed the bot-user subgraph contains a *giant connected component*—a group of connected vertices that occupies a significant portion of the subgraph, while the normal-user subgraph contains only isolated vertices and/or very small connected components. We introduce the random graph theory to interpret this phenomenon and to model the giant connected components formed by bot-users. The theory also serves as a guideline for designing our graph-based bot-user detection algorithm.

4.1.1 Giant Component in User-User Graph

Let us first consider the following three typical strategies used by spammers for assigning bot-accounts to bots, and examine the corresponding user-user graphs.

- Bot-user accounts are randomly assigned to bots. Obviously, all the bot-user pairs have the same probability p to be connected by an edge.
- The spammer keeps a queue of bot-users (i.e., the spammer maintains all the bot-users in a predefined order). The bots come online in a random order. Upon request from a bot when it comes online, the spammer assigns to the requesting bot the top k available (currently not used) bot-users in the queue. To be stealthy, a bot makes only one request for k bot-users each day.
- The third case is similar to the second case, except that there is no limit on the number of bot-users a bot can request for one day and that $k = 1$. Specifically, a bot requests one bot-account each time, and it asks for another account after finishing sending enough spam emails using the current account.

We simulate the above typical spamming strategies and construct the corresponding user-user graph. In the simulation, we have 10,000 spamming accounts ($n = 10,000$) and 500 bots in the botnet. We assume all the bots are active for 10 days and the bots do not change IP addresses. In model 2, we pick $k = 20$. In model 3, we assume the bots go online with a Poisson arrival distribution and the length of bot online time fits a exponential distribution. We run each simulation setup 10 times and present the average results.

Figure 2 shows the simulation results. We can see that there is a sharp increase of the size of the largest connected component as the threshold T decreases (*i.e.*, the probability of two vertices being connected increases). In other words, there exists some transition point of T. If T is above this transition point, the graph contains only isolated vertices and/or small components. Once T crosses the transition point, the giant component "suddenly" appears. Note that different spamming strategies may lead to different transition values. Model 2 has a transition value of $T = 2$, while Model 1 and 3 have the same transition value of $T = 3$.

Using email server logs and a set of known botnet accounts provided by the Hotmail operational group, we have confirmed that generally bot-users are above the transition point of forming giant components, while normal users usually cannot form large components with more than 100 nodes.

[3] We assume majority of hosts are physically located in only one AS. We discuss how to prune legitimate mobile users in Section 4.2.2.

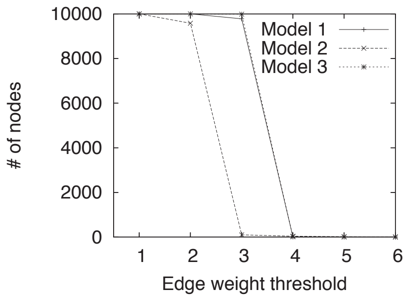


Figure 2: The size of the largest connected component.

4.1.2 Random Graph Theory

The sudden appearance of a giant subgraph component after a transition point can be interpreted by the theory of random graphs.

Denote $G(n, p)$ as the random graph model, which generates a n-vertex graph by simply assigning an edge to each pair of vertices with probability $p \in (0, 1]$. We call the generated graph an instance of the model $G(n, p)$. The parameter p determines when a giant connected component will appear in the graph generated by $G(n, p)$. The following property is derived from theorems in [7, p.65∼67]:

Theorem 1 *A graph generated by $G(n, p)$ has average degree $d = n \cdot p$. If $d < 1$, then with high probability the largest component in the graph has size less than $O(\log n)$. If $d > 1$, with high probability the graph will contain a giant component with size at the order of $O(n)$.*

For a group of bot-users that share a set of IPs, the average degree will be larger than one. According to the above theorem, the giant component will appear with a high probability. On the other hand, normal users rarely share IPs, and the average degree will be far less than one when the number of vertices is large. The resulted graph of normal users will therefore contain isolated vertices and/or small components, as we observe in our case. In other words, the theorem interprets the appearance of giant components we have observed in subsection 4.1.1. Based on the theorem, the sizes of the components can serve as guidelines for bot-user pruning and grouping (discussed in subsection 4.2.2 and 4.2.3).

4.2 Bot-User Detection Algorithm

As we have shown in section 4.1, a bot-user group forms a connected component in the user-user graph. Intuitively one could identify bot-user groups by simply extracting the connected components from the user-user graph generated with some predefined threshold T (the least number of shared IPs for two vertices to be connected by an edge). In reality, however, we need to handle the following issues:

- It is hard to choose a single fixed threshold of T. As we can see from Figure 2, different spamming strategies may lead to different transition points.
- Bot-users from different bot-user groups may be in the same connected component. This happens due to: 1) bot-users may be shared by different spammers, and 2) a bot may be controlled by different spammers.
- There may exist connected components of normal users. For example, mobile device users roaming around different locations will be assigned IP addresses from different ASs, and therefore appeared as a connected component.

To handle these problems, we propose a hierarchical algorithm to extract connected components, followed by a pruning and grouping procedure to remove false positives and to separate mixed bot-user groups.

4.2.1 Hierarchical Connected-Component Extraction

Algorithm 1 describes a recursive function Group_Extracting that extracts a set of connected components from a user-user graph in a hierarchical way. Having such a recursive process avoids using a fixed threshold T, and is potentially robust to different spamming strategies.

Using the original user-user graph as input, BotGraph begins with applying Group_Extracting(G, T) to the graph with $T = 2$. In other words, the algorithm first identifies all the connected components with edge weight $w \geq 2$. It then recursively increases w to extract connected subcomponents. This recursive process continues until the number of nodes in the connected component is smaller than a pre-set threshold M ($M = 100$ in our experiments). The final output of the algorithm is a hierarchical tree of the connected components with different edge weights.

```
  procedure Group_Extracting($G$, $T$)
1 Remove all the edges with weight $w < T$ from $G$
  and suppose we get $G'$;
2 Find out all the connected subgraphs $G_1$, $G_2$, $\cdots$,
  $G_k$ in $G'$;
3 for $i = 1 : k$ do
4     Let $|G_k|$ be the number of nodes in $G_k$;
5     if $|G_k| > M$ then
6         Output $G_k$ as a child node of $G$ ;
7         Group_Extracting($G_k$, $T + 1$) ;
      end
  end
```

Algorithm 1: A Hierarchical algorithm for connected component extraction from a user-user graph.

4.2.2 Bot-User Pruning

For each connected component output by Algorithm 1, we want to compute the level of confidence that the set of users in the component are indeed bot-users. In particular, we need to remove from the tree (output by Al-

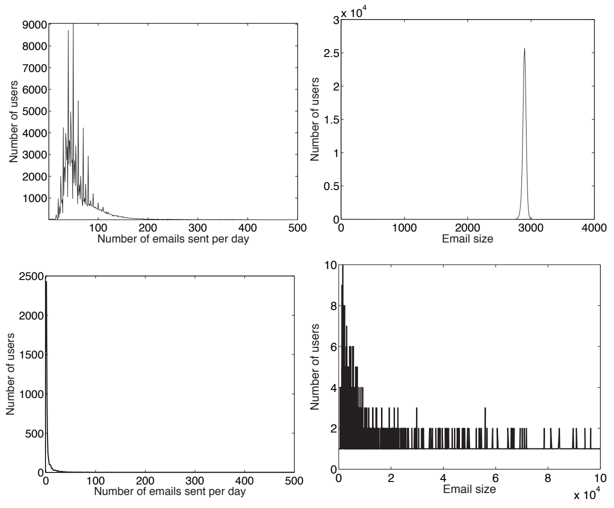


Figure 3: Histograms of (1) number of emails sent per day and (2) email size. First row: aggressive bot-users; second row: normal users.

gorithm 1) the connected components involving mostly legitimate/normal users.

A major difference between normal users and bot-users is the way they send emails. More specifically, normal users usually send a small number of emails per day on average, with different email sizes. On the other hand, bot-users usually send many emails per day, with identical or similar email sizes, as they often use a common template to generate spam emails. It may be difficult to use such differences in email-sending statistics to classify bot-accounts individually. But when a group of accounts are viewed in aggregate, we can use these statistics to estimate how likely the entire group are bot-users. To do so, for each component, BotGraph computes two histograms from a 30-day email log:

- h_1: the numbers of emails sent per day per user.
- h_2: the sizes of emails.

Figure 3 shows two examples of the above two histograms, one computed from a component consisting of bot-users (the first row), the other from a component of normal users (the second row). The distributions are clearly different. Bot-users in a component sent out a larger number of emails on average, with similar email sizes (around 3K bytes) that are visualized as the peak in the email-size histogram. Most normal users sent a small number of emails per day on average, with email sizes distributing more uniformly. BotGraph normalizes each histogram such that its sum equals to one, and computes two statistics, s_1 and s_2, from the normalized histograms to quantify their differences:

- s_1: the percentage of users who sent more than 3 emails per day;
- s_2: the areas of peaks in the normalized email-size histogram, or the percentage of users who sent out emails with a similar size.

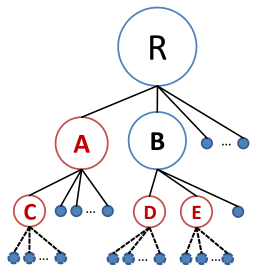


Figure 4: An example of extracting bot-user groups using the random graph model.

Since the histograms are normalized, both s_1 and s_2 are in the range of $[0, 1]$ and can be used as confidence measures. A large confidence value means that the majority of the users in the connected component are bot-users. We use only s_1 to choose the candidates of bot-user components, because s_1 represents a more robust feature. We use s_2 together with other features (e.g., account naming patterns) for validation purpose only (see Section 6).

In the pruning process, BotGraph traverses the tree output by Algorithm 1. For each node in the tree, it computes s_1, the confidence measure for this node to be a bot-user component, and removes the node if s_1 is smaller than a threshold S. In total, fewer than 10% of Hotmail accounts sent more than 3 emails per day, so intuitively, we can set the threshold $S = 0.1$. In order to minimize the number of false positive users, we conservatively set the threshold $S = 0.8$, i.e., we only consider nodes where at least 80% of users sent more than 3 emails per day as suspicious bot-user groups (discussed further in Section 6.2).

4.2.3 Bot-User Grouping

After pruning, a candidate connected-component may contain two or more bot-user groups. BotGraph proceeds to decompose such components further into individual bot-user groups. The correct grouping is important for two reasons:

- We can extract validation features (e.g., s_2 mentioned above and patterns of account names) more accurately from individual bot-user groups than from a mixture of different bot-user groups.
- Administrators may want to investigate and take different actions on different bot-user groups based on their behavior.

We use the random graph model to guide the process of selecting the correct bot-user groups. According to the random graph model, the user-user subgraph of a bot-user group should consist of a giant connected-component plus very small components and/or isolated vertices. So BotGraph traverses the tree again to select tree nodes that are consistent with such random graph property. For each node V being traversed, there are two cases:

- V's children contain one or more giant components whose sizes are $O(N)$, where N is the number of users in node V;

- V's children contain only isolated vertices and/or small components with size of $O(\log(N))$.

For case 1, we recursively traverse each subtree rooted by the giant components. For case 2, we stop traversing the subtree rooted at the V. Figure 4 illustrates the process. Here the root node R is decomposed into two giant components A and B. B is further decomposed into another two giant components D and E, while A is decomposed into one giant component C. The giant component disappears for any further decomposition, indicated by the dash-lines. According to the theory, A, C, D, and E are bot-user groups. If a node is chosen as a bot-user group, the sub-tree rooted at the chosen node is considered belonging to the same bot-user group. That is, if we pick A, we disregard its child C as it is a subcomponent of A.

5 Large-scale Parallel Graph Construction

The major challenge in applying BotGraph is the construction of a large user-user graph from the Hotmail login data – the first stage of our graph-based analysis described in Section 3.2. Each record in the input log data contains three fields: *UserID*, *IPAddress*, and *Login-Timestamp*. The output of the graph construction is a list of edges in the form of *UserID*$_1$, *UserID*$_2$, and *Weight*. The number of users on the graph is over 500 million based on a month-long login data (220 GB), and this number is increasing as the Hotmail user population is growing. The number of edges of the computed graph is on the order of hundreds of billions. Constructing such a large graph using a single computer is impractical. An efficient, scalable solution is required so that we could detect attacks as early as possible in order to take timely reactive measures.

For data scalability, fault tolerance, and ease of programming, we choose to implement BotGraph using Dryad/DryadLINQ, a powerful programming environment for distributed data-parallel computing. However, constructing a large user-user graph using Dryad/DryadLINQ is non-trivial. This is because the resulting graph is extremely *large*, therefore a straightforward parallel implementation is inefficient in performance. In this section, we discuss in detail our solutions. We first present both a simple parallelism method and a selective filtering method, and then describe several optimization strategies and their performance impacts. We also discuss several important issues arising in the system implementation, such as data partitioning, data processing flow, and communication methods. Using a one-month log as input, our current implementation can construct a graph with tens of millions of nodes in 1.5 hours using a 240-machine cluster. During this process, BotGraph filters out weight one edges, and the remaining number of edges for the next-stage processing is around 8.6 billion.

We also implemented the second stage of finding connected components using Dryad/DryadLINQ. This stage can be solved using a divide and conquer algorithm. In particular, one can divide the graph edges into multiple partitions, identify the connected subgraph components in each partition, and then merge the incomplete subgraphs iteratively. To avoid overloading the merging node, instead of sending all outputs to a single merging node, each time we merge two results from two partitions. This parallel algorithm is both efficient and scalable. Using the same 240-machine cluster in our experiments, this parallel algorithm can analyze a graph with 8.6 billion edges in only 7 minutes — 34 times faster than the 4 hour running time by a single computer. Given our performance bottleneck is at the first stage of graph construction instead of graph analysis, we do not further elaborate this step.

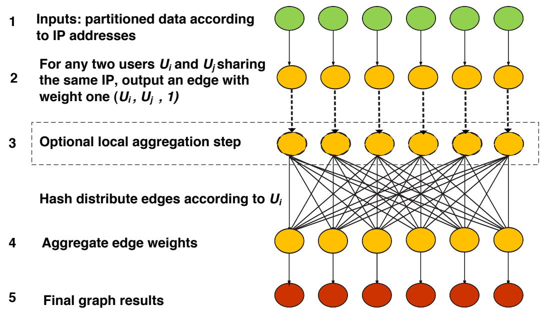

1 Inputs: partitioned data according to IP addresses
2 For any two users U_i and U_j sharing the same IP, output an edge with weight one (U_i, U_j , 1)
3 Optional local aggregation step
Hash distribute edges according to U_i
4 Aggregate edge weights
5 Final graph results

Figure 5: Process flow of Method 1.

5.1 Two Implementation Methods

The first step in data-parallel applications is to partition data. Based on the ways we partition the input data, we have different data processing flows in implementing graph construction.

5.1.1 Method 1: Simple Data Parallelism

Our first approach is to partition data according to IP address, and then to leverage the well known Map and Reduce operations to straightforwardly convert graph construction into a data-parallel application.

As illustrated in Figure 5, the input dataset is partitioned by the user-login IP address (Step 1). During the Map phase (Step 2 and 3), for any two users U_i and U_j sharing the same IP-day pair, where the IP address is from Autonomous System AS_k, we output an edge with weight one $e = (U_i, U_j, AS_k)$. Only edges pertaining to different ASes need to be returned (Step 3). To avoid outputting the same edge multiple times, we use a local hash table to filter duplicate edges.

After the Map phase, all the generated edges (from all partitions) will serve as inputs to the Reduce phase. In particular, all edges will be hash partitioned to a set of processing nodes for weight aggregation using (U_i, U_j) tuples as hash keys (Step 4) . Obviously, for those user pairs that only share one IP-day in the entire dataset, there is only one edge between them. So no aggregation can be performed for these weight one edges. We will show later in Figure 7 that weight one edges are the dominate source of graph edges. Since BotGraph focuses on only

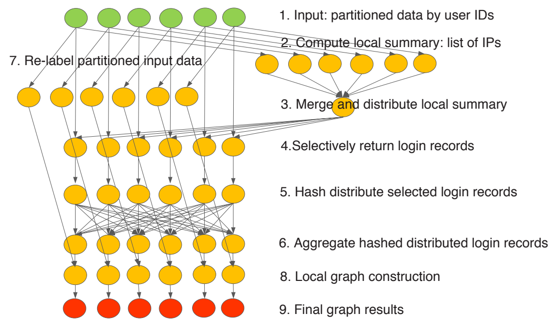

1. Input: partitioned data by user IDs
2. Compute local summary: list of IPs
7. Re-label partitioned input data
3. Merge and distribute local summary
4.Selectively return login records
5. Hash distribute selected login records
6. Aggregate hashed distributed login records
8. Local graph construction
9. Final graph results

Figure 6: Process flow of Method 2.

edges with weight two and above, the weight one edges introduce unnecessary communication and computation cost to the system. After aggregation, the outputs of the Reduce phase are graph edges with aggregated weights.

5.1.2 Method 2: Selective Filtering

An alternative approach is to partition the inputs based on user ID. In this way, for any two users that were located in the same partition, we can directly compare their lists of IP-day pairs to compute their edge weight. For two users whose records locate at different partitions, we need to ship one user's records to another user's partition before computing their edge weight, resulting in huge communication costs.

We notice that for users who do not share any IP-day keys, such communication costs can be avoided. That is, we can reduce the communication overhead by *selectively filtering* data and distributing only the related records across partitions.

Figure 6 shows the processing flow of generating user-user graph edges with such an optimization. For each partition p_i, the system computes a *local summary* s_i to represent the union of all the IP-day keys involved in this partition (Step 2). Each local summary s_i is then distributed across all nodes for selecting the relevant input records (Step 3). At each partition $p_j (j \neq i)$, upon receiving s_i, p_j will return all the login records of users who shared the same IP-day keys in s_i. This step can be further optimized based on the edge threshold w: if a user in p_j shares fewer than w IP-day keys with the summary s_i, this user will not generate edges with weight at least w. Thus only the login records of users who share at least w IP-day keys with s_i should be selected and sent to partition p_i (Step 4)). To ensure the selected user records will be shipped to the right original partition, we add an additional label to each original record to denote their partition ID (Step 7). Finally, after partition p_i receives the records from partition p_j, it joins these remote records with its local records to generate graph edges (Step 8 and 9).

Other than Map and Reduce, this method requires two additional programming interface supports: the operation to join two heterogeneous data streams and the operation to broadcast a data stream.

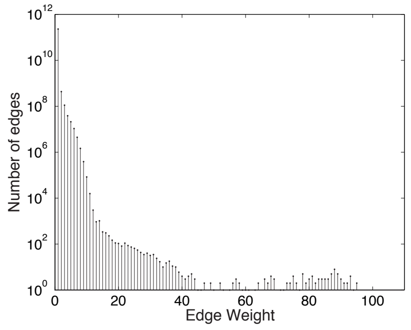


Figure 7: Edge weight distribution.

5.1.3 Comparison of the Two Methods

In general, Method 1 is simple and easy to implement, but Method 2 is more optimized for our application. The main difference between the two data processing flows is that Method 1 generates edges of weight one and sends them across the network in the Reduce phase, while Method 2 directly computes edges with weight w or more, with the overhead of building a local summary and transferring the selected records across partitions. Figure 7 shows the distribution of edge weights using one-month of user login records as input. Here, the number of weight one edges is almost three orders of magnitude more than the weight two edges. In our botnet detection, we are interested in edges with a minimum weight two because weight one edges do not show strong correlated login activities between two users. Therefore the computation and communication spent on generating weight one edges are not necessary. Although in Method 1, Step 3 can perform local aggregation to reduce the number of duplicated weight one edges, local aggregation does not help much as the number of unique weight one edges dominates in this case.

Given our implementation is based on the existing distributed computing models such as MapReduce and DryadLINQ, the amount of intermediate results impacts the performance significantly because these programming models all adopt disk read/write as cross-node communication channels. Using disk access as communication is robust to failures and easy to restart jobs [6, 29]. However, when the communication cost is large such as in our case, it becomes a major bottleneck of the overall system running time. To reduce this cost, we used a few optimization strategies and will discuss them in the next subsection. Completely re-designing or customizing the underlying communication channels may improve the performance in our application, but is beyond the scope of this paper.

Note the amount of cross-node communication also depends on the cluster size. Method 1 results in a constant communication overhead, i.e., the whole edge set, regardless of the number of data partitions. But for Method 2, when the number of computers (hence the number of data partitions) increases, both the aggregated local summary size and the number of user-records to be shipped

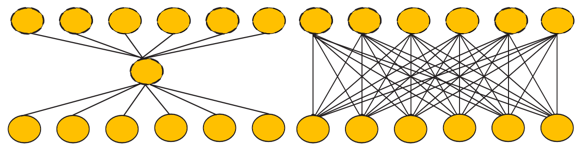


Figure 8: (a) Default query execution plan (b) Optimized query execution plan.

increase, resulting in a larger communication overhead. In the next subsections, we present our implementations and evaluate the two different methods using real-data experiments.

5.2 Implementations and Optimizations

In our implementation, we have access to a 240-machine cluster. Each machine is configured with an AMD Dual Core 4.3G CPU and 16 GB memory. As a pre-processing step, all the input login records were hash partitioned evenly to the computer cluster using the DryadLINQ built-in hash-partition function.

Given the Hotmail login data is on the order of hundreds of Gigabytes, we spent a number of engineering efforts to reduce the input data size and cross-node communication costs. The first two data reduction strategies can be applied to both methods. The last optimization is customized for Method 2 only.

1. User pre-filtering: We pre-filter users by their login AS numbers: if a user has logged in from IP addresses across multiple ASes in a month, we regard this user as a suspicious user candidate. By choosing only suspicious users (using 2 ASes as the current threshold) and their records as input, we can reduce the number of users to consider from over 500 million (about 200-240GB) to about 70 million (about 100GB). This step completes in about 1-2 minutes.

2. Compression: Given the potential large communication costs, BotGraph adopts the DryadLINQ provided compression option to reduce the intermediate result size. The use of compression can reduce the amount of cross-node communication by 2-2.5 times.

3. Parallel data merge: In Method 2, Step 3 merges the local IP-day summaries generated from every node and then broadcasts the aggregated summary to the entire cluster. The old query plan generated by DryadLINQ is shown in Figure 8 (a), where there exists a single node that performs data aggregation and distribution. In our experiments, this aggregating node becomes a big bottleneck, especially for a large cluster. So we modified DryadLINQ to generate a new query plan that supports parallel data aggregation and distribution from every processing node (Figure 8 (b)). We will show in Section 5.3 that this optimization can reduce the broadcast time by 4-5 times.

	Communication data size	Total running time
Method 1	12.0 TB	> 6 hours
Method 2	1.7 TB	95 min

Table 1: Performance comparison of the two methods using the 2008-dataset.

	Communication data size	Total running time
Method 1 (no comp.)	2.71 TB	135 min
Method 1 (with comp.)	1.02 TB	116 min
Method 2 (no comp.)	460 GB	28 min
Method 2 (with comp.)	181 GB	21 min

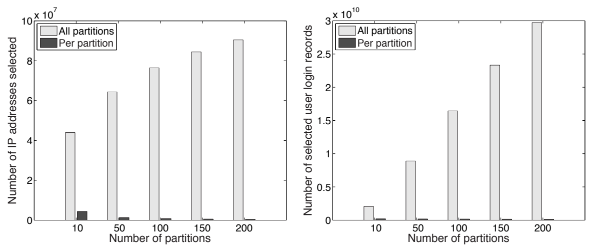
Table 2: Performance comparison of the two methods using a subset of the 2008-dataset.

5.3 Performance Evaluation

In this section, we evaluate the performance of our implementations using a one-month Hotmail user-login log collected in Jan 2008 (referred to as the 2008-dataset). The raw input data size is 221.5 GB, and after pre-filtering, the amount of input data is reduced to 102.9 GB. To use all the 240 machines in the cluster, we generated 960 partitions to serve as inputs to Method 1 (so that the computation of each partition fits into memory), and generated 240 partitions as inputs to Method 2. With compression and parallel data merge both enabled, our implementation of Method 2 finishes in about 1.5 hours using all the 240 machines, while Method 1 cannot finish within the maximum 6 hour quota allowed by the computer cluster (Table 1). The majority of time in Method 1 is spent on the second Reduce step to aggregate a huge volume of intermediate results. For Method 2, the local summary selection step generated about 5.8 GB aggregated IP-day pairs to broadcast across the cluster, resulting 1.35 TB out of the 1.7 TB total traffic.

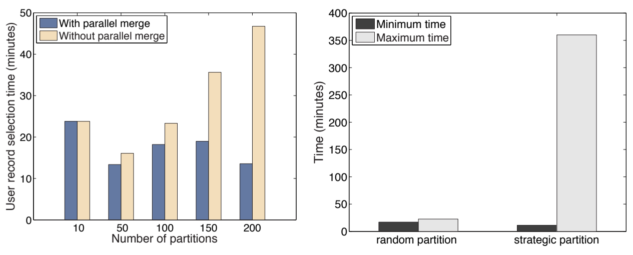
In order to benchmark performance, we take a smaller dataset (about 1/5 of the full 2008-dataset) that Method 1 can finish within 6 hours. Table 2 shows the communication costs and the total running time using the 240 machine cluster. While Method 1 potentially has a better scalability than Method 2 as discussed in Section 5.1.3, given our practical constraints on the cluster size, Method 2 generates a smaller amount of traffic and outperforms Method 1 by about 5-6 times faster. The use of compression reduces the amount of traffic by about 2-3 times, and the total running time is about 14-25% faster.

To evaluate the system scalability of Method 2, we vary the number of data partitions to use different number of computers. Figure 9 shows how the communication overheads grow. With more partitions, the amount of data generated from each processing node slightly decreases, but the aggregated local summary data size increases (Figure 9 (a)). This is because popular IP-day pairs may appear in multiple data partitions and hence in the aggregated summary multiple times. Similarly, the same user login records will also be shipped across a larger number of nodes, increasing the communication



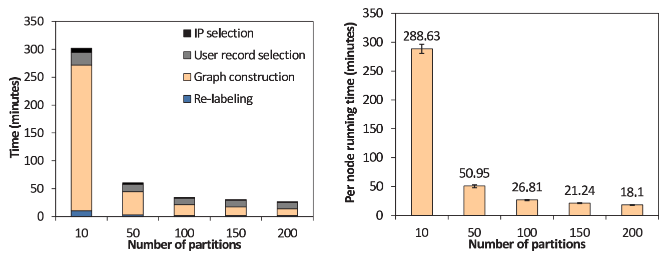
(a) (b)

Figure 9: Communication data size as we vary the number of input data partitions (a) Local summary size in terms of the number of IP-day keys. (b) Total number of selected user login records to be sent across the network.



(a) (b)

Figure 11: (a) The processing time of user-record selection with and without parallel data merge. (b) Minimal and maximum running time of partitions with and without strategic data partitioning.



(a) (b)

Figure 10: Running time as we vary the number of input data partitions for Method 2. (a) Total running time of all partitions. (b) The running of each partition. The error bars show the max and the min running time across all partitions.

costs as the system scales (Figure 9 (b)).

Even though the communication costs increase, the total running time is still reduced with a larger cluster size. Figure 10 (a) shows the total running time and its breakdown across different steps. When the cluster size is small (10 partitions), a dominant amount of time is spent on computing the graph edges. As the system scales, this portion of time decreases sharply. The other three steps are I/O and network intensive. Their running time slightly decreases as we increase the number of partitions, but the savings get diminished due to the larger communication costs. Figure 10 (b) shows the average running time spent on processing each partition, and its variations are very small.

We now examine the benefits of adopting parallel data merge. The purpose of parallel data merge is to remove the bottleneck node that performs data aggregation and broadcasting. Since it is difficult to factor out the network transfer time savings alone (network, disk I/O, and computation are pipelined), we compare the time spent on the user record selection step (Figure 11 (a)). This optimization can reduce the processing latency significantly as the cluster size increases (75% reduction in the 200 node scenario). Without parallel data merge, the processing time increases almost linearly, but with this optimization, the amount of time remains roughly constant.

For Method 2, one reason for the large communication costs is that for botnet users, their graph component is both large and dense. Therefore, one potential optimization technique is to strategically partition the login records. Intuitively, we can reduce the communication costs if we pre-group users so that users who are heavily connected are placed in one partition, and users who are placed in different partitions have very few edges between them. If so, Step 4 in Method 2 will return only a small number of records to ship across different nodes. Surprisingly, we found this strategy actually induced negative impact on the system performance.

Figure 11 (b) shows the graph construction time spent at a processing node with and without strategic data partitioning. We chose the 240 input data partition scenario and use the full dataset to illustrate the performance difference. In the first case, we evenly distributed login records by hashing user IDs. In the second case, we chose a large botnet user group with 3.6M users and put all their login records evenly across 5 partitions, with the remaining data evenly distributing across the remaining partitions. This scenario assumes the best prior knowledge of user connections. Although in both cases, the total amount of input data in each partition is roughly uniform, we observe a big difference between the maximum and minimum time in computing the edges across nodes. Without strategic partitioning, the maximum and minimum processing time is very close. In contrast, strategic partitioning caused a huge degree of unbalance in workload, resulting in much longer total job running time.

6 Bot-user Detection and Validation

We use two month-long datasets as inputs to our system: a 2007-dataset collected in Jun 2007, and a 2008-dataset collected in Jan 2008. Each dataset includes two logs: a Hotmail login log (format described in Section 5) and a Hotmail signup log. Each record in the signup log contains a user-ID, the remote IP address used for signup, and the signup timestamp. For each dataset, we run our EWMA-based anomaly detection on the signup log and run our graph based detection on the login log. Using both components, BotGraph detected tens of millions of bot users and millions of botnet IPs. Table 3 summarizes the results for both months. We present the detailed results and perform evaluations next.

Month	06/2007	01/2008
# of bot-users	5.97M	20.58M
# of bot-IPs	2.71M	1.84M

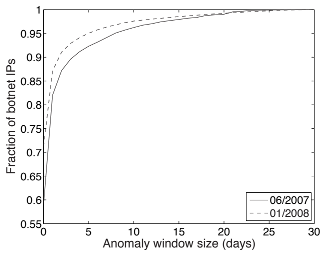
Table 3: Total bot-users and bot IP addresses detected using both history based detection and user-user graph.

Month	06/2007	01/2008
# of bot IPs	82,026	240,784
# of bot-user accounts	4.83 M	16.41 M
Avg. anomaly window	1.45 day	1.01 day

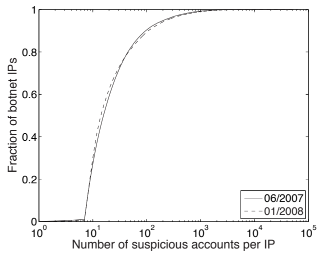
Table 4: History based detection of bot IP addresses and bot-user accounts.

Month	06/2007	01/2008
# of bot-groups	13	40
# of bot-accounts	2.66M	8.68M
# of unique IPs	2.69M	1.60M

Table 5: Bot IP addresses and bot-user accounts detected by user-user graphs.

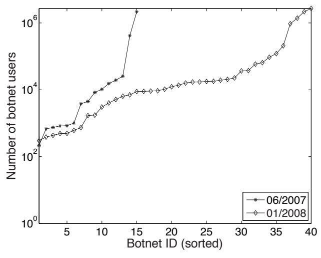


(a)

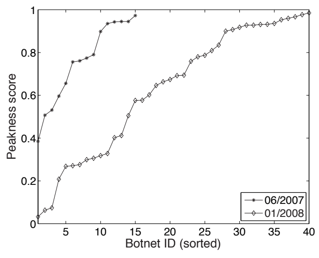


(b)

Figure 12: (a) Cumulative distribution of anomaly window size in terms of number of days. (b) Cumulative distribution of the number of accounts signed up per suspicious IP.



(a) Accounts in each group



(b) Group peakness scores

Figure 13: Bot-user group properties: (a) The the number of users per group, (b) The peakness score of each group, reflecting whether there exists a strong sharp peak for the email size distribution.

6.1 Detection Using Signup History

Table 4 shows that the EWMA algorithm detected 21.2 million bot-user accounts when applied to the two Hotmail signup logs. Comparing Jan 2008 with Jun 2007, both the number of bot IPs and the signed-up bot-users increased significantly. In particular, the total number of bot-accounts signed up in Jan 2008 is more than three times the number in Jun 2007. Meanwhile, the anomaly window is shortened from an average of 1.45 days to 1.01 days, suggesting each attack became shorter in Jan 2008.

Figure 12 (a) shows the cumulative distribution of the anomaly window sizes associated with each bot IP address. A majority (80% - 85%) of the detected IP addresses have small anomaly windows, ranging from a few hours to one day, suggesting that many botnet signup attacks happened in a burst.

Figure 12 (b) shows the cumulative distributions of the number of accounts signed up per bot IP. As we can see, the majority of bot IPs signed up a large number of accounts, even though most of them have short anomaly windows. Interestingly, the cumulative distributions derived from Jun 2007 and Jan 2008 overlap well with each other, although we observed a much larger number of bot IPs and bot-users in Jan 2008. This indicates that the overall bot-user signup activity patterns still remain similar perhaps due to the reuse of bot-account signup tools/software.

6.2 Detection by User-User Graph

We apply the graph-based bot-user detection algorithm on the Hotmail login log to derive a tree of connected components. Each connected component is a set of bot-user candidates. We then use the procedures described in Section 4.2.2 to prune the connected components of normal users. Recall that in the pruning process, we apply a threshold on the confidence measure of each component (computed from the "email-per-day" feature) to remove normal user components. In our experiments, the confidence measures are well separated: most of the bot-groups have confidence measures close to 1, and a few groups are between 0.4 and 0.6. We observe a wide margin around confidence measure of 0.8, which we choose as our threshold. As discussed in Section 4.2.2, this is a conservative threshold and is in-sensitive to noises due to the wide margin. For any group that has a confidence measure below 0.8, we regard it as a normal user group and prune it from our tree.

Table 5 shows the final detection results after pruning and grouping. Both the number of bot-users and the number of bot IP addresses are on the order of millions — a non-trivial fraction of all the users and IP addresses observed by Hotmail. We find the two sets of bot-users detected in two months hardly overlap. These accounts were stealthy ones, each sending out only a few to tens of spam emails during the entire month. Therefore, it is difficult to capture them by looking for aggressive sending patterns. Due to their large population, detecting and sanitizing these users are important both to save Hotmail resources and to reduce the amount of spam sent to the Internet. Comparing Jan 2008 with Jun 2007, the number of bot-users tripled, suggesting that using Web portals as a spamming media has become more popular.

Now we study the properties of bot-users at a group level. Figure 13 (a) shows that the number of users in each group ranges from thousands to millions. Comparing Jan 2008 with Jun 2007, although the largest bot-user group remains similar in size, the number of groups increased significantly. This confirms our previous observation that spammers are more frequently using Web email accounts for spam email attacks.

We next investigate the email sending patterns of the detected bot user groups. We are interested in whether there exists a strong peak of email sizes. We use the peakness score metric s_2 (defined in Section 4.2.2) to quantify the degree of email size similarity for each group. Figure 13 (b) shows the distributions of s_2 in sorted order. A majority of groups have peakness scores higher than 0.6, meaning that over 60% of their emails have similar sizes. For the remaining groups, we performed manual investigation and found they have multiple peaks, resulting in lower scores. The similarity of their email sizes is a strong evidence of correlated email sending activities.

In the next two sub-sections, we explore the quality of the total captured 26 million bot-users. First, we examine whether they are known bad and how many of them are our new findings. Second, we estimate our detection false positive rates.

6.3 Known Bot-users vs. New Findings

We evaluate our detected bot-users against a set of known spammer users reported by other email servers in Jan 2008 [4].

Denote H as the set of bot-users detected by signup history using EWMA, K_s as the set of known spammer accounts signed up in the month that we study, and $K_s \cap H$ as the intersection between H and K_s. The ratio of $\frac{K_s \cap H}{H}$ represents the percentage of captured bot-users that are previously known bad. In other words, $1 - \frac{K_s \cap H}{H}$ is our new findings. The ratio of $\frac{K_s \cap H}{K_s}$ denotes the recall of our approach. Table 6 shows that, in Jun 2007, 85.15% of the EWMA-detected bot-user detected are already known bad, and the detected bot-user covers a significant fraction of bad account, i.e., recall = 67.96%. Interestingly, Jan 2008 yields quite different results. EWMA is still able to detect a large fraction of known bad account. However, only 8.17% of detected bad-users were reported to be bad. That means 91.83% of the captured spamming accounts are our new findings.

We apply a similar study to the bot-users detected by the user-user graph. Denote K_l as the set of known spammers users that log in from at least 2 ASes, L as the set of bot-users detected using our user-user graph based approach, and $K_l \cap L$ as the intersect between K_l and L. Again we use the ratios of $\frac{K_l \cap L}{L}$ and $\frac{K_l \cap L}{K_l}$ to evaluate our result L, as shown in Table 7. Using our graph-based approach, the recall is higher. In total, we were able to detect 76.84% and 85.80% of known spammer users in Jun 2007 and Jan 2008, respectively. Similar to EWMA, the graph-based detection also identified a large number (54.10%) of previously unknown bot-accounts in Jan 2008. This might be because these accounts are new ones and haven't been used aggressively to send out a massive amount of spam emails yet. So, they are not yet reported by other mail servers as of Jan 2008. The ability of detecting bot-accounts at an early stage is important to to give us an upper hand in the anti-spam battle.

[4]These users were complained of having sent outbound spam emails.

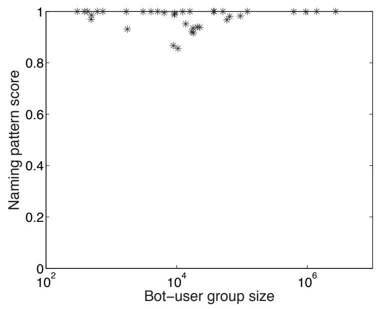


Figure 14: Validation of login-graph detected bot-users using naming scores.

6.4 False Positive Analysis

In the previous subsection, we analyzed the overlap between our results and the set of known bad accounts. For the remaining ones, validation is a challenging task without the ground truth. We examine the following two account features to estimate the false positive rates: naming patterns and signup dates.

6.4.1 Naming Patterns

For the identified groups, we found almost every group follows a very clear user-name template, for example, a fixed-length sequence of alphabets mixed with digits [5]. Examples of such names are "w9168d4dc8c5c25f9" and "x9550a21da4e456a2".

To quantify the similarity of account names in a group, we introduce a *naming pattern score*, which is defined as the largest fraction of users that follow a single template. Each template is a regular expression derived by a regular expression generation tool [27]. Since many accounts detected in Jun 2007 were known bad and hence cleaned by the system already, we focus on bot-user groups detected in Jan 2008.

Figure 14 shows the naming score distribution. A majority of the bot-user groups have close to 1 naming pattern scores, indicating that they were signed up by spammers using some fixed templates. There are only a few bot-user groups with scores lower than 0.95. We manually looked at them and found that they are also bad users, but the user names come from two naming templates. It is possible that our graph-based approach mixed two groups, or the spammers purchased two groups of bot-users and used them together. Overall, we found in total only 0.44% of the identified bot-users do not strictly follow the naming templates of their corresponding groups.

6.4.2 Signup Dates

Our second false positive estimate is based on examining the signup dates of the detected bot-users. Since the Web-account abuse attack is recent and started in summer 2007, we regard all the accounts signed up before 2007 as legitimate accounts. Only 0.08% of the identified bot-users were signed up before year 2007. To cal-

[5]Note it is hard to directly use the naming pattern itself to identify spamming accounts due to the easy countermeasures.

$I = K_s \cap H$	06/2007	01/2008
I/H	85.15%	8.17%
I/K_s	67.96%	52.41%

Table 6: Comparing bot-users detected by signup history using EWMA with known spammer user sets, using the ratios of $\frac{K_s \cap H}{H}$ and $\frac{K_s \cap H}{K_s}$. See text for the definition of H and K_s.

$I = K_l \cap L$	06/2007	01/2008
I/L	90.95%	45.9%
I/K_l	76.84%	85.8%

Table 7: Comparing bot-users detected by user-user graph with known spammer user sets, using the ratios of $\frac{K_l \cap L}{L}$ and $\frac{K_l \cap L}{K_l}$. See text for the definition of K_l and L.

ibrate our results against the entire user population. We look at the sign up dates of all users in the input dataset. About 59.1% of the population were signed up before 2007. Assuming the normal user signup-date distributions are the same among the overall population and our detected user set, we adjust the false positive rate to be $0.08\%/59.1\% = 0.13\%$

The above two estimations suggest that the false positive of BotGraph is low. We conservatively pick the higher one 0.44% as our false positive rate estimate.

7 Discussion

In this paper, we demonstrated that BotGraph can detect tens of millions of bot-users and millions of bots. With this information, operators can take remedy actions and mitigate the ongoing attacks. For bot-users, operators can block their accounts to prevent them from further sending spam, or apply more strict policies when they log in (e.g., request them to do additional CAPTCHA tests). For detected bot IP addresses, one approach is to blacklist them or rate limit their login activities, depending on whether the corresponding IP address is a dynamically assigned address or not. Effectively throttling botnet attacks in the existence of dynamic IP addresses is ongoing work.

Attackers may wish to evade the BotGraph detection by developing countermeasures. For example, they may reduce the number of users signed up by each bot. They may also mimic the normal user email-sending behavior by reducing the number of emails sent per account per day (e.g., fewer than 3). Although mimicking normal user behavior may evade history-based change detection or our current thresholds, these approaches also significantly limit the attack scale by reducing the number of bot-accounts they can obtain or the total number of spam emails to send. Furthermore, BotGraph can still capture the graph structures of bot-user groups from their login activity to detect them.

A more sophisticated evasion approach may bind each bot-user to only bots in one AS, so that our current implementation would pre-filter them by the two AS threshold. To mitigate this attack, BotGraph may revise the edge weight definition to look at the number of IP prefixes instead of the number of ASes. This potentially pushes the attacker countermeasures to be more like a fixed IP-account binding strategy. As discussed in Section 3.2, binding each bot-user to a fixed bot is not desirable to the spammers. Due to the high botnet churn rate, it would result in a low bot-user utilization rate. It also makes attack detection easier by having a fixed group of aggressive accounts on the same IP addresses all the time. If one of the bot-accounts is captured, the entire group can be easily revealed. A more generalized solution is to broaden our edge weight definition by considering additional feature correlations. For example, we can potentially use email sending patterns such as the destination domain [24], email size, or email content patterns (e.g., URL signatures [27]). As ongoing work, we are exploring a larger set of features for more robust attack detection.

In addition to using graphs, we may also consider other alternatives to capture the correlated user activity. For example, we may cluster user accounts using their login IP addresses as feature dimensions. Given the large data volume, how to accurately and efficiently cluster user accounts into individual bot-groups remains a challenging research problem.

It is worth mentioning that the design and implementation of BotGraph can be applied in different areas for constructing and analyzing graphs. For example, in social network studies, one may want to group users based on their buddy relationship (e.g., from MSN or Yahoo messengers) and identify community patterns. Finally, although our current implementations are Dryad/DryadLINQ specific, we believe the data processing flows we propose can be potentially generalized to other programming models.

8 Conclusion

We designed and implemented $BotGraph$ for Web mail service providers to defend against botnet launched Web-account abuse attacks. BotGraph consists of two components: a history-based change-detection component to identify aggressive account signup activities and a graph-based component to detect stealthy bot-user login activities. Using two-month Hotmail logs, $BotGraph$ successfully detected more than 26 million botnet accounts. To process a large volume of Hotmail data, BotGraph is implemented as a parallel Dryad/DryadLINQ application running on a large-scale computer cluster. In this paper, we described our implementations in detail and presented performance optimization strategies. As general-purpose distributed computing frameworks have become increasingly popular for processing large datasets, we believe our experience will be useful to a wide category of applications for constructing and analyzing large graphs.

9 Acknowledgement

We would like to thank Mihai Budiu, Úlfar Erlingsson, and Dennis Fetterly for their help on DryadLINQ. We are grateful to the Hotmail operation team for providing us with data and valuable feedbacks. We would also like to thank Nick Feamster and anonymous reviewers for their constructive comments.

A EWMA based Aggressive Signup Detection

Exponentially Weighted Moving Average (EWMA) is a well known moving average based algorithm to detect sudden changes. EWMA is both simple and effective, and has been widely used for anomaly detection [12].

Given a time series data, let the observation value at time t be Y_t. Let S_t be the predicted value at time t and α ($0 \leq \alpha \leq 1$) be the weighting factor, EWMA predicts S_t as

$$S_t = \alpha \times Y_{t-1} + (1 - \alpha) \times S_{t-1} \quad (1)$$

We define the absolute prediction error E_t and the relative prediction error R_t as:

$$E_t = Y_t - S_t, \quad R_t = Y_t / \max(S_t, \epsilon) \quad (2)$$

where ϵ is introduced to avoid the divide-by-zero problem. A large prediction error E_t or R_t indicates a sudden change in the time series data and should raise an alarm. When the number of new users signed up has dropped to the number before the sudden change, the sudden change ends. We define the time window between the start and the end of a sudden change as *the anomaly window*. All the accounts signed up during this anomaly window are suspicious bot-users.

In our implementation, we consider the time unit of a day, and hence E_t is the predicted number of daily signup accounts. For any IP address, if both $E_t > \delta_E$ and $R_t > \delta_R$, we mark day t as the start of its anomaly window. From a two-year Hotmail signup log, we derive the 99%-tile of the daily number of account signups per IP address. To be conservative, We set the threshold δ_E to be twice this number to rule out non-proxy normal IPs. For proxies, the relative prediction error is usually a better metric to separate them from bots. It is very rare for a proxy to increase its signup volume by 4 times overnight. So we conservatively set δ_R to 4.

References

[1] D. S. Anderson, C. Fleizach, S. Savage, and G. M. Voelker. Spamscatter: Characterizing internet scam hosting infrastructure. In *USENIX Security Symposium*, 2007.

[2] Apache. Hadoop. http://lucene.apache.org/hadoop/.

[3] http://blogs.zdnet.com/security/?p=1835.

[4] K. Chiang and L. Lloyd. A case study of the rustock rootkit and spam bot. In *First workshop on hot topics in understanding botnets*, 2007.

[5] N. Daswani, M. Stoppelman, the Google Click Quality, and S. Teams. The anatomy of Clickbot.A. In *First workshop on hot topics in understanding botnets*, 2007.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[7] R. Durrett. *Random graph dynamics*. Cambridge University Press, 2006.

[8] G. Gu, J. Zhang, and W. Lee. Botsniffer: Detecting botnet command and control channels in network traffic. In *NDSS*, 2008.

[9] T. Holz, M. Steiner, F. Dahl, E. W. Biersack, and F. Freiling. Measurements and mitigation of peer-to-peer-based botnets: A case study on storm worm. In *LEET*, 2008.

[10] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys*, 2007.

[11] C. Kanich, C. Kreibich, K. Levchenko, B. Enright, G. Voelker, V. Paxson, and S. Savage. Spamalytics: An empirical analysis of spam marketing conversion. In *ACM CCS*, 2008.

[12] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen. Sketch-based change detection: Methods, evaluation, and applications. In *ACM SIGCOMM IMC*, 2003.

[13] Moving average. http://en.wikipedia.org/wiki/Moving_average.

[14] Message passing interface. http://www-unix.mcs.anl.gov/mpi/.

[15] Massive parallel processing. http://en.wikipedia.org/wiki/Massive_parallelism.

[16] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *ACM SIGMOD*, 2008.

[17] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems (2nd edition)*. Prentice-Hall, 1999.

[18] R. Pike, S. Dorward, R. Griesemer, and S. Quinla. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 13(4), 2005.

[19] P. Porras, H. Saidi, and V. Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. Technical report, SRI Computer Science Laboratory, 2007.

[20] H. Project and R. Alliance. Know your enemy: Tracking botnets. http://www.honeynet.org/papers/bots/, 2005.

[21] Parallel virtual machine. http://www.csm.ornl.gov/pvm/.

[22] M. A. Rajab, J. Zarfoss, F. Monrose, and A. Terzis. A multifaceted approach to understanding the botnet phenomenon. In *IMC*, 2006.

[23] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *ACM SIGCOMM*, 2006.

[24] A. Ramachandran, N. Feamster, and S. Vempala. Filtering spam with behavioral blacklisting. In *ACM CCS*, 2007.

[25] http://news.bitdefender.com/NW544-en-\--Trojan-Now-Uses-Hotmail-Gmail-as\-Spam-Hosts.html.

[26] Y. Xie, F. Yu, K. Achan, E. Gillum, M. Goldszmidt, and T. Wobber. How dynamic are ip addresses? In *ACM SIGCOMM*, 2007.

[27] Y. Xie, F. Yu, K. Achan, R. Panigrahy, G. Hulten, and I. Osipkov. Spamming botnets: Signatures and characteristics. In *SIGCOMM*, 2008.

[28] H.-C. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-Reduce-Merge: Simplified relational data processing on large clusters. In *ACM SIGMOD*, 2007.

[29] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

Unraveling the Complexity of Network Management

Theophilus Benson, Aditya Akella
University of Wisconsin, Madison

David Maltz
Microsoft Research

Abstract

Operator interviews and anecdotal evidence suggest that an operator's ability to manage a network decreases as the network becomes more complex. However, there is currently no way to systematically quantify how complex a network's design is nor how complexity may impact network management activities. In this paper, we develop a suite of *complexity models* that describe the routing design and configuration of a network in a succinct fashion, abstracting away details of the underlying configuration languages. Our models, and the *complexity metrics* arising from them, capture the difficulty of configuring control and data plane behaviors on routers. They also measure the inherent complexity of the reachability constraints that a network implements via its routing design. Our models simplify network design and management by facilitating comparison between alternative designs for a network. We tested our models on seven networks, including four university networks and three enterprise networks. We validated the results through interviews with the operators of five of the networks, and we show that the metrics are predictive of the issues operators face when reconfiguring their networks.

1 Introduction

Experience has shown that the high complexity underlying the design and configuration of enterprise networks generally leads to significant manual intervention when managing networks. While hard data implicating complexity in network outages is hard to come by, both anecdotal evidence and operator interviews suggest that more complex networks are more prone to failures, and are difficult to upgrade and manage.

Today, there is no way to systematically quantify how complex an enterprise configuration is, and to what extent complexity impacts key management tasks. Our experiments show that simple measures of complexity, such as the number of lines in the configuration files, are *not* accurate and do not predict the number of steps management tasks require.

In this paper, we develop a family of *complexity models and metrics* that *do* describe the complexity of the design and configuration of an enterprise network in a succinct fashion, abstracting away all the details of the underlying configuration language. We designed the models and metrics to have the following characteristics: (1) They align with the complexity of the mental model operators use when reasoning about their network—networks with higher complexity scores are harder for operators to manage, change or reason about correctly. (2) They can be derived automatically from the configuration files that define a network's design. This means that automatic configuration tools can use the metrics to choose between alternative designs when, as frequently is the case, there are several ways of implementing any given policy.

The models we present in this paper are targeted toward the Layer-3 design and configuration of enterprise networks. As past work has shown [19], enterprises employ diverse and intricate routing designs. Routing design is central both to enabling network-wide reachability and to limiting the extent of connectivity between some parts of a network.

We focus on modeling three key aspects of routing design complexity: (1) the complexity behind configuring network routers accurately, (2) the complexity arising from identifying and defining distinct roles for routers in implementing a network's policy, and (3) the inherent complexity of the policies themselves.

Referential Complexity. To model the complexity of configuring routers correctly, we develop the *referential dependence* graph. This models dependencies in the definitions of routing configuration components, some of which may span multiple devices. We analyze the graph to measure the average number of reference links per router, as well as the number of atomic units of routing policy in a network and the references needed to configure each unit. We argue that the number of steps operators take when modifying configuration increases monotonically with these measures.

Router Roles. We identify the implicit roles played by routers in implementing a network's policies. We argue that networks become more complex to manage, and updating configurations becomes more challenging, as the number of different roles increases or as routers simultaneously play multiple roles in the network. Our algorithms automatically identify roles by finding routers that share similar configurations.

Inherent Complexity. We quantify the impact of the reachability and access control policies on the network's complexity. Networks that attempt to implement sophisticated reachability policies, enabling access between some sets of hosts while denying it between others, are more complex to engineer and manage than networks with more uniform reachability policies. However, a network's policies cannot be directly read from the network's configuration and are rarely available in any other machine-readable form. Our paper explains how the complexity of the policies can be automatically extracted by extending the concept of *reachability sets* first introduced by Xie *et al.* [27]. Reachability sets identify the set of packets that a collection of network paths will allow based on the packet filters, access control rules and routing/forwarding configuration in routers on path. We compute a measure of the inherent complexity of the reachability policies by computing differences or variability between reachability sets along different paths in the network. We develop algorithms based on firewall rule-set optimization to compare reachability sets and to efficiently perform set operations on them (such as intersection, union and cardinality).

We validated our metrics through interviews with the operators and designers of six, four universities and two commercial enterprises. The questionnaires used in these interviews can be found online [4]. We also measured one other network where we did not have access to operators. Through this *empirical study of the complexity of network designs* we found we are able to categorize networks in terms of their complexity using the metrics that we define. We also find that the metrics are predictive of issues the operators face in running their networks. The metrics gave us insights on the structure and function of the networks that the operators corroborated. A surprising result of the study was uncovering the reasons why operator chose the designs they did.

Given the frequency with which configuration errors are responsible for major outages [22], we argue that creating techniques to quantify systematically the complexity of a network's design is an important first step to *reducing* that complexity. Developing such metrics is difficult, as they must be automatically computable yet still enable a direct comparison between networks that may be very different in terms of their size and routing design. In databases [14], software engineering [21], and other fields, metrics and benchmarks have driven the direction of the field by defining what is desirable and what is to be avoided. In proposing these metrics, we hope to start a similar conversation, and we have verified with operators through both qualitative and quantitative measures that these metrics capture some of the trickiest parts of network configuration.

2 Application to Network Management

Beyond aiding in an empirical understanding of network complexity, we believe that our metrics can augment and improve key management tasks. We illustrate a few examples that are motivated by our observations.

Understanding network structure: It is common for external technical support staff to be brought in when a network is experiencing problems or being upgraded. These staff must first learn the structure and function of the network before they can begin their work, a daunting task given the size of many networks and the lack of accurate documentation. As we show in Section 7, our techniques for measuring reachability have the side-effect of identifying routers which play the same role in a network's design. This creates a summary of the network, since understanding each role is sufficient to understand the purpose of all the similar routers.

Identify inconsistencies: Inconsistency in a network generally indicates a bug. When most routers fit into a small number of roles, but one router is different from the others, it probably indicates a configuration or design error (especially as routers are often deployed in pairs for reasons of redundancy). As we show in Section 6.3, when our inherent complexity metric found the reachability set to one router to be very different from the set to other routers, it pointed out a routing design error.

What-if analysis: Since our metrics are computed from configuration files, and not from a running network, proposed changes to the configuration files can be analyzed before deployment. Should any of the metrics change substantially, it is an excellent indication that the proposed changes might have unintended consequences that should be examined before deployment.

Guiding and automating network design: Networks are constantly evolving as they merge, split, or grow. Today, these changes must be designed by humans using their best intuition and design taste. In future work, we intend to examine how our complexity metrics can be used to direct these design tasks towards simpler designs that still meet the objectives of the designer.

Type	# rtrs	# hosts	Interviewed?
Univ-1	12	29,000	Y
Univ-2	19	9,000	N
Univ-3	24	8,000	Y
Univ-4	36	26,000	Y
Enet-1	8	6,000	Y
Enet-2	83	N/A	N
Enet-3	19	5,000	Y

Table 1: Studied networks.

3 Methodology and Background

Our project began with a review of formal training materials for network engineers (e.g., [25, 24]) and interviews with the operators of several networks to understand the tools and processes they use to manage their networks. From these sources, we extracted the "best common practices" used to manage the networks. On the hypothesis that the use of these practices should be discernible in the configuration files of networks that use them, we developed models and techniques that tie these practices to patterns that can be automatically detected and measured in the configurations.

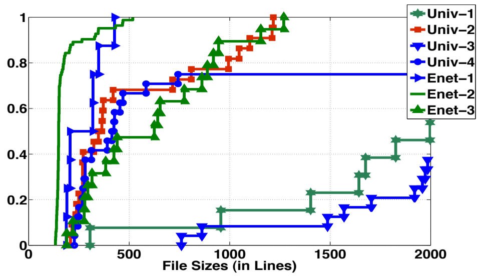
The remainder of this section describes the networks we studied, the best common practices we extracted, and a tutorial summary of network configuration in enterprise networks. The next sections precisely define our metrics, the means for computing them, and their validation.

3.1 Studied Networks

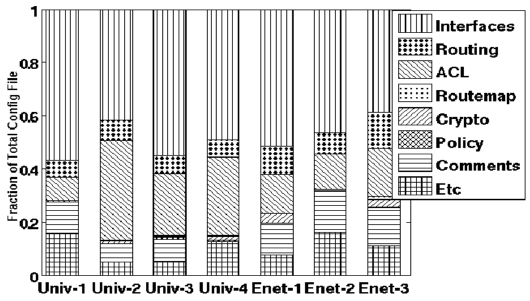
We studied a total of seven networks: four university networks and three enterprise networks, as these were the networks for which we could obtain configuration files. For four of the university networks and two enterprises, we were also able to interview the operators of the network to review some of the results of our analysis and validate our techniques. Table 1 shows the key properties of the networks.

Figure 1(a) plots the distribution of configuration file sizes for the networks. The networks cluster into three groups: Univ-2 and the enterprises consist of relatively small files, with 50% of their files being under 500 lines, while 90% of the files in Univ-1 and Univ-3 are over 1,000 lines and Univ-4 has a mix of small and large files. As we will see, configuration file size is not a good predictor of network complexity, as Univ-2 (small files) is among the most complicated networks and Univ-3 (large files) among the simplest.

Figure 1(b) breaks down the lines of configuration by type. The networks differ significantly in the fraction of their configurations devoted to Packet filters, widely known as ACLs, and routing stanzas. Univ-1 and the enterprises spend as many configuration lines on routing stanzas as on ACLs, while Univ-2, -3 and -4 define proportionately more ACLs than routing stanzas. Interface definitions, routing stanzas, and ACL definitions (the key



(a)



(b)

Figure 1: (a) Distribution of configuration file size across networks. (b) Fraction of configuration dedicated to configuring each aspect of router functionality.

building blocks for defining layer-3 reachability) account for over 60% of the configuration in all networks.

All the networks used some form of tools to maintain their configurations [16, 1]. Most tools are home-grown, although some commercial products are in use. Most had at least spreadsheets used to track inventory, such as IP addresses, VLANs, and interfaces. Some used template tools to generate portions of the configuration files by instantiating templates using information from the inventory database. In the sections that follow, we point out where tools helped (and sometimes hurt) operators.

3.2 Network Design and Configuration

Based on our discussions with operators and training materials, we extract the best common practices that operators follow to make it easier to manage their networks. Our complexity metrics quantify how well a network adheres to these strategies, or equivalently, to what extent a network deviates from them.

Uniformity. To the extent possible, operators attempt to make their networks as *homogeneous* as possible. Special cases not only require more thought and effort to construct in the first place, but often require special handling during all future network upgrades. To limit the number of special cases operators must cope with, they often define a number of archetypal configurations which they then reuse any time that special case arises. We call these archetypes *roles*.

Tiered Structure. Operators often organize their network devices into tiers to control the complexity of their

```
1    Interface Vlan901
2      ip 10.2.1.23  255.255.255.252
3      ip access-group 9 out
4    !
5    Router ospf 1
6      router-id 10.1.2.133
7      passive-interface default
8      no passive-interface Vlan901
9      network 10.2.0.0 0.0.255.255
10     distribute-list in 11
11     redistribute connected subnets
12   !
13   access-list 9 permit  10.2.1.23  0.0.0.3 any
14   access-list 9 deny any
15   access-list 11 permit 10.2.0.0 0.0.255.255
```

Figure 2: A sample configuration file.

design. For example, defining some routers to be border routers that connect with other networks, some routers to be core routers that are densely connected, and the remaining routers as edge routers that connect hosts.

Short Dependency Chains. Routers cannot be configured in isolation, as frequently one part of the configuration will not behave correctly unless other parts of the configuration, sometimes on other routers, are consistent with it. We define this to be a dependency between those configuration lines. Operators attempt to minimize the number of dependencies in their networks. This is because making a change to one configuration file but not updating all the other dependent configurations will introduce a bug. Since the configurations do not explicitly declare all their dependencies, operators' best strategy is to minimize the number of dependencies.

3.3 Overview of a Configuration File

All our complexity metrics are computed on the basis of router configuration files. Before defining the metrics, we describe the layout of the configuration file for a network router and provide an overview of the mechanisms (e.g., routing, ACLs and VLANs) used when designing enterprise networks.

The configuration file for a Cisco device consists of several types of *stanzas* (devices from other vendors have similar stanza-oriented configurations). A stanza is defined as the largest continguous block of commands that encapsulate a piece of the router's functionality.

In Figure 2, we show a simple configuration file consisting of the three most relevant classes of stanzas: interface in lines 1-3, routing protocol in lines 5-11, and ACL in lines 13-15. The behavior exhibited by a router can be explained by the interactions between various instances of the identified stanzas.

Egress filtering, i.e., preventing local hosts from sending traffic with IP addresses that does not belong to them, has become a popular way to combat IP-address hijacking. Networks implement egress filtering by defining a packet filter for each interface and creating a reference to the appropriate ACL from the interfaces. For example, line 3 exemplifies the commands an operator would use to setup the appropriate references.

The purpose of most layer-3 devices is to provide network-wide reachability by leveraging layer-3 routing protocols. Network-wide reachability can be implemented by adding a routing stanza and making references between that stanza and the appropriate interfaces. Lines 5-11 declare a simple routing stanza with line 8 making a reference between this routing protocol and the interface defined earlier. Even in this simple case, the peer routing protocol stanza on neighboring devices must be configured consistent with this stanza before routes can propagate between the devices and through the network.

More complex reachability constraints can be imposed by controlling route distribution using ACLs. Line 15 is a filter used to control the announcements received from the peer routing process on neighboring routers.

VLANs are widely used to provide fine grain control of connectivity, but they can complicate configuration by providing an alternate means for packets to travel between hosts that is independent of the layer-3 configuration. In a typical usage scenario, each port on a switch is configured as layer-2 or layer-3. For each layer-3 port there is an interface stanza describing its properties. Each layer-2 port is associated with a VLAN V. The switches use trunking and spanning tree protocols to ensure that packets received on a layer-2 port belonging to VLAN V can be received by every host connected to a port on VLAN V on any switch.

Layer-2 VLANs interact with layer-3 mechanisms via virtual layer-3 interfaces — an interface stanza not associated with any physical port but bound to a specific VLAN (lines 1–3 in Figure 2). Packets "sent" out the virtual interface are sent out the physical ports belonging to the VLAN and packets received by the virtual interface are handled using the layer-3 routing configuration.

4 Reference Chains

As the above description indicates, enabling the intended level of reachability between different parts of a network requires establishing reference links in the configuration files of devices. Reference links can be of two types: those between stanzas in a configuration file (*intra-file* references) and those across stanzas in different configuration files (*inter-file*). Intra-file references are explicitly stated in the file, e.g. the links in line 8 (Figure 2) from a routing stanza to an interface, and in line 10 from a routing stanza to an ACL — these must be internally consistent to ensure router-local policies (e.g. ingress filters and locally attached networks) are correctly implemented. Inter-file references are created when multiple routers refer to the same network object (e.g., a VLAN or subnet); these are central to configuring many network-wide functions, and crucially, routing and reachability.

Unlike their intra-file counterparts, not all inter-file references can be explicitly declared. For example, line 2 refers to a subnet which is an example of an entity that cannot be explicitly declared.

As our interviews with operators indicate (§4.3), in some networks the reference links must be manually established. In other networks, some of the reference links within a device are set using automated tools, but many of the inter-file references such as trunking a VLAN on multiple routers and setting routing protocol adjacencies must be managed manually.

To quantify the complexity of reference links, we first construct a *referential dependency graph* based on device configuration files. We compute a set of *first-order complexity metrics* which quantify the worst case complexity of configuring reference links in the network. Because reference links often play a role in implementing some network-wide functionality, we also define *second order metrics* that estimate the overall complexity of configuring such functionality. We focus on routing in this discussion, as operators report it is a significant concern.

4.1 Referential Dependence Graph

We use a two-step approach to parse configuration files and create a dependency graph.

1. Symbol Table Creation. Router vendor documentation typically lists the commands that can appear within each configuration stanza and the syntax for the commands. Based on this, we first create a grammar for configuration lines in router configuration files. We build a simple parser that, using the grammar, identifies "tokens" in the configuration file. It records these tokens in a symbol table along with the stanza in which they were found and whether the stanza defined the token or referred to it. For example, the access-list definitions in lines 13-14 of Figure 2 define the token ACL 9 and line 3 adds a reference to ACL 9.

2. Creating Links. In the linking stage, we create reference edges between stanzas within a single file or across files based on the entries in the symbol table. We create unidirectional links from the stanzas referencing the tokens to the stanza declaring the tokens. Because every stanza mentioning a subnet or VLAN is both declaring the existence of the subnet or VLAN and referencing the subnet/VLAN, we create a separate node in the reference graph to represent each subnet/VLAN and create bidirectional links to it from stanzas that mention it.

We also derive maximal sub-graphs relating to Layer-3 control plane functionality, called "routing instances" [19]. A routing instance is the collection of routing processes of the same type on different devices in a network (e.g. OSPF processes) that are in the transitive closure of the "adjacent-to" relationship. We derive these adjacencies by tracing relationships between routing processes across subnets that are referenced in common by neighboring routers. Taken together, the routing instances implement control plane functionality in a network. In many cases, enterprise networks use multiple routing instances to achieve better control over route distribution, and to achieve other administrative goals [19]. For example, some enterprises will place routes to different departments into different instances — allowing designers to control reachability by controlling the instances in which a router participates. Thus, it is important to understand the complexity of configuring reference links that create routing instances.

4.2 Complexity Metrics

We start by capturing the baseline difficulty of creating and tracking reference links in the entire network. The first metric we propose is the *average configuration complexity*, defined as the total number of reference links in the dependency graph divided by the number of routers. This provides a holistic view of the network.

We also develop three second-order metrics of the complexity of configuring the Layer-3 control plane of a network. First, we identify the number of interacting routing policy units within the network that the operator must track globally. To do this, we count the number of distinct routing instances in the entire network. Second, we capture the average difficulty of correctly setting each routing instance by calculating the average number of reference links per instance. Finally, we count the number of routing instances each router participates in. In all three cases, it follows from the definition of the metrics that higher numbers imply greater complexity for a network.

4.3 Insights From Operator Interviews

We derived referential complexity metrics for all seven networks. Our observations are summarized in Table 2. Interestingly, we note that the referential metrics are different across networks – e.g. very low in the cases of Enet-1 and much higher for Univ-1. For five of the seven networks, we discussed our findings regarding referential dependencies with network operators.

We present the insights we derived focusing on 3 key issues: (1) *validation*: are the referential dependencies we inferred correct and relevant in practice (meaning that these are links that must be created and maintained for consistency and/or correctness)? (2) *complexity*: are our complexity metrics indicative of the amount of difficulty operators face in configuring their networks? (3) *causes*: what caused the high referential complexity in the networks (where applicable)?

Network	Avg ref complexity per router	Layer-3 functionality			Int?
		Num routing instances	Complexity per instance	Instances per router	
Univ-1	41.75	14	35.8	2.5	Y
Univ-2	8.3	3	58.3	1.1	N
Univ-3	4.1	1	99	1	Y
Univ-4	75	2	902	1	Y
Enet-1	1.6	1	16	0.7	Y
Enet-2	7.5	10	62	1.2	N
Enet-3	22	8	52	1.4	Y

Table 2: Complexity due to referential dependence. Networks where we validated results are marked with a "Y."

Validation. We showed each network's referential dependence graph to the network operators, along with subgraphs corresponding to the routing protocol configuration in their network. All operators confirmed that the classes of links we derived (e.g. between stanzas of specific kinds, link within stanzas and across routers) were relevant. We also gave operators tasks involving changes to device configurations (specifically, add or remove a specific subnet, apply a new filter to a collection of interfaces). We verified that our reference links tracked the action they took. These two tests, while largely subjective, validated our referential dependency derivation.

As an aside, the dependency graph seems to have significant practical value: Univ-1 and Enet-1 operators felt the graph was useful to visualize their networks' structure and identify anomalous configurations.

Do the metrics reflect complexity? Our second goal was to test if the metrics tracked the difficulty of maintaining referential links in the network. To evaluate this, we gave the operators a baseline task: add a new subnet at a randomly chosen router. We measured the number of steps required and the number of changes made to routing configuration. This is summarized below.

Network	Num steps	Num changes to routing
Univ-1	4-5	1-2
Univ-3	4	0
Enet-1	1	0

In networks where the metrics are high (Table 2), operators needed more steps to set up reference links and to modify more routing stanzas. Thus, the metrics appear to capture the difficulty faced by operator in ensuring consistent device-level and routing-level configuration. We elaborate on these findings below.

In Univ-1, the operators used a home-grown automated tool that generates configuration templates for adding a new subnet. Thus, although there are many references to set, automation does help mitigate some aspects of this complexity.

Adding the subnet required Univ-1's operator to modify routing instances in his network. Just as our second order complexity metrics predicted, this took multiple steps of manual effort. The operator's automation tool actually made it *harder* to maintain references needed for Layer-3 protocols. Note from Table 2 that an average Univ-1 router has two routing instances present on it. These are: a "global" OSPF instance present on all core routers and a smaller per-router RIP instance. The RIP instance runs between a router and switches directly attached to the router, and is used to distribute subnets attached to the switches into OSPF. On the other hand, OSPF is used to enable global reachability between subnets and redistribute subnets that are directly attached to the router. When a new subnet is added to Univ-1, the operator's tool automatically generates a `network` command and incorporates it directly into the OSPF instance. When the subnet needs to be attached to a Layer-2 switch, however, the `network` statement needs to be incorporated into RIP (and not OSPF). Thus, the operator must manually undo the change to OSPF and update the RIP instance. Unlike the OSPF instance, the `network` statements in RIP require parameters that are specialized to a switch's location in the network.

Univ-3 presents a contrast to Univ-1. The operator in Univ-3 required 4 steps to add the subnet and this is clearly shown by the first order complexity metric for Univ-3. In contrast to Univ-1, however, almost all of the steps were manual. In another stark difference from Univ-1, the operator had no changes to make to the routing configuration. This is because the network used exactly one routing instance that was setup to redistribute the entire IP space. This simplicity is reflected in the very low second order metrics for Univ-3.

The operator in Enet-1 had the simplest job overall. He had to perform 1 simple step: create an interface stanza (this was done manually). Again, the routing configuration required little perturbation.

In general, we found that the metrics are not directly proportional to the number of steps required to complete a management task like adding a subnet, but *the number of steps required is monotonically increasing with referential complexity*. For example, Univ-1 with a reference metric of 41.75 required 4-5 steps to add a subnet. Univ-2, with a metric of 4.1 needed 4 steps and Enet-1 with a metric of 1.6 needed just one step.

Causes for high complexity. The most interesting part of our interviews was understanding what caused the high referential complexity in some networks. The reasons varied across networks, but our study highlights some of the key underlying factors.

The first cause we established was the impact of a network's *evolution over time* on complexity. In Univ-1, approximately 70% of reference links arose due to "no passive interface" statements that attempt to create routing adjacencies between neighboring devices. Upon closer inspection, we found that a large number of these links were actually *dangling references*, with no corresponding statement defined at the neighboring router; hence, they played no role in the network's routing functionality. When questioned, the operator stated that the commands were used at one point in time. As the network evolved and devices were moved, however, the commands became irrelevant but were never cleaned up.

The high second order complexity in Univ-1 results from an interesting cause - *optimizing for monetary cost* rather than reducing complexity. Univ-1's operator could have used a much smaller number of routing instances (e.g. a single network-wide OSPF) with lower referential counts to achieve the goal of spreading reachability information throughout the network. However, according to the operator, using OSPF on a small number of routers, and RIP between switches and routers, was significantly cheaper as OSPF-licensed switches cost more. Hence this routing design was adopted although it was more complex.

Sometimes, *the policies being implemented may require high referential complexity*. For instance, Univ-3 imposes global checks for address spoofing, and therefore applies egress filters on all network interfaces. These ACLs accounted for approximately 90% of the links in the dependency graph. Similarly, Univ-4 uses ACLs extensively, resulting in high referential complexity. Despite similar underlying cause, Univ-4 has a higher complexity value than Univ-3 because it employs significantly more interfaces and devices.

5 Router Roles

When creating a network, operators typically start by defining a base set of behaviors that will be present across all routers and interfaces in the network. They then specialize the role of routers and interfaces as needed to achieve the objectives for that part of the network, for example, adding rate shaping to dorm subnets, and additional filters to protect administrative subnets.

Designers often implement these roles using *configuration templates* [6]. They create one template for each role, and the template specifies the configuration lines needed to make the router provide the desired role. Since the configuration might need to be varied for each of the routers, template systems typically allow the templates to contain parameters and fill in the parameters with appropriate values each time the template is used. For example, the template for an egress filter might be as shown in Figure 3, where the ACL restricts packets sent by interface III to those originating from the subnet configured to the interface. The designer creates specific configuration stanzas for a router by concatenating together the lines output by the template generator for each behavior the router is supposed to implement.

From a complexity stand-point, the more base behaviors defined within the network, the more work an operator will have to do to ensure that the behaviors are all defined and configured correctly and consistently. Further, the greater the degree of specialization required by routers to implement a template role, the more complex it becomes to configure the role.

```
interface III
   ip access-group 5 in
   ip address AAA SSS
access-list 5 permit AAA SSS
access-list 5 deny any
```

Figure 3: Example of a configuration template.

We show how to work backwards from configurations to retrieve the original base behaviors that created them. By doing so, we can measure two key aspects of the difficulty of configuring roles on different routers in a network: (1) how many distinct roles are defined in the network? (2) How many routers implement each role?

5.1 Copy-Paste Detection

We identify roles that are "shared" by multiple routers using a *copy-paste* detection technique. This technique looks for similar stanzas on different routers.

We build the copy-paste detection technique using CCFinder [17], a tool that has traditionally been used to identify cheating among students by looking for text or code that has been cut and paste between their assignments. We found that CCFinder by itself does not identify templates of the sort used in router configuration (e.g., Figure 3). To discover templates, we automatically preprocess every file with *generalization*. *Generalization* replaces the command arguments that may vary with wild card entries – for example, IP addresses are replaced by the string "IPADDRESS". Our implementation uses the grammar of the configuration language (Section 4) to identify what parameters to replace.

5.2 Complexity Metrics

Our first metric is the *number of base behaviors* defined within the network. We define a base behavior as a *maximal* collection of shared-template stanzas that appear together on a set of two or more routers. As the number of base behaviors increases, the basic complexity of configuring multiple roles across network routers increases.

To compute the number of base behaviors, we first identify the *shared-template device set* of each template — this is the set of devices on which the configuration template is present. Next, we coalesce identical sets. To elaborate, we write the device set for a shared-template stanza as $ST_i = \{D_1^i, D_2^i, \ldots, D_{k_i}^i\}$ where the D_j^i represents a router that contains a configuration stanza generated from shared template i. We scan the shared-template device sets to identify identical sets: If two

N/w	# Rtrs	Shared template behaviors			Int?
		#	Device set size		
			Median	Mean	
Univ-1	12	7	2	4.43	Y
Univ-2	19	19	2	5.75	N
Univ-3	24	10	2	8.3	Y
Univ-4	24	28	3.5	4.3	Y
Enet-1	10	1	2	2	Y
Enet-2	83	5	3	34.2	N
Enet-3	19	6	7.5	8.8	Y

Table 3: Roles extracted from ACLs.

shared-template stanzas are present on the same set of routers, then the stanzas can be considered to have arisen from a single, larger template; the stanzas are merged and one of the sets discarded. The final number of distinct device sets that remain is the number of base behaviors.

As a second order metric, we quantify the *uniformity* among devices in terms of the behaviors defined on them. If all devices in the network exhibit the same set of behaviors (i.e., they all have the same shared-template), then once an operator understands how one router behaves, it will be easier for him to understand how the rest of the routers function. Also, updating the roles is simple, as all routers will need the same update.

To measure uniformity, we compute the median and mean numbers of devices in the device sets. We evaluated other information-theoretic metrics such as entropy. However, as our empirical study will show, these simple metrics, together with the number of base behaviors, suffice to characterize the behaviors defined in a network.

5.3 Insights from Operator Interviews

Like the referential metrics, we validated our role metrics through interviews with five operators. For this discussion, we focus on the use of ACLs, and Table 3 shows the role metrics for each network. We also evaluated roles across the entire configuration file, and the results are consistent with those for ACLs.

Validation. When shown the shared templates extracted by our system, each of the operators immediately recognized them as general roles used in their networks and stated that no roles were missed by our technique. For example, Univ-1 operators reported seven roles for ACLs in their network: one role for a SNMP-related ACL, one role for an ACL that limits redistribution of routes between OSPF and RIP (these first two roles are present on most routers) and five ACLs that filter any bogus routes that might be advertised by departmental networks connected to the university network, one for each of the five departments (these are found only on the routers where the relevant networks connect).

Enet-3 has separate templates for sub-networks that permit multicast and those that do not, as well as templates encoding special restrictions applied to several labs and project sub-networks. Enet-1, the network with the fewest shared-templates, has a pair of core routers that share the same set of ACLs. The remaining ACLs in the network are specific to the special projects subnets that are configured on other routers. Univ-4, the network with the most shared-templates, has so many roles as it uses multiple different types of egress filters, each of which is applied to subset of the routers. There are also several special case requests from various departments, each represented as an ACL applied to 2-3 routers.

Do the metrics reflect complexity? The relationship between number of roles and the complexity of the network is indicated by type of tools and work process used by the operators.

Operators of the network with the fewest roles, Enet-1, modify all the ACLs in their network manually — they are able to manage without tools due to the uniformity of their network. Operators at Univ-1 have tools to generate ACLs, but not track relationships between ACLs, so they push all ACLs to all routers (even those that do not use the ACL) in an effort to reduce the complexity of managing their network by increasing the consistency across the configuration files (our shared template system was programmed to ignore ACLs that are not used by the router: this explains why the mean device set size is not larger for Univ-1). The environment at Univ-3 is similar to Univ-1, with roughly the same number of ACL roles and similar tools that can create ACLs from templates, but not track relationships between them. The Univ-3 operators took the opposite approach to Univ-1, pushing each ACL only to the routers that use it, but using manual process steps to enforce a discipline that each ACL contain a comment line listing all the routers where an instance of that ACL is found. Operators then rely on this meta-data to help them find the other files that need to be updated when the ACL is changed.

Causes for high complexity. In general, the number of shared-templates we found in a network directly correlates with the complexity of the policies the operators are trying to realize. For example, for Univ-1's goal of filtering bogus route announcements from departmental networks requires applying a control plane filter at each peering point. Similarly, Univ-4 has policies defining many different classes of subnets that can be attached to the network, each one needing its own type of ACL (e.g., egress filtering with broadcast storm control and filtering that permits DHCP). There is no way around this type of complexity.

Interestingly, the number of roles found in a network appears to be largely independent of the size of the network. For example, Enet-2 and Enet-3 have the same number of roles even though they differ greatly in size. Rather, the number of roles seems to stem directly from the choices the operators made in designing their networks, and how uniform they chose to make them.

6 Inherent Complexity

A network's configuration files can be viewed as the "tools" used by network operators to realize a set of *network-wide reachability policies*. These policies determine whether a network's users can communicate with different resources in the network (e.g other users or services). The policies that apply to a user could depend on the user's "group," her location, and other attributes.

The reachability policies fundamentally bound an operator's ability to employ simple configurations network-wide. Consider a network with a "simple" reachability policy, such as an all-open network that allows any pairs of users to have unfettered communication, or at the opposite end of the spectrum, a network where all communication except those to a specific set of servers is shut off. Such policies can be realized using fairly simple network configurations. On the other hand, for networks where the reachability policies are complex, i.e., where subtle differences exist between the constraints that apply to different sets of users, implementing the policies will require complex configuration.

We develop a framework for quantifying the complexity of a network's reachability policies. We refer to this as the network's *inherent complexity*. We use feedback from operators to both validate our metrics and understand the factors behind the inherent complexity (where applicable). Ultimately, we wish to tie inherent complexity back to the configuration complexity and examine the relationship between the two. We discuss this in §6.3.

To derive inherent complexity, we first derive the static reachability between network devices, which is the set of packets that can be exchanged between the devices. We also refer to this as the *reachability set* for the device pair. Our inherent complexity metrics essentially quantify the level of uniformity (or the lack of it) in the reachability sets for various paths in a network.

6.1 Reachability Sets

For simplicity, we assume that network routers have IP subnets attached to them, and that each IP address in a subnet corresponds to a single host. The reachability set for two routers A and C in a network, denoted by $\mathcal{R}(A, C)$, is the set of all IP packets that can originate from hosts attached to A (if any), traverse the $A \rightarrow C$ path, and be delivered to hosts attached at C (if any).

The composition of the reachability sets reflects how a network's policy limits the hosts at a certain network location from being reachable from hosts at another network location. At Layer-3, these policies generally apply to 5 fields in the packet's IP header – the source/destination addresses, ports and protocol. When first sent, the source and destination addresses on the packets could take any of the possible 2^{32} values (the same with ports and the protocol field). Control and data plane mechanisms on the path might then drop some of the packets, either because a router on the path lacks a forwarding entry to that destination or due to packet filters. $\mathcal{R}(A, C)$ identifies the packets that are eventually delivered to hosts attached to C. Note that the maximum size of $\mathcal{R}(A, C)$ is $2^{32} \times |C| \times 2^{16} \times 2^{16} \times 2^{8}$, where $|C|$ is the total number of hosts attached to C.

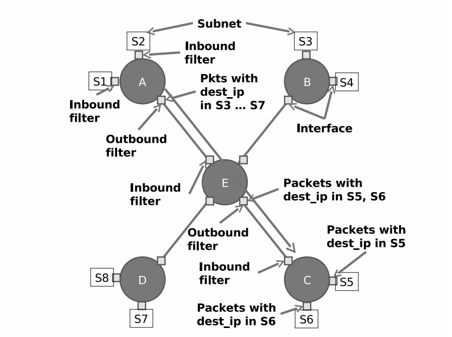


Figure 4: A toy network with 8 subnets and 5 routers. The different constituent sets that play a role in the reachability set for the A→C path are shown.

6.1.1 Reachability Set Computation

To compute the reachability sets for a network we consider three separate yet interacting mechanisms: control-plane mechanisms (i.e., routing protocols), data-plane mechanisms (i.e. packet filters), and Layer-2 mechanisms (such as VLANs).

We compute the reachability sets using the following three steps: (1) we first compute valid forwarding paths between network devices by simulating routing protocols (In the interest of space, we omit the details of routing simulation; the details are in [5]); (2) we calculate the "per-interface" reachability set on each path – this is the set of all packets that can enter or leave an interface based both on forwarding entries as well as packet filters; and (3) we compute reachability sets for end-to-end paths by intersecting the reachability sets for interfaces along each path. The last two steps are illustrated for a simple toy network in Figure 4, and explained in detail below.

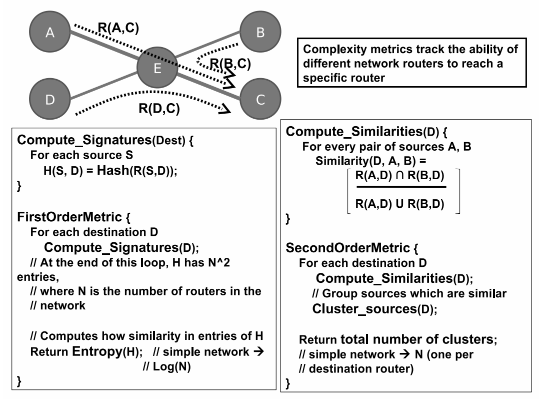
We note that our reachability calculation is similar to Xie *et al.*'s approach for static reachability analysis of IP networks [27]. However, our approach differs both in the eventual goal and the greater flexibility it provides. Xie *et al.* derive all possible forwarding states for a network to study the impact of failures, rerouting, etc. on reachability. Because we are interested in examining the inherent complexity of reachability policies, we focus on the computationally simpler problem of computing a single valid forwarding state for the network, assuming there are no failures. Also, our approach takes into account the impact of VLANs on reachability within a network (as described in [5]), which Xie *et al.* does not. The presence of VLANs means that routing is effectively a two step process: first routing to the VLAN interface, and then routing through the VLAN to the destination. Our calculation tracks which routers trunk which VLANs to enable this second step of the routing computation.

Single interface. The reachability set for interfaces on a path is defined as the set of packets that can enter or leave an on-path interface (see figure 4 for examples). For interfaces that receive packets, this is composed just of the set of packets allowed by inbound data plane filters. For interfaces which forward packets further along a path, this is the union of packets which are permitted by outbound filters and packets whose destination IPs are reachable from the interface (this depends on the router's forwarding state).

Path. To compute $\mathcal{R}(A, C)$, we first compute the following supersets: (1) For A, we compute the $Entry$ set which is the union of the inbound interface sets for interfaces on A — as mentioned above, each set is shaped by the inbound filters on the corresponding interface. (2) For C, we compute the $Exit$ set which is union of the outbound interface sets for interfaces on C. (3) For intermediate routers, we compute the intersection of the inbound interface set for the interface that receives packets from A and the outbound interface set for the interface that forwards to C. Then, $\mathcal{R}(A, C)$ is simply the intersection of $Entry$, $Exit$ and the intermediate sets.

Some optimizations for efficiency. The above computation requires us to perform set operations on the interface and intermediate reachability sets (i.e. set unions and intersections). These operations could be very time-consuming (and potentially intractable) because we are dealing with 5-dimensional reachability sets that could have arbitrary overlap with each other.

To perform these operations efficiently, we convert each set into a "normalized" form based on *ACL optimization*. Specifically, we represent each reachability set as a linear series of rules like those used to define an ACL in a router's configuration, i.e., a sequence of permit and deny rules that specify attributes of a packet and whether packets having those attributes should be allowed or forbidden, where the first matching rule determines the outcome. Next, we optimize this ACL representation of the sets using techniques that have traditionally been employed in firewall rule-set optimization [2, 11]. In the final ACL representation of a reachability set, no two rules that make up a set overlap with each other, and we are guaranteed to be using the minimal number of such rules possible to represent the set. Set operations are easy to perform over the normalized ACL representations. For instance, to compute the union of two reachability sets we merge the rules in the corresponding optimized ACLs to create one ACL, and then we optimize the resulting ACL. Intersection can be computed in a similar fashion.


A
B
E
D
C
R(A,C)
R(B,C)
R(D,C)
Complexity metrics track the ability of different network routers to reach a specific router

```
Compute_Signatures(Dest) {
   For each source S
      H(S, D) = Hash(R(S,D));
}

FirstOrderMetric {
   For each destination D
      Compute_Signatures(D);
   // At the end of this loop, H has N^2
entries,
   // where N is the number of routers in the
   // network

   // Computes how similarity in entries of H
   Return Entropy(H);   // simple network →
                                   // Log(N)
}
```

```
Compute_Similarities(D) {
   For every pair of sources A, B
      Similarity(D, A, B) =
         $\left[ \frac{R(A,D) \cap R(B,D)}{R(A,D) \cup R(B,D)} \right]$
}

SecondOrderMetric {
   For each destination D
      Compute_Similarities(D);
      // Group sources which are similar
      Cluster_sources(D);

   Return total number of clusters;
   // simple network → N (one per
   // destination router)
}
```

Figure 5: Computing the first and second order metrics for inherent complexity.

6.2 Complexity Metrics

As stated before, our metrics for inherent complexity quantify the similarity, or equivalently, the uniformity, in the reachability sets for various end-to-end paths. If the sets are uniformly restrictive (reflecting a "default deny" network) or uniformly permissive (an all open network), then we consider the network to be inherently simple. We consider dissimilarities in the reachability sets to be indicative of greater inherent complexity.

First order metric: variations in reachability. To measure how uniform reachability is across a network, we first compute the reachability set between all pairs of routers. We then compute the entropy of the resulting distribution of reachability sets and use this value, the *reachability entropy,* as a measure of uniformity.

Figure 5 summarizes the computation of reachability entropy. To compute the distribution of reachability sets over which we will compute the entropy, we must count how many pairs of routers have the same reachability. Intuitively, if there are N routers this involves comparing N^2 reachability sets for equality. To simplify this task, we compute the reachability set for a pair of routers, turn it into optimized ACL form, and then compute a hash of the text that represents the optimized set. Identical reachability sets have identical hashes, so computing the distribution is easy.

Using the standard information-theoretic definition of entropy, the reachability entropy for a network with N routers varies from $log(N)$ in a very simple network (where the reachability sets between all pairs of routers are identical) and $log(N^2)$ in a network where the reachability set between each pair of routers is different. We interpret larger values of entropy as indicating the network's policies are inherently complex.

Second order metric: Extent of variations. The entropy simply tracks whether the reachability sets for network paths differ, but it does not tell us the extent of the differences. If the reachability sets had even minute differences (not necessarily an indication of great complexity), the entropy could be very high. Thus, entropy alone may over-estimate the network's inherent complexity.

To quantify more precisely the variability between the reachability sets, we examine the similarity between sets using the approach outlined in Figure 5. Unlike the entropy calculation, where we examined the N^2 reachability sets between pairs of routers, we examine differences from the view point of a single destination router (say C). For each pair of source routers, say A and B, we compute the similarity metric, $Sim(C, A, B) = \frac{|R(A,C) \cap R(B,C)|}{|R(A,C) \cup R(B,C)|}$.

We use the set union and intersection algorithms described in Section 6.1.1 to compute the two terms in the above fraction. To compute the sizes of the two sets, we first optimize the corresponding ACLs. In an optimized ACL the rules are non-overlapping, so the number of packets permitted by an ACL is the sum of the number of packets allowed by the ACL's permit rules. Since each rule defines a hypercube in packet space, the number of packets permitted by a rule is found by multiplying out the number of values the rule allows on each dimension (e.g., address, port).

After computing the similarities in this manner, we cluster source routers that have very similar reachability sets (we use an *inconsistency cutoff* of 0.9) [20, p. 1-61]. Finally, we sum the number of clusters found over all destination routers to compute the *number of per-destination clusters* as our second order metric for inherent complexity. Ideally, this should be N; large values indicate specialization and imply greater complexity.

6.3 Insights from Operator Interviews

Our study of the configuration complexity in Sections 4 and 5 showed that some of the networks we studied had complex configurations. In this section, we examine the inherent complexity of these networks. We validate our observations using operator feedback. We also use the feedback to understand what caused the complexity. (Were the policies truly complex? Was there a bug?)

Our observations regarding the inherent complexity for the networks we studied are shown in Table 4. Interestingly, we see that a majority of the networks actually had reasonably uniform reachability policies (i.e. observed entropy $\approx$ ideal entropy of $log(N)$). In other words, *most networks seem to apply inherently simple policies at Layer-3 and below*.

To validate this observation, we verify with the operators if the networks were special cases that our approach somehow missed. We discussed our observations with the operators of 4 of the 7 networks. The operator for Enet-1 essentially confirmed that the network did not impose any constraints at Layer-3 or below and simply provided universal reachability. All constraints were imposed by higher-layer mechanisms using middleboxes such as firewalls.

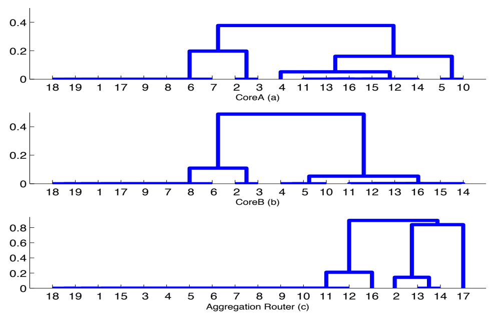


Figure 6: This figure shows the clusters of routers in Univ-2 that have similar reachability to the given destination router. The X axis is the source router ID. The Y axis is distance between the centers of the clusters.

We turn our attention next to the networks where the reachability entropy was slightly higher than ideal (Univ-1 and Univ-3). This could arise due to two reasons: either the network's policies make minor distinctions between some groups of users creating a handful of special cases (this would mean that the the policy is actually quite simple), or there is an anomaly that the operator has missed.

In the case of Univ-3, our interaction with the operator pointed to the former reason. A single core router was the cause of the deviation in the entropy values. During discussions with the operator, we found out that the router was home to two unique subnets with restricted access.

Interestingly, in the case of Univ-1 the slight change in entropy was introduced by a *configuration bug*. Upon discussing with the operator, we found that one of the routers was not redistributing one of its connected subnets because a `network` statement was missing from a routing stanza on the device. The bug has now been fixed. This exercise shows how our first and second order inherent complexity metrics can *detect inconsistencies between an operator's intent and the implementation* within a network. In networks where the configuration is complex – Univ-1 is an example with high referential counts and many router roles – such inconsistencies are very hard to detect. However, our complexity metrics were able to unearth this subtle inconsistency. We finally discuss networks where the entropy is much higher than ideal. Of these networks, we were able to speak to the operator of Univ-2, where both the first and the second order metrics are very high. In such networks, one can safely conclude that the policies themselves are complex. Indeed, Figure 6 examines how similar or different is the reachabilty from each of the routers in Univ-2 to three key routers: CoreA (Figure 6(a)), CoreB (Figure 6(b)),

Network	Entropy (ideal)	Num Clusters (Num routers)	Int?
Univ-1	3.61(3.58)	12(12)	Y
Univ-2	6.14(4.52)	36(19)	Y
Univ-3	4.63(4.58)	26(24)	Y
Univ-4	5.70(4.58)	85(24)	N
Enet-1	2.8(2.8)	8(8)	Y
Enet-2	6.69(6.47)	92(83)	N
Enet-3	5.34(4.25)	40(19)	N

Table 4: Inherent complexity measures.

Aggregation (Figure 6(c)). For each router C, the reachability set from every other router to that router is computed, and the distance between the reachability sets from routers A and B is computed as $1 - Sim(C, A, B)$. A distance of 0 means the sets are identical, and a distance 1 means the sets do not overlap. The dendrogram shows a horizontal line between clusters of routers at the distance between the centroids of the clusters.

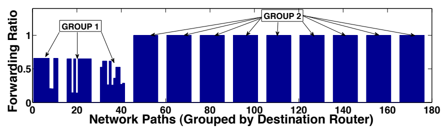
Interpreting Figure 6, there are 3-5 clusters of routers that have essentially the same reachability to both coreA and coreB (the only significant difference is that 4, 5, 10 have identical reachability to coreB, while 4 has slightly different reachability to coreA than 5 and 10 do). The presence of multiple clusters implies that traffic is being controlled by fine grain policies. That the clusters of reachability to the Aggregation Router are so different than those to the core implies that not only are policies fine grain, they differ in different places in the network. We argue this means the policies are inherently complex, and that any network implementing them will have a degree of unavoidable complexity. The operator for Univ-2 agreed with our conclusions.

Applying this analysis to all the networks we studied, Table 4 shows the number of per-destination clusters, that is, the total number of clusters found summing across all the routers in the network (second order metric). This complexity metric confirms that Univ-1 and Enet-1 have inherently simple reachability policies.

However, this metric's value stems from the information it provides about networks like Univ-2, -4 and Enet-3. Enet-3 and Univ-4 both have an entropy value roughly 1.0 higher than ideal. However, Univ-4 has on average four different clusters of reachability for each router (85/24), while Enet-3 has two clusters per router (40/19). This indicates that Enet-3 has reachability sets that are not identical, but are so similar to each other they cluster together, while Univ-4 truly has wide disparity in the reachability between routers. Similarly, Univ-2 has an entropy metric 1.6 above ideal yet less than two different clusters per router, indicating that even when reachability sets are not identical, they are very similar.

Summary of our study. Through interviews with the operators we have verified the correctness of our techniques. We show that our metrics capture the difficulty of adding new functionality such as interfaces, of updating existing functionality such as ACLs, and of achieving

(a) Univ-1



(b) Univ-2

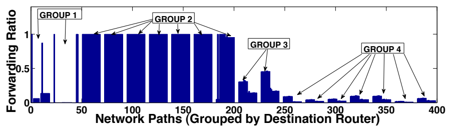


Figure 7: Sink profiles for Univs 1, 2. Network paths for each device are grouped by the destination router.

high-level policies such as restricting user access. In addition to this, we find that other factors, largely ignored by previous work (e.g. cost and design) play a larger role in affecting a network's complexity than expected.

7 An Application: Extracting Hierarchy

In addition to creating a framework for reasoning about the complexity of different network designs, complexity metrics have several practical uses including helping operators visualize and understand networks. In this section, we show how our models can discover a network's heirarchy, information that proves invaluable to operators making changes to the network.

Many networks are organized into a hierarchy, with tiers of routers leading from a core out towards the edges. The ability to automatically detect this tiering and classify routers to it would be helpful to outside technical experts that must quickly understand a network before they can render assistance.

We found that computing the *sink ratio* for each router rapidly identifies the tiering structure of a network. The sink ratio is based on the reachability analysis done on each path, and measures the fraction of packets that a router sinks (delivers locally) versus the number it forwards on. Formally, the sink ratio for a path $A \rightarrow B$ is $\frac{|\mathcal{R}_{Sink}(A,B)|}{|\mathcal{R}(A,B)|}$. If the ratio is 1, then B does not forward traffic from A any further. If not, then B plays a role in forwarding A's packets to the rest of the network.

Figure 7 shows the sink ratio for each path in networks Univ-1 and Univ-2. Univ-2 contains roughly 4 classes of devices: the edge (Group 2), the core (Group 1), intermediate-core (Group 4), and intermediate-edge(Group 3). Univ-1 consists of a two-layer architecture with three core routers and nine edge routers, respectively labeled Group 1 and Group 2. Enet-2 (not shown) has low forwarding ratios overall: the maximum forwarding ratio itself is just 0.4 and the minimum is 0.15. Thus, we can deduce that all routers in Enet-2 play roughly identical forwarding roles and there is no distinction of core versus edge routers.

8 Discussion

We now discuss procedural limitations in our approach to quantifying complexity as well as some notions of complexity that we are currently unable to capture.

Limitations and extensions. Our approach uses the static configuration state of the network. Relying on static configurations means that operators can use our techniques to do "what-if analysis" of configuration changes. The downside is that we ignore the effect of dynamic events such as link/router failures and load balancing, the mechanisms in place to deal with these, and the complexity arising from them. It is unclear if our approach can be extended easily to account for these.

Our current work ignores the impact of packet transformations due to NATs and other middleboxes on complexity. Packet transformations could alter reachability sets in interesting ways, and might not be easy to configure. Fortunately, transformations were not employed in any of the networks we studied. We do believe, however, it is possible to extend our techniques to account for on-path changes to IP headers.

Of course, our approaches do not account for techniques employed above Layer-3 or at very low levels. In particular, we currently do not have an easy way to quantify the complexity of mechanisms which use higher-layer handles (e.g. usernames and services) or lower-layer identifiers such as MAC addresses. One potential approach could be to leverage dynamic mappings from the high/low level identifiers to IP addresses (e.g. from DNS bindings and ARP tables) and then apply the techniques we used this in paper.

Absolute vs relative configuration complexity. We note that our metrics for referential complexity and roles capture complexity that is apparent from the current configuration; hence they are *absolute* in nature. An increase in these metrics indicates growing complexity of implementation, meaning that configuration-related tasks could be harder to conduct. However, the metrics themselves do not reflect how much of the existing configuration is *superfluous*, or equivalently, what level of configuration complexity is actually necessary. For this, we would need a *relative* complexity metric that compares the complexity of the existing configuration against the simplest configuration necessary to implement the operators goals (including reachability, cost, and other contraints). However, determining the simplest configuration that satisfies these requirements is a hard problem and a subject for future research.

9 Related Work

The work most closely related to ours is [18], which creates a model of route redistribution between routing instances and tries to quantify the complexity involved in configuring the redistribution logic in a network. Glue Logic and our complexity metrics are similar in that both create abstract models of the configuration files and calculate complexity based on that information. However, while [18] limits itself to the configuration complexity of route redistribution (the "glue logic"), we examine both configuration and inherent complexity, and the relationship between the two. Our approach also accounts for complexity arising from the routing, VLANs and filtering commands in a configuration file.

Our study is motivated by [19, 13], which studied operational networks and observed that the configuration of enterprise networks are quite intricate. In [19, 13], models were developed to capture the interaction between routing stanzas in devices. However, to make inferences about the complexity of the networks studied, the authors had to manually inspect the models of each network. Our work automates the process of quantifying complexity.

As mentioned in Section 4, we borrow from [19] the idea of a routing instance and use it as a way to group routing protocols. Also, our referential dependence graph is similar to the abstractions used in [6, 9]. Unlike [6, 9] our abstraction spans beyond the boundaries of a single device, which allows us to define the complexity of network-wide configuration.

Several past studies such as [12, 10, 28, 26, 27] have considered how network objectives and operational patterns can be mined from configuration files. Of these, some studies [28, 26, 27] calculate the reachability sets and argue for their usage in verifying policy compliance. In contrast, the group of complexity metrics we provide allow operators to not only verify policy compliance, but they also quantify the impact of policy decisions on the ability to achieve a simple network-wide configuration. Complementary to [10], which proposes high-level constraints that if met ensure the correctness of routing, we start with the assumption that the network is correct and then derive its properties.

Contrary to the "bottom-up" approach we take, several studies [8, 15, 3] have considered how to make network management simpler by building inherent support for the creation and management of network policies. We presume that our study of configuration and inherent complexity can inform such ideas on clean slate alternatives. Finally, our metrics could be easily integrated into existing configuration management tools such as AANTS [1] and OpenView [16], and can aid operators in making informed changes to their network configurations.

The notion of "complexity" has been explored in domains such as System Operations [7]. In [7], complexity is defined as the number of steps taken to perform a task, similar to our metrics. Recently, Ratnasamy has proposed that protocol complexity be used in addition to efficiency to compare network protocols [23]. Just as Ratnasamy's metrics help choose the right protocol, our metrics help pick the right network design.

10 Conclusions

Configuration errors are responsible for a large fraction of network outages, and we argue that as networks become more complex the risk of configuration error increases. This paper takes the first step towards quantifying the types of complexity that lead operators to make configuration mistakes. Creating such metrics is difficult as they must abstract away all non-essential aspects of network configuration to enable the meaningful comparison of networks with very different sizes and designs.

In this paper, we define three metrics that measure the complexity of a network by automatic analysis of its configuration files. We validate the metrics' accuracy and utility through interviews with the network operators. For example, we show networks with higher complexity scores require more steps to carry out common management tasks and require more tools or more process discipline to maintain. Our study also generated insights on the causes of complexity in enterprise networks, such as the impact of the cost of network devices on routing design choices and the effect of defining multiple classes of subnets and multiple device roles.

We believe our metrics are useful in their own right, and we show how they can aid with finding configuration errors and understanding a network's design. However, our hope is that these metrics start a larger discussion on quantifying the factors that affect network complexity and management errors. The definition of good metrics can drive the field forward toward management systems and routing designs that are less complex and less likely to lead human operators into making errors.

Acknowledgements. We would like to thank our shepherd, Kobus van der Merwe, and the reviewers for their useful feedback. We would also like to thank Dale Carder, Perry Brunelli, and the other operators for their network configuration files. This work was supported in part by an NSF CAREER Award (CNS-0746531) and an NSF NeTS FIND Award (CNS-0626889).

References

[1] Authorized Agent Network Tool Suite (AANTS). http://www.doit.wisc.edu/network/upgrade/faq/aants.asp.

[2] ACHARYA, S., WANG, J., GE, Z., ZNATI, T., AND GREENBERG, A. Simulation study of firewalls to aid improved performance. In *ANSS '06*.

[3] BALLANI, H., AND FRANCIS, P. CONMan: A Step towards Network Manageability. In *Proc. of ACM SIGCOMM* (2007).

[4] BENSON, T., AKELLA, A., AND MALTZ, D. A. Operator questionnaire. http://pages.cs.wisc.edu/ tbenson/questionnaire.html.

[5] BENSON, T., AKELLA, A., AND MALTZ, D. A. A case for complexity models in network design and management. Tech. Rep. 1643, UW Madison, August 2008.

[6] CALDWELL, D., GILBERT, A., GOTTLIEB, J., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. The cutting EDGE of IP router configuration. In *HotNets* (2003).

[7] CANDEA, G. Toward quantifying system manageability. In *HotDep* (2008), USENIX Association.

[8] CASADO, M., FRIEDMAN, M., PETTITT, J., MCKEOWN, N., AND SHENKER, S. Ethane: Taking Control of the Enterprise. In *SIGCOMM '07*.

[9] CHEN, X., MAO, Z. M., AND VAN DER MERWE, J. Towards automated network management: network operations using dynamic views. In *INM '07*.

[10] FEAMSTER, N. Rethinking routing configuration: Beyond stimulus-response reasoning. In *WIRED* (Oct '03).

[11] FELDMANN, A., AND MUTHUKRISHNAN, S. Tradeoffs for packet classification. In *INFOCOM 2000*.

[12] FELDMANN, A., AND REXFORD, J. IP network configuration for intradomain traffic engineering. *Network, IEEE 15* (Sep '01).

[13] GARIMELLA, P., SUNG, Y.-W. E., ZHANG, N., AND RAO, S. Characterizing VLAN usage in an operational network. In *INM '07*.

[14] GRAY, J., Ed. *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan Kaufmann, 1991.

[15] GREENBERG, A., HJALMTYSSON, G., MALTZ, D. A., MYERS, A., REXFORD, J., XIE, G., YAN, H., ZHAN, J., AND ZHANG, H. A Clean Slate 4D Approach to Network Control and Management. *ACM Sigcomm CCR* (2005).

[16] HEWLETT-PACKARD. Enterprise Management Software: HP OpenView. http://h20229.www2.hp.com/.

[17] KAMIYA, T., KUSUMOTO, S., AND INOUE, K. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Softw. Eng. 28*, 7 (2002).

[18] LE, F., XIE, G. G., PEI, D., WANG, J., AND ZHANG, H. Shedding light on the glue logic of the Internet routing architecture. In *SIGCOMM* (2008).

[19] MALTZ, D. A., ZHAN, J., XIE, G., HJALMTYSSON, G., GREENBERG, A., AND ZHANG, H. Routing Design in Operational Networks: A Look from the Inside. In *SIGCOMM* (2004).

[20] MATHWORKS. *Statistics Toolbox for Use with MATLAB*, 1999.

[21] MCCABE, T., AND BUTLER, C. Design Complexity Measurement and Testing. *Communications of the ACM 32*, 12 (1989).

[22] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. A. Why do Internet services fail, and what can be done about it? In *USITS* (2003).

[23] RATNASAMY, S. Capturing Complexity in Networked Systems Design: The Case for Improved Metrics. In *HotNets* (2006).

[24] RYBACZYK, P. *Network Design Solutions for Small-Medium Businesses*. Cisco, 2004.

[25] THOMAS, T., AND KHAN, A. *Network Design and Case Studies (CCIE Fundamentals)*. Cisco, 1999.

[26] WONG, E. W. W. Validating network security policies via static analysis of router ACL configuration. Master's thesis, Naval Postgraduate School (U.S.), 2006.

[27] XIE, G., ZHAN, J., MALTZ, D. A., ZHANG, H., GREENBERG, A., HJALMTYSSON, G., AND REXFORD, J. On static reachability analysis of IP networks. In *Proc. IEEE INFOCOM* (2005).

[28] ZHANG, B., NG, T. S. E., AND WANG, G. Reachability monitoring and verification in enterprise networks. In *SIGCOMM Poster* (Nov. 2008).

NetPrints: Diagnosing Home Network Misconfigurations Using Shared Knowledge

Bhavish Aggarwal[‡], Ranjita Bhagwan[‡], Tathagata Das[‡],
Siddharth Eswaran[*], Venkata N. Padmanabhan[‡], and Geoffrey M. Voelker[†]

[‡]Microsoft Research India [⋆]IIT Delhi [†]UC San Diego

Abstract

Networks and networked applications depend on several pieces of configuration information to operate correctly. Such information resides in routers, firewalls, and end hosts, among other places. Incorrect information, or *misconfiguration*, could interfere with the running of networked applications. This problem is particularly acute in consumer settings such as home networks, where there is a huge diversity of network elements and applications coupled with the absence of network administrators.

To address this problem, we present *NetPrints*, a system that leverages shared knowledge in a population of users to diagnose and resolve misconfigurations. Basically, if a user has a working network configuration for an application or has determined how to rectify a problem, we would like this knowledge to be made available automatically to another user who is experiencing the same problem. NetPrints accomplishes this task by applying decision tree based learning on working and non-working configuration snapshots and by using network traffic based problem signatures to index into configuration changes made by users to fix problems. We describe the design and implementation of NetPrints, and demonstrate its effectiveness in diagnosing a variety of home networking problems reported by users.

1 Introduction

A typical network comprises several components, including routers, firewalls, NATs, DHCP, DNS, servers, and clients. Configuration information residing in each component controls its behaviour. For example, a firewall's configuration tells it which traffic to block and which to let through. Correctness of the configuration information is thus critical to the proper functioning of the network and of networked applications. *Misconfiguration* interferes with the running of these applications.

This problem is particularly acute in consumer settings such as home networks given the huge diversity in network elements and applications which are deployed without the benefit of vetting and standardization that is typical of enterprises. An application running in the home may experience a networking problem because of a misconfiguration on the local host or the home router, or even on the remote host/router that the application attempts to communicate with. Worse still, the problem could be caused by the *interaction* of various configuration settings on these network components. Table 1 illustrates this point by showing a set of typical problems faced by home users. Owing to the myriad problems that home users can face, they are often left helpless, not knowing which, if any, of a large set of configuration settings to manipulate.

Nevertheless, it is often the case that another user has a working network configuration for the same application or has found a fix for the same problem. Motivated by this observation, we present *NetPrints* (short for Network Problem Fingerprints), a system that helps users diagnose network misconfigurations by leveraging the knowledge accumulated by a population of users. This approach is akin to how users today scour through online discussion forums looking for a solution to their problem. However, a key distinction is that the accumulation, indexing, and retrieval of shared knowledge in NetPrints happens *automatically*, with little human involvement.

NetPrints comprises client and server components. The client component, which runs on end hosts such as home PCs, gathers *configuration information* pertaining to the local host and network configuration, and possibly also the remote host and network that the client application is attempting to communicate with. In addition, it captures a trace of the network traffic associated with an application run and extracts a *feature vector* that characterizes the corresponding network communication. The client uploads this information to the NetPrints server at various times, including when the user encounters a problem and initiates diagnosis. We enlist the user's help in a minimally intrusive manner to have the uploaded information labeled as "good" or "bad", depending on whether the corresponding application run was successful or not.

[*]The author was an intern at Microsoft Research India during the course of this work.

[†]The author was a visiting researcher at Microsoft Research India during the course of this work.

The NetPrints server performs decision tree based learning on the labeled configuration information submitted by clients to construct a *configuration tree*, which encodes its knowledge of the configuration settings that work and ones that do not. Furthermore, it uses the labeled network feature vectors to learn a set of *signatures* that help distinguish among different modes of failure of an application. These signatures are used to index into a set of *change trees*, which are constructed using configuration snapshots gathered before and after a configuration change was made to fix a problem. At the time of diagnosis, given the suspect configuration information from the client, the NetPrints server uses a *configuration mutation* algorithm to automatically suggest fixes back to the user.

We have prototyped the NetPrints system on Windows Vista and made a small-scale deployment on 4 broadband-connected PCs. We present a list of 21 configuration-related home networking problems and their resolutions from online discussion boards, user surveys, and our own experience. We believe that all of these problems and others similar to them can be diagnosed and fixed by NetPrints. We were able to obtain the necessary resources to reproduce 8 of these problems for 4 applications in our small deployment and also our laboratory testbed. Since we do not have configuration data or network traces from a large population of users, we perform learning on real data gathered for the applications run in our testbed, where we artificially vary the network configuration settings to mimic real-world diversity of configurations. Our evaluation demonstrates the effectiveness and robustness of NetPrints even in the face of mislabeled data.

Our focus in this paper is on the diagnostics aspects of NetPrints. We are doing separate work on the privacy, data integrity, and incentives aspects as well but do not discuss these here. Also, our focus here is on network configuration problems that interfere with specific applications but do not result in full disconnection and, in particular, do not prevent communication with the NetPrints server. Indeed, these subtle problems tend to be much more challenging to diagnose than basic connectivity problems such as full disconnection. In future work, we plan to investigate the use of out-of-band communication (e.g., via a physical medium) to enable NetPrints diagnosis even with full disconnection.

2 Related Work

We discuss prior work on problem diagnosis in computer systems and in networks, and how NetPrints relates to it.

2.1 Peer Comparison-based Diagnosis

There has been prior work on leveraging shared knowledge across end hosts, which provides inspiration for a similar approach in NetPrints. However, the prior work differs from NetPrints in significant ways.

Strider [19] uses a state-based black-box approach for diagnosing Windows registry problems by performing temporal and spatial comparisons with respect to known healthy states. It assumes the ability to explicitly trace what configuration information is accessed by an application run. Such state tracing would be difficult to do with network configuration, which governs policy (e.g., port-based filtering) that implicitly impacts an application's network communication rather than being explicitly accessed by applications.

PeerPressure [18] extends Strider by eliminating the need to identify a single healthy machine for comparison. Instead, it relies on registry settings from a large population of machines, under the assumption that most of these are correct. It then uses Bayesian estimation to produce a rank-ordered list of the individual registry key settings presumed to be the culprits. While this unsupervised approach has the advantage of not requiring the samples to be labeled, it also means that PeerPressure will necessarily find a "culprit", even when there is none. This outcome might not be appropriate in a networking setting, where a problem might be unrelated to client configuration. Also, PeerPressure is unable to identify *combinations* of configuration settings that are problematic.

Finally, Autobash [15] helps diagnose and recover from system configuration errors by recording the user actions to fix a problem on one computer and then replaying and testing these on another computer that is experiencing the same problem. Autobash assumes support for causality tracking between configuration settings and the output, which is akin to state tracing in Strider discussed above.

2.2 Problem Signature Construction

There has been work on developing compact signatures for systems problems for use in indexing a database of known problems and their solutions.

Yuan et al. [21] generate problem signatures by recording system call traces, representing these as n-grams, and then applying support vector machine (SVM) based classification. Cohen et al. [8, 9] consider the problem of automated performance diagnosis in server systems. They use Tree-Augmented Bayesian Networks (TANs) to identify combinations of low-level system metrics (e.g., CPU usage) that correlate well with high-level service metrics (e.g., average response time).

In contrast, NetPrints uses a set of network traffic features, which we have picked based on our networking domain knowledge, to construct problem signatures. Since these network traffic features tend to be OS-independent, NetPrints would be in a position to share signatures across OSes. Furthermore, we use a decision tree based classifier to learn the signatures.

2.3 Network Problem Diagnosis

Active probing is widely used for diagnosing network problems. For example, Tulip [12] probes routers to localize anomalies such as packet reordering and loss. Such diagnosis relies on a model of how network elements such as routers operate. Likewise, several model- or rule-based engines have been developed for diagnosing configuration-related and other faults in wireless LANs. These include systems that rely on infrastructure-based monitoring (e.g., DAIR [5], Jigsaw [7]) and those that rely on cooperation among wireless clients (e.g., WiFiProfiler [6]).

Other diagnosis systems such as SCORE [11] and Sherlock [4] have modeled, and in some cases automatically discovered, dependencies between higher-layer, observable network events and the underlying network components. Formal methods have also been used to check the correctness of network configurations. For example, rcc [10] checks for a range of well-understood BGP properties.

In the context of NetPrints, it may be possible to construct such models for certain well-understood configuration settings (e.g., port-based filters), thereby allowing diagnosis based on active probing, rules, or formal methods. However, in general, configuration settings may not be documented or well-understood, hence NetPrints' black-box approach.

2.4 NetPrints Compared to Prior Work

We view NetPrints as being complementary to prior work on network diagnosis in two ways. First, NetPrints focuses on configuration problems that impact *specific* applications rather than on broad problems that impact the network infrastructure. Second, NetPrints uses a *blackbox* approach appropriate for arbitrary and poorly understood configuration information, avoiding the need for the network behaviour or dependencies to be modeled explicitly.

NetPrints draws inspiration from prior work on black-box techniques to diagnose systems problems and index them with signatures to enable recall. However, NetPrints' goal of identifying how to *mutate* a broken configuration to *fix a problem* leads us to use a different approach — decision tree based learning — compared to prior work. This is primarily because of the interpretable nature of a decision tree. Furthermore, NetPrints leverages domain-specific knowledge to construct *signatures* of networking problems. The diagnosis procedure in NetPrints is both state-based and signature-based.

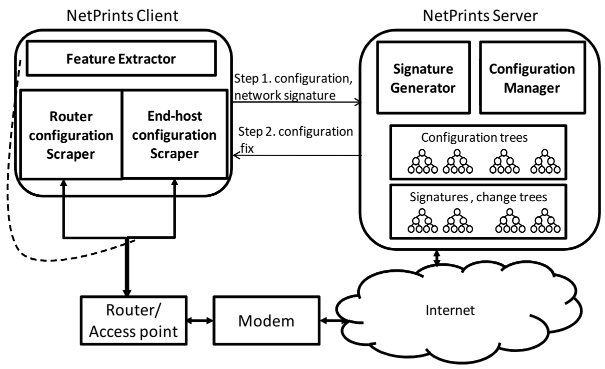


Figure 1: NetPrints system design

3 Overview of NetPrints Design

We begin with an overview of NetPrints, before turning to a more detailed discussion in the sections that follow.

Figure 1 depicts the client and server components of NetPrints, and their interaction. NetPrints has two modes of operation: "construction" and "diagnosis".

In the construction mode, the NetPrints server gathers configuration snapshots (Section 4) and network traffic features from NetPrints clients. This information is labeled as "good" or "bad" depending on whether the application run was successful or not. The NetPrints server, using this information, constructs a *configuration tree* (Section 5) that encodes its knowledge of which configuration settings work. It constructs a *change tree* (Section 7) based on the before and after snapshots of configuration changes that fixed a problem. Change trees are indexed by *network traffic signatures* (Section 6) that characterize how an application run fails. All these are constructed on a per-application basis.

When users experience a problem with an application, they invoke the diagnosis procedure. The NetPrints client, which runs on the user's machine, identifies which application to diagnose, either automatically (e.g., the application that last had focus) or with the help of the user. The client then gathers and uploads local configuration information and network traffic features, both labeled as "bad", to the NetPrints server (step 1 in Figure 1).

The NetPrints server performs diagnosis in two phases. In phase I, it uses the application-specific configuration tree to determine whether the client's configuration is problematic and, if so, identifies remedial *configuration mutations*, which it then conveys to the client (step 2 in Figure 1).

While configuration tree based diagnosis would work in many cases, it might fail, for instance, when there are "hidden" configuration parameters that impact a subset of the clients, so that the main configuration tree does not find anything amiss with the configuration of such clients (e.g., #4, #8, #10, and #12 in Table 1; see Sec-

#	App.	Router	Problem	Cause	Fix
1	VPN	WGR614	VPN Client does not connect	Stateful firewall was off	Turn on the stateful firewall
2	VPN	WRT54G	VPN drops connection after 3 minutes	(n/a)	Set MTU to 1350–1400, uncheck "block anonymous internet request", "filter multicast boxes" in router configuration
3	VPN	WRT54G	No VPN connectivity	No PPTP passthrough	turn on PPTP passhthrough
4	VPN	WRT54G	No VPN connectivity	double NAT, second NAT was dropping PPTP packets	Switch from PPTP server to SSTP server
5	File Sharing	any	Only unidirectional sharing	End-host firewall is not properly configured	Allow file sharing through all firewalls
6	File Sharing	WGR614v5	No file sharing	Client machine is on a domain, server machine is on workgroup	Put both machines either on the same domain or workgroup
7	FTP	any	Cannot connect to FTP server from outside home network	Port forwarding incorrect	Turn on port forwarding on port 21
8	FTP	WGR614	Cannot connect to FTP server at home	Client firewall blocking traffic, active FTP being used	Turn on firewall rule to allow active FTP connections
9	VPN server	WRT54G	PPTP server behind NAT does not work despite port forwarding and PPTP passthrough allowed	IP of server is 192.168.1.109, which is inside default DHCP range of router; router's port forward to IPs inside default range of router does not work	Use static IP outside DHCP range for server
10	Outlook	WRT54G	Outlook does not connect via VPN to office	Default IP range of router was same as that of the remote router	Change the IP range of home router
11	Outlook	WGR614	Router not able to email logs	SMTP server not configured properly	Setup SMTP server details in the router configuration
12	Outlook	Linksys	Not able to send mail through Linksys router; Belkin router works fine	MTU value too high for remote router, so remote router discards packets	Reduce MTU to 1458 or 1365
13	SSH	WGR614	SSH client times out after 10 minutes	NAT table entry times out	Change router or increase NAT table timeout
14	Office Communicator	WRTP54G	IM client does not connect to office	DNS requests not resolved	Turn off DNS proxy on router
15	STEAM games	WGR614	Listing game servers causes connection drops	Router misinterprets the sudden influx of data as an attack and drops connection	Upgrade to latest firmware
16	Real-Player	BEFW11s4	Streaming kills router	Firmware upgrade caused problems	Downgrade to previous firmware
17	Xbox	WRT54G	Xbox does not connect and all games do not run	Some ports are blocked and NAT traversal is restricted	Set static IP address on Xbox and configure it as DMZ, enable port forwarding on UDP 88,TCP 3074 and UDP 3074, disable UPnP to open NAT
18	Xbox	WRT54G	Xbox works with wired network but not with wireless	WPA2 security is not supported	Change wireless security feature from WPA2 to WPA personal security
19	Xbox	WGR614	Not able to host Halo3 games	NAT settings too strict	Set Xbox as DMZ
20	IP Camera	DG834GT	Camera disconnects periodically at midnight, router needs reboot	DHCP problem	Configure static IP on the camera
21	ROKU	DIR-655	ROKU did not work with mixed b, g and n wireless modes	(n/a)	Change to mixed b and g mode

Table 1: Recent configuration-related problems in home networks.

tion 7 for an elaboration of #8). So in phase II, the NetPrints server uses a signature of the application problem to identify the appropriate change tree, which has been constructed by focusing specifically on such problematic cases. If the change tree is unable to diagnose the problem either, NetPrints gives up; it is possible that the problem is not configuration-related.

4 Configuration Scraper

The configuration scraper gathers configuration information from the local Internet Gateway Device (IGD) — which we loosely refer to as the local router — the local client host, and possibly also from a remote host and network.

4.1 Internet Gateway Configuration

The scraper gathers two categories of IGD information:
(i) *IGD identification information:* This information includes the make, model and firmware version of the device, which in most cases is a home router, although in some cases it could be a DSL or cable modem. The scraper obtains this information using the UPnP interface which is supported and enabled by default on most modern IGDs [16]. UPnP is a standard with which our client can obtain basic information such as the URL for the Web interface for the device, and the make and model of the device. However, if the router has UPnP turned off, we ask the user to manually input the IGD identification information. Note that the user will need to input this information only very rarely, i.e., when they install a new router that has UPnP turned off.
(ii) *Network-specific configuration information:* The IGD also includes configuration information such as port forwarding and triggering tables, MTU value, VPN pass-through parameters, DMZ settings, and wireless security settings. The scraper uses both the UPnP interface and the Web interface that most routers and modems provide to glean such configuration information. On some of the routers we tested, the port tables from the Web page and the port tables from the UPnP interface were not kept consistent with each other. Consequently, we scrape and combine the tables via both interfaces. Some router firmware versions also allow us to scrape the maximum NAT table size and the per-connection timeout for each table entry. These fields can be particularly useful in diagnosing problems such as #2 and #13 in Table 1.

While the UPnP interface gives us access to only device-identifying parameters and the UPnP port forwarding and port triggering tables, the Web interface is richer but not standardized across routers.

In particular, there is no standardized way for parsing the HTML to extract the (key,value) pairs defining the configuration. To address this problem, we make the observation that each configuration Web page of the device is typically an HTML form that includes a "submit" operation. We invoke this operation programmatically on each configuration Web page. Doing so causes the creation of an HTTP POST request containing all of the (key,value) pairs in an easy-to-parse form. For example, the body of the POST request might contain: `submit_button=index&dhcp_start=100&dhcp_num=50&dhcp_lease=1440`. It is then straightforward to extract the various DHCP-related configuration settings from this string.

While scraping Web forms, the NetPrints client asks for the user name and password set on the router. The user will need to input this information once, after which a cookie within the NetPrints client will remember the input to use every time it scrapes the Web interface of the router. Note that no such information is needed for the UPnP-based scraping.

4.2 Local Host Configuration

There is also much configuration information of relevance to network operation on the local client host itself, such as whether the network connection is wired or wireless, whether TCP window scaling is on or off, and end-host firewall rules. We currently scrape all interface-specific network parameters, TCP-specific parameters and firewall rules from the end-host. Our implementation uses the `netsh` utility available on Windows operating systems to get this information.

4.3 Remote Configuration

In general, the configuration of the remote host and network also impacts the health of network applications. In some cases, the configuration information at the remote end may be inaccessible to us (e.g., the remote host might be a server in a different administrative domain). In other cases, however, the remote host might be under the control of the same user as the local host. One example is communication between a client and a server on the same home network, say as part of a file or printer sharing application. Another example is when a user tries to access a service running in their home network from an external location, such as a user in their workplace accessing their home FTP server.

If the user installs the NetPrints client on the remote host as well, then, using simple password-based authentication, the local NetPrints client can obtain remote host and network configuration information. For every application, the NetPrints client keeps track of all remote hosts that it accesses or tries to access and, if the remote site runs NetPrints under the same administration as the local NetPrints client, the local client collects remote configuration information.

The impact of remote configuration on the health of a networked application can vary. In some instances, a problem may arise because of misconfiguration at the remote end. For example, if the remote network blocks access to port 21, attempts to connect to an FTP server on that network would fail. In other instances, the remote configuration may not be problematic *per se*. Rather, it is the mismatch between the local configuration and the remote configuration that is problematic. For instance, while some users might be able to access a file server, others may not be able to because their credentials are not included in the access control list (ACL) on the server. In other words, there is a mismatch between the local configuration (the local user's credentials) and the remote configuration (the ACL on the server).

Once the remote configuration information has been obtained, it is incorporated into NetPrints' diagnostics procedure in the same manner as local configuration information. The one exception, which requires some additional pre-processing, is incorporating the mismatch between local and remote configurations, a problem we turn to next.

4.4 Composing Configurations

Since it is the combination of local and remote configurations that matters in some cases, we introduce new, composite configuration parameters that are derived by combining local and remote configurations parameters. Conceptually, a composite parameter, C, is a Boolean derived by applying a comparison operator, $\bigotimes$, to the local parameter, L and a remote parameter, R. That is, $C = L \bigotimes R$.

The specific comparison operators we focus on are equality "$=$" and set membership "$\in$". For example, if the local Windows workgroup $L1$ and the remote Windows workgroup $R1$ are the same, then $C1$ = 1. Else, $C1$ is set to 0. Another example is of checking whether the local username $L2$ is part of the remote ACL $R2$ for a file sharing application. If it is (i.e., $L2 \in R2$), the corresponding composite parameter $C2$ is set to 1.

4.5 Reducing Composite Parameters

Blindly comparing all pairs of local and remote configuration parameters results in an explosion in the number of composite parameters, most of which would be meaningless (e.g., a comparison of the local user name with the DHCP setting on the remote router). To limit the number of such composite parameters, without requiring an understanding of the semantics of the parameters, Netprints (1) only uploads composites that explicitly match, and (2) excludes parameters that exclusively have one value from the learning process.

In our experimental setup, the configuration scraper captures roughly 500 configuration parameters from the router and 2100 from the end-host, at each of the local and remote ends. This yields an additional 1500 composite parameters, after reduction is applied, and hence a total of (2100+500)x2+1500=6700 parameters.

5 Configuration Trees

Based on the labeled configuration information obtained from clients, we construct per-application decision trees, called *configuration trees*, which encode NetPrints' learning of which parameter settings work and which do not. We start with a brief introduction to decision trees and then turn to how NetPrints constructs configuration trees and uses these for diagnosis.

5.1 Decision Trees

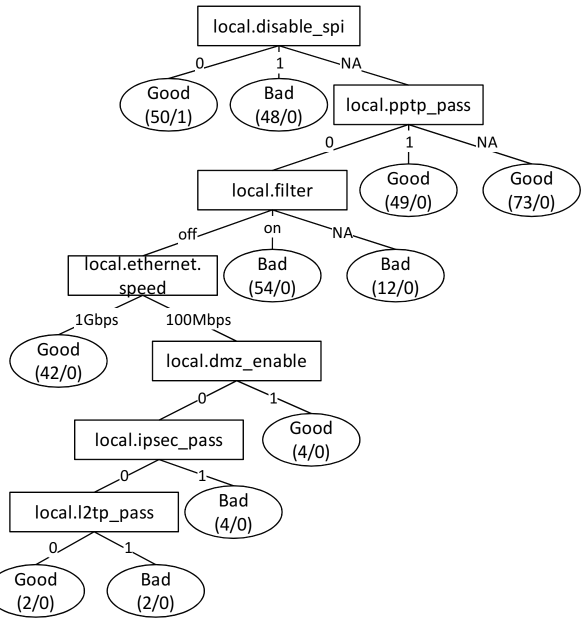


Figure 2: Configuration tree for the VPN client application discussed in Section 9.2.

NetPrints uses decision trees as a basis for performing configuration mutation. A decision tree (see Figure 2 for an example) is a predictive model that maps observations (e.g., a client's network configuration) to their target values or *labels* (e.g., "good" or "bad"). Each non-leaf node in the decision tree corresponds to an attribute of the observation, and the edges out of the node indicate the values that this attribute can take. Thus, each leaf node corresponds to an entire observation and carries a label. Given a new observation, we start at the root of the decision tree, walk down the tree, taking branches corresponding to the individual attributes of the observation, until we reach a leaf node. The label on the leaf node identifies configurations as "good" or "bad".

There are several algorithms for decision tree learning. We chose a widely-used algorithm, C4.5 [14], which builds trees using the concept of information gain.

The C4.5 tool starts with the root, and at each level of the tree chooses the attribute to split the data that reduces the entropy by the maximum amount. The result is that the branch points (i.e., non-leaf nodes with multiple children) at the higher levels of the tree correspond to attributes with greater predictive power, i.e., those with distinct values or ranges corresponding to distinct labels.

When the training data is noisy (e.g., it contains mislabeled samples) or there are too few samples, there is the risk that the above algorithm will over-fit the training data. To address this concern, C4.5 also include a pruning step, wherein some branches in the tree are discarded so long as this does not result in a significant error with respect to the training data (a process called generalization). C4.5 uses a confidence threshold to determine when to stop pruning. In our implementation, we use the default threshold. A consequence of pruning is that, if the number of samples is insufficient, these samples will not be reflected in the decision tree.

A decision tree has two key properties. First, it enables classification of observations that include both quantitative and categorical attributes. For example, the decision tree in Figure 7 includes quantitative attributes such as the WAN MTU and categorical attributes such as the security mode. Second, a decision tree is amenable to easy interpretation. It not only enables classification of observations, it also helps identify in what minimal way an observation could be *mutated* so as to change its label (e.g., from "bad" to "good"). We elaborate on this property in Section 5.4. The interpretability of decision trees, in particular, makes it an attractive alternative to SVMs or Bayesian classification.

5.2 Labeling Configuration Information

As explained in Section 4, the NetPrints client extracts configuration information from the local host and network as well as from the remote end. Before this information can be fed to the NetPrints server, it has to be labeled as either "good" or "bad", depending on whether the application in question was working or not. In general, it is hard to determine automatically whether an arbitrary application is working well. We sidestep this difficulty by enlisting the help of the human user to label the application runs. If we assume that the majority of users are honest, then most of the configuration information submitted to the NetPrints server will be labeled correctly. As we discuss in Section 9.6, decision tree based learning employed by the server is robust to mislabeling to a large extent. Also, in Section 10.1, we discuss ways of reducing the burden of labeling on users.

5.3 Configuration Manager

The configuration manager at the NetPrints server uses the labeled configuration information submitted by clients to learn and construct *per-application configuration trees*, using C4.5. The tree comprises decision nodes, which are branch points, and leaf nodes, which correspond to "good" or "bad" labels. A path from the root to a "good" ("bad") leaf node indicates the parameter settings for a working (non-working) configuration.

Figure 2 shows an example of such a configuration tree that we generated for the Microsoft Connection Manager VPN application [13] using configuration information from clients using several different router devices (see Table 5). We note that the `local.disable_spi` attribute (corresponding to whether stateful packet inspection (SPI) is disabled) is the clearest, even if not a perfect, indicator of whether a configuration is good or bad. So it is at the root of the configuration tree.

Note that a decision node in the configuration tree may have a branch labeled NA (not applicable), in addition to branches corresponding to the various parameter settings (e.g., 0 and 1 with `local.disable_spi`). The NA branch is needed since some parameters may be absent in particular routers.

Currently, the decision tree algorithm we use does not allow for incremental training of the trees, hence we use a cache of configurations to perform the training at each step. However, incremental update based algorithms exist [17] and we plan to evaluate these in future work.

5.4 Misconfiguration Diagnosis

When users experience application failure, they initiate the diagnosis procedure on the NetPrints client. The NetPrints client scrapes and submits its suspect configuration information to the NetPrints server for diagnosis. At the server end, the configuration manager starts at the root and walks down the configuration tree corresponding to the application that the user is complaining about. If it ends at a "bad" node, it means that the client's configuration is known to be non-working. On the other hand, if it ends at a "good" node, it means that the configuration tree is unable to help with the diagnosis, a case we consider in Section 7.

If the client's configuration corresponds to a known "bad" state, then the goal of diagnosis is to identify the *configuration mutations* that would move the configuration to a known "good" state. In general, there would be multiple "good" leaf nodes, so which one should we mutate towards?

Intuitively, we would like to pick the mutation path that is easiest to traverse. The easiest path is not necessarily the one with the fewest changes. The difficulty of making the changes also matters. For example, changing the router hardware (say switching from a Linksys router to a Netgear router) would likely be more difficult than modifying a software-settable parameter on

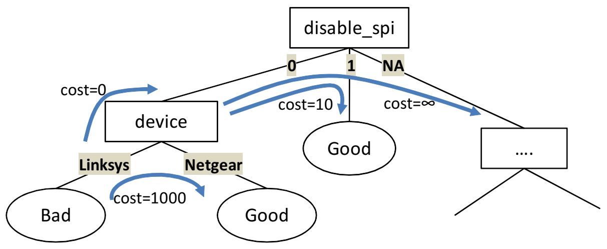


Figure 3: Illustration of the costs of different configuration mutations.

the router because of the costs involved. Even among software-settable parameters, some changes might be less desirable, and hence more difficult to make, than others. For example, putting the client host on the DMZ, and thereby exposing it to external traffic, would likely be less desirable than say enabling port forwarding for a specific port.

To determine the degree of difficulty automatically, NetPrints records the frequency with which various configuration parameters are modified across all clients. It might find, for instance, that the `disable_spi` parameter is modified 100 times as often as the `device` is. We quantify the cost of a mutation as the reciprocal of the change frequency, possibly scaled by a constant factor, of the corresponding configuration parameter. We might record some spurious changes, say when a mobile client moves from one network to another and mistakenly thinks that its router `device` and various configuration settings have "changed". However, we can counter the effect of mobility by hard-coding the fact that changing routers is a low-frequency, and therefore high-cost, change. Thereafter, when a client is mobile and associates with a new router, we infer that the corresponding changes in configuration detected by NetPrints are because the router changed, not because the user explicitly changed configurations. Hence we do not increase the change frequency of the parameters.

Figure 3 illustrates how the configuration tree is annotated with costs. The cost of changing the router `device` is 100 times greater than the cost of changing the `disable_spi` setting. Some mutations are impossible to effect, so the corresponding cost is set to ∞. For instance, it is not possible to set `disable_spi` to NA when the parameter does not exist on the router in question. Also, note that the cost is incurred only when a parameter is changed, hence the zero cost for merely walking up the tree.

Given the mutation costs indicated above, we compute the cost of moving from a "bad" leaf node to a "good" leaf node as the sum of the costs of the mutations on the path from the former to the latter. NetPrints recommends the set of mutations corresponding to the path with the lowest cost.

5.5 Going Beyond Configuration Trees

The per-application configuration trees help diagnose misconfigurations based on configuration information on which there is broad agreement across a large number of participating NetPrints clients. Basically, the configuration manager learns about the goodness or otherwise of various configuration settings based on *static snapshots* of labeled configuration information uploaded by clients.

However, as noted in Section 3, diagnosis based on the configuration tree would not work in the case of misconfigurations that are exceptions to the norm. Such exceptions could arise, for instance, from hidden configuration settings (as noted in Section 3) or from decision tree pruning (as explained in Section 5.1). In such cases, the configuration tree might suggest that the suspect configuration is "good" and hence not be in a position to suggest any mutations.

To address this issue, we introduce *change trees*, which seek to learn based on *dynamic* information, i.e., configuration changes. Furthermore, to reduce the chances of exceptions being buried by the mass, we use *network traffic signatures* to index the change trees.

Note, however, that multiple configuration errors could yield the same network signature, so a network signature is, in general, not as informative as the configuration information itself. Hence our approach is to use the configuration tree as the option, with the change trees indexed using network signatures as the fallback option.

We now discuss how NetPrints constructs network traffic signatures, and then turn to change trees.

6 Network Traffic Signature

We use a network traffic signature to characterize application runs. For instance, an application could fail because it is unable to establish a TCP connection (SYN handshake failure) or because the TCP connection is reset prematurely. The *network traffic signature* is used to distinguish between these failure modes. In essence, the signature records the *symptom* of the failure, which is used to index the change trees of the application, as explained in Section 7.

The basic approach is for the NetPrints clients to extract a set of *network traffic features* from a packet trace of the application run. The NetPrints server then applies learning on these features to identify the important ones, which are then included as part of the network traffic signature for that application.

6.1 Network Traffic Feature Extractor

The network traffic feature extractor characterizes the network usage of each application running on the client machine. In our current implementation, it uses the Winpcap library and the IPHelper API on Windows to tie all

#	Feature Description	Unit
1	TCP: Three SYN no response	5-tuple
2	TCP:RST after SYN, no data exchanged	5-tuple
3	TCP:RST after no activity for 2 mins	5-tuple
4	TCP:RST after some data exchanged	5-tuple
5	UDP: Data sent but not received	5-tuple
6	Other: Data sent but not received	src-dst IP addr pair
7	All: No data sent or received	all traffic

Table 2: Network traffic features and the unit of communication over which the feature is extracted. Each feature is maintained separately for inbound and outbound directions, except for "All", which is maintained for both directions together.

observed network traffic to the individual processes, and hence applications, running on the client machine. For each running application, it extracts a set of features by examining its network activity. These features form the *feature vector* for the application.

Table 2 lists the set of features we extract in the form of rules. Most of these features are maintained separately for the inbound (I) and outbound (O) directions, depending on whether the communication was initiated by the remote host or by the local host. While many of these features are extracted on a per-5-tuple basis (i.e., on per-connection basis for TCP), we combine the features across all connections of an application to compute the bits of the feature vector. Specifically, if *at least* one connection of an application satisfies any of these rules, the corresponding bit in the feature vector is set. Note that it is possible for multiple bits in an application's feature vector to be set. Also, while all of the features we consider at present are binary, the feature set could be expanded to include non-binary features.

We identified the set of features in Table 2 based on empirical observations of the ways in which an application's network communication may typically fail. The first four features in the table capture various kinds of TCP-level issues that we commonly see in malfunctioning applications. Several applications and services such as multimedia streaming, DNS and VPN clients use transport protocols other than TCP. For all of these, the lack of connectivity in one direction often indicates a networking problem. Consequently, we have included features #5 and #6 to capture the behavior of such applications. For both features, we use a timeout of 2 minutes: if no data is received for a period of 2 minutes, we interpret this as a possible problem and set the feature. Feature #7 characterizes a total loss of connectivity for an application using any transport protocol; problem #18 in Table 1, for instance, is a scenario in which our system would use this feature.

Finally, we briefly discuss two issues pertaining to the recording of network features for an application run. First, since the instance of an application could run for an extended period of time (e.g., a Web browser could run for days or weeks), we only consider network traffic features over a short window of time (typically a few minutes long) extending into the recent past. Second, extracting the network traffic feature for an application run requires capturing its traffic. One possibility is to run traffic capture continuously, which has the advantage that a record of the traffic will be available even when an application run failed.

To reduce the overhead of the NetPrints client with such traffic continuous capture, we split the network signature generator into two parts: a lightweight, continuously running component to capture selected packet headers and connection-to-process bindings, and a relatively more CPU-intensive component that creates the feature vector from the trace only when needed. Measurements of our implementation show that the overhead is low (0.8% CPU load) on a 1.8 GHz laptop PC running Windows Vista Enterprise, while streaming video over the Internet and simultaneously synchronizing email folders with the server.

6.2 Network Signature Generator

The NetPrints client records and uploads the feature vector for an application run to the NetPrints server, either when the user invokes NetPrints to complain about a non-working application or when the user is prompted, as explained in Section 5.2. In either case, the feature vector is labeled as "good" or "bad", just as the accompanying configuration information is. The NetPrints server then applies learning on the mass of labeled feature vectors for an application to identify the most significant features, i.e., ones that correspond most strongly to the fate of an application run. These significant features define the *network signature* of the application.

The signature generator, again, uses the C4.5 algorithm to learn the network signatures, which are represented as *per-application signature trees*. However, unlike with learning applied to configuration information, interpretability is not necessary for signature construction (since there are no mutations to perform), so we could have also used a different learning algorithm such as SVM. Figure 5 shows the signature tree generated for an FTP application, where 2 features, out of the 13 in all, are sufficient to capture the network problems seen.

7 Change Trees

As noted in Section 5.5, *change trees* are used as the fallback option when the configuration tree fails to diagnose a problem. To understand why configuration tree based diagnosis might fail, consider problem #8 in Table 1. The FTP server in question enables passive mode by default, so that all connections are initiated at the client end. However, in a small number of cases, the server may disable passive mode, i.e., only the server can initiate FTP data connections. The client will disallow these connections unless the client-side firewall has been configured to let them in. Note that the application-specific configuration parameter that captures the information that the server has disabled passive FTP is "hidden" from NetPrints since, in general, NetPrints is not in a position to scrape such parameters. Nevertheless, there are non-hidden configuration parameters (the firewall parameters on the client, in this instance) that could be manipulated to fix the problem.

Since the discriminating parameter is hidden, it is hard to tell apart the majority of clients that are configured for passive mode from the minority that are configured for active mode. So the majority prevails and the configuration tree learns to ignore the firewall settings since these are not of relevance for the majority of clients (i.e., FTP works for such clients regardless of the firewall settings). So when an active FTP connection to a client fails, the configuration tree would *not* find anything amiss with its configuration, i.e., it will find the configuration to be "good" and leave no scope for remedial action.

Change trees try to address this problem by isolating the cases where a traversal of the configuration tree ends up in leaf nodes labeled as "good" and then applying learning separately on these. For the purposes of this learning, the suspect configurations (which the configuration tree thinks of as "good") are labeled as "bad". Since we also need configurations labeled as "good" to perform learning, the NetPrints client in such cases looks for any out-of-band configuration changes that are made and, when such a change is detected, it prompts the user to determine whether the application problem has now been resolved. If and when the user indicates that the problem has been resolved, it uploads a "good" configuration to the NetPrints server.

The NetPrints server uses the C4.5 algorithm to learn a decision tree — the *change tree* — based on the change information: the "before" configurations labeled as "bad" and the "after" configurations labeled as "good". To isolate the relevant cases and minimize the mixing of unrelated problems, we use the network signature corresponding to application failure to index the change trees. So, in effect, each "bad" leaf node in the signature tree can point to a separate change tree.

Each change tree is also traversed the same way as the main configuration tree. If a traversal of the relevant change tree also ends in a leaf node labeled as "good", NetPrints gives up. It could be that NetPrints does not have sufficient information to identify the misconfiguration or that the problem is not configuration-related.

8 Summary of NetPrints Operation

In summary, NetPrints performs the following steps in the construction and diagnosis phases.

Construction Steps:

1) The NetPrints clients upload labeled configuration information and network feature vectors to the NetPrints server, either when users invoke NetPrints for diagnosis or are prompted by NetPrints (the latter happens for a small fraction of application runs).

2) The NetPrints server feeds the labeled configuration information into the C4.5 decision tree algorithm to construct an *application-specific configuration tree*. It feeds the labeled network feature vector to the same algorithm to learn an *application-specific signature tree*.

3) During the diagnosis phase (see below), if the traversal of the configuration tree with a suspect configuration terminates in a "good" leaf node, then this configuration, now labeled as "bad", is fed into the *application-specific change tree* construction procedure.

4) Furthermore, the NetPrints client prompts the user to determine if future configuration changes, if any, help restore the application to a working state. If so, the corresponding configuration, labeled as "good", is fed into change tree construction at the NetPrints server.

Diagnosis Steps:

1) When the user encounters a problem and invokes diagnosis, the NetPrints client uploads configuration information, along with the network feature vector for the affected application, to the NetPrints server.

2) The NetPrints server traverses the configuration tree with the suspect configuration submitted by the client. If this traversal ends in a "bad" leaf node, NetPrints identifies the set of configuration mutations, with the lowest cost, that would help move the configuration to a "good" state.

3) If the traversal of the configuration tree ends in a "good" leaf node, the NetPrints server first computes the signature of the failed application run based on the network feature vector submitted by the NetPrints client.

4) The NetPrints server uses the signature to identify the relevant change tree and then traverses this tree with the suspect configuration. If this traversal ends in a "bad" leaf node, then the NetPrints server uses the same procedure as indicated above to identify mutations.

5) However, if the traversal of the change tree ends in a "good" leaf node, the NetPrints server gives up.

Client		Server	
Config scraper	Feature extractor	Config manager	Signature generator
3159	701	1767	460

Table 3: Lines of code for NetPrints prototype.

9 Experimental Evaluation

Our experimental evaluation of NetPrints is based on the prototype we have implemented on Windows Vista SP1, using a combination C# and C++. Table 3 summarizes some information on the implementation; for C4.5, we used a standalone distribution [14].

We deployed the NetPrints client on 4 hosts behind separate broadband connections. Given this small scale of our current deployment, we used hosts on a separate testbed to scale up the effective size of the deployment, as we elaborate on below. The data gathered from the testbed was used in the "construction" phase of NetPrints during which the NetPrints server, which ran on a separate host, learnt the configuration, signature, and change trees. The "diagnosis" phase was initiated from one of the 4 broadband hosts and involved communication with the NetPrints server to perform diagnosis.

9.1 Setup and Methodology

We evaluated NetPrints with 4 applications: Microsoft's VPN client, a Perl-based FTP client, Windows Vista file sharing, and Xbox Live. These applications were run both on our testbed (construction phase) and a separate set of broadband hosts (diagnosis phase). Our testbed included a Windows Vista laptop (two in the case of the file sharing application), each running the NetPrints client, and also an Xbox 360 gaming console, all of which were uplinked via a home router and a DSL broadband modem. We also had 4 other hosts, including 2 at people's homes, on separate broadband connections, each running the NetPrints client from which diagnosis was initiated. Finally, for the FTP application, we also had an external machine running the client, not on a broadband network, that connected to one of the broadband hosts via the Internet.

For diversity, we used 7 different routers from Netgear, Linksys, D-Link, and Belkin (Table 5), in turn, as the home router in our testbed. To obtain greater diversity, as one might see with a large-scale deployment, we varied the configuration settings on these routers, re-running the applications each time. Note that although we varied these configuration settings artificially, we ran the applications and NetPrints just as they would be run in the real world.

We identified 11 parameters (Table 4) and learnt variations in their settings based on a study of online discussion forums. Even with this subset of parameters, many

Router parameters:
MTU {1100, 1200, 1300, 1400, 1500 bytes}: supported by all routers except Belkin F5D7230.
VPN-specific parameters {on, off}: the D-Link router supports pass-through for IPSEC and PPTP, while the Linksys routers support these and also L2TP pass-through.
Stateful Packet Inspection (SPI) {on,off}: supported by all routers except Linksys WRT54G and Belkin F5D7230.
Wireless security parameters {none, WEP, WPA, WPA2}: all modes supported by all routers, except that the Netgear WGR614v5 does not support WPA2.
DMZ {on, off}: supported by all routers.
UPnP {on, off}: supported by all routers.
NAT type {symmetric, full cone, restricted cone}: only supported by Netgear WGR614v7 and D-Link DIR-635.
Port forwarding for FTP {on, off}: supported by all routers but only used for our FTP experiment.
End-host parameters:
Domain or Workgroup joined
Current user {Administrator, Guest, Everyone, other}
Windows Vista firewall rules {on, off}

Table 4: Parameters varied in our experiments

configurations are possible (e.g., 4800 with the D-Link DIR-635 router). So for each application, we only experimented with a subset of these variations.

To automate the data collection process, we used AutoHotKey [1], a GUI scripting tool. To change configuration settings on the router, we used customized HTTP POST messages. To configure end-hosts, we manually changed the relevant parameters. For every configuration setting, we ran the applications and used simple application-specific heuristics to automatically determine whether the application worked (labeled as "good") or not ("bad"). These heuristics varied based on the application. For example, when the VPN client successfully connects, opening the VPN application's window displays the status of the connection. If the VPN connection was unsuccessful, then the same window shows the user an option to re-initiate the connection. Using AutoHotKey, we captured exactly which kind of message followed our attempt to set up the VPN connection, thereby determining if the application worked or not.

We recreated all of the problems related to VPN clients, file sharing, FTP, and the Xbox shown in Table 1, except for #2 and #6. In addition, our testbed itself presented new problems.

The diversity of configurations that we artificially induce in our testbed facilitates the construction of the application-specific configuration, signature, and change trees. However, it is hard to know how much diversity there would be in practice, in the absence of a large-scale deployment. Nevertheless, in Section 9.6, we demonstrate NetPrints' robustness to noisy data.

Finally, there is no standardized nomenclature for router configuration parameters. The parameter names vary across routers even when the functionality involved is the same. We avoid any manual steps to establish correspondence across routers or segregate information based on router model. If two router models happen to use the same parameter name, NetPrints will recognize and incorporate this in its learning process. Otherwise, it will treat the parameters as separate and unrelated. As standards such as HNAP [2] become prevalent, duplication would be reduced, resulting in more compact and better interpretable configuration trees.

9.2 Microsoft Connection Manager

The Microsoft Connection Manager (CM) [13] is a PPTP-based VPN client. For our evaluation, we used the 7 different routers in turn, varying the settings on each and then using CM to try connecting to an external VPN server. Table 5 shows the number of "good" and "bad" cases recorded with each router through this process.

Figure 2 shows the configuration tree for CM generated by the NetPrints server. Of all the configuration parameters, the algorithm picked `disable_spi`, `pptp_pass`, `filter`, `ethernet.speed`, `ipsec_pass` and `l2tp_pass` as the discerning ones. The numbers at every leaf node are of the form (x/y), where x is the total number of data points that the path from root to that leaf captures, and y is the number of misclassifications on that path.

We can explain the structure of the tree as follows. Only the Netgear routers support the specific `disable_spi` parameter. For these routers, CM works if `disable_spi` is *not* set and does not work if `disable_spi` is set, irrespective of the other parameter settings. On one of the runs involving the Netgear WGR614v5 router, CM failed even though `disable_spi` was not set, explaining the one misclassification on this path.

If `disable_spi` is not applicable, as for the Linksys, D-Link and Belkin routers, the next parameter that the tree learns is `pptp_pass`, which is available only on the Linksys routers. When `pptp_pass`=1, CM works with all three Linksys routers. If `pptp_pass`=0, there are further conditions, depending on the specific Linksys router. Finally, `pptp_pass`=NA for the D-Link and Belkin routers, through which CM works regardless of the settings. The `alg_pptp` parameter on the D-Link DIR-635, which is supposed to control PPTP pass-through, is apparently a no-op.

Next, the tree looks at `filter`, the stateful packet inspection parameter on the Linksys WRT310N and DD-WRT routers. The WRT54G does not support this option, so all configurations with `filter`=NA, i.e., all WRT54G configurations with `pptp_pass`=0, are bad.

The next parameter in the tree, on the `filter`=off branch, is `ethernet.speed`, an interface-specific parameter on the end-host. This is a little counter-intuitive but explainable. The only gigabit ethernet router we used was the WRT310N. Instead of using the model name to distinguish between the WRT310N and the DD-WRT routers, the C4.5 algorithm picked the ethernet speeds instead, since this has the same discriminating power as the model name in this case. This illustrates that learning is data-driven rather than based on intuition. If data were available from more routers supporting gigabit ethernet, we believe that C4.5 would have fallen back to the model name to differentiate among the various routers.

On the WRT310N (`ethernet.speed`=1Gbps), if `filter`=off, CM works irrespective of the other parameters. On the DD-WRT (`ethernet.speed`=100Mbps), CM's success depends on whether the client is placed on the DMZ. In particular, if the client is not on the DMZ, then CM works only if `ipsec_pass`=0 *and* `l2tp_pass`=0. We were unaware of this restriction until NetPrints constructed its configuration tree.

Next, we deployed the NetPrints client on 4 broadband networks using misconfigured Linksys WRT54G and DD-WRT, and Netgear WGR614v5 and WGR614v7 routers. When CM was invoked but the VPN connection failed, the user pressed the "diagnose" button on the NetPrints client. The NetPrints server then used its mutation algorithm to identify remedial configuration changes, which were then conveyed to the client. For the Netgear routers, the fix was to set `disable_spi`=0, whereas for the Linksys routers, it was to set `pptp_pass`=1. The NetPrints client automatically applies these fixes to the router using an HTTP POST to the corresponding Web form on the router.

This case study shows that NetPrints' configuration tree has *automatically* captured application behaviour with a large number of configuration settings across 7 routers and the client host, using a small number of branch points (only 7, in this case) in an intuitive representation. The tree also flagged configuration-related problems that we were unaware of previously.

9.3 Perl-based FTP Client

Users often set up FTP servers within their home networks so that they can have easy access to data on

Application	Netgear WGR614v5		Netgear WGR614v7		Linksys WRT54G		Linksys DD-WRT		Linksys WRT310N		DLink DIR-635		Belkin F5D7230	
	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×
Conn. Manager	25	25	24	24	13	12	34	20	50	40	48	0	25	0
FTP Client	–	–	156	254	309	169	–	–	–	–	67	89	46	26
Xbox	29	20	–	–	33	108	–	–	–	–	–	–	–	–

Table 5: A summary of the number of configuration settings we obtained from each router for VPN, FTP, and Xbox experiments. A "✓" lists the number of good configurations, and a "×" lists bad configurations. Cases where a particular router was not used with an application are marked with "–".

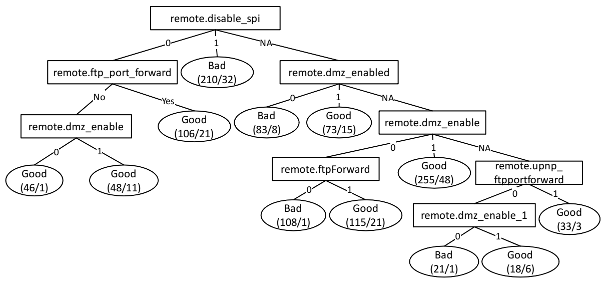


Figure 4: NetPrints configuration tree for the FTP client.

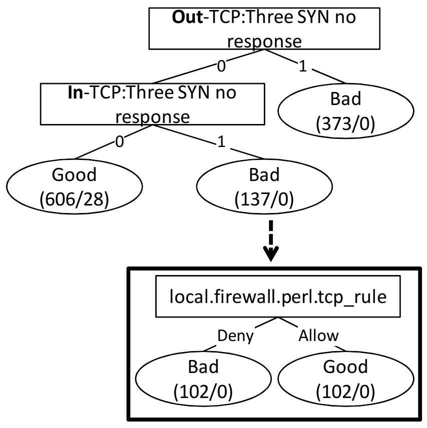

Out-TCP:Three SYN no response
0
1
In-TCP:Three SYN no response
Bad (373/0)
0
1
Good (606/28)
Bad (137/0)
local.firewall.perl.tcp_rule
Deny
Allow
Bad (102/0)
Good (102/0)

Figure 5: NetPrints change tree for the FTP client.

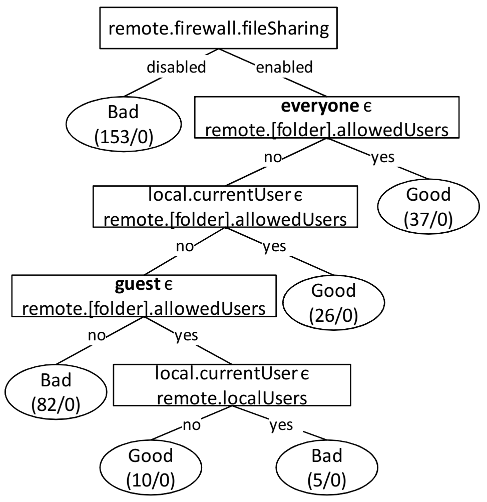

remote.firewall.fileSharing
disabled
enabled
Bad (153/0)
everyone ϵ remote.[folder].allowedUsers
no
yes
local.currentUser ϵ remote.[folder].allowedUsers
Good (37/0)
no
yes
guest ϵ remote.[folder].allowedUsers
Good (26/0)
no
yes
Bad (82/0)
local.currentUser ϵ remote.localUsers
no
yes
Good (10/0)
Bad (5/0)

Figure 6: NetPrints configuration tree for file sharing.

their home computers from remote locations. However, the online discussions forums include several user complaints about the FTP service not running as expected when behind a NAT (e.g., #7 and #8 in Table 1).

To investigate #8, in particular, we evaluated NetPrints when a Perl-based FTP client running on a remote machine tries to connect to an IIS FTP server [3] running on a home network behind a NAT. Besides varying the router configuration settings, we also manually set and reset an application-specific parameter on the FTP client that determined whether the client used passive- or active-mode FTP. This corresponds to the hidden configuration example discussed in Section 7.

Figure 4 shows the NetPrints configuration tree, indicating the various server-side router settings (depending on the router model) needed for FTP to work. Since variable names for the same functionality vary based on the router, the tree has learnt three different variable names to capture the state of the DMZ (`dmz_enable`, `dmz_enabled`, and `dmz_enable_1`).

Note, however, that the misclassification count for most of the leaf nodes in the figure is significant. To understand why, consider the network signature and change trees shown in Figure 5. When the client uses active FTP, all of the server's connection attempts to the client fail, unless a firewall rule on the client host is enabled for allowing incoming TCP connections to the FTP client (this rule is disabled by default). The network signature for this problem has the "**Inbound**:Three SYN no response" feature set, since the client's firewall drops incoming connection attempts from the FTP server. Figure 5 also shows the change tree corresponding to this signature, which essentially says that the above firewall rule should be enabled.

While we used a Perl-based FTP client in this experiment for ease of automation, similar hidden configuration parameters exist in other clients. For example, IE 7.0 has a parameter to "turn off passive FTP connections", which, if set, would result in similar problems and call for similar fixes as those discussed above.

9.4 Windows File Sharing

Home users often use file sharing within the home network. Online forums contain several complaints related to file sharing in Windows Vista, often caused by end-host configuration errors (e.g., #5 and #6 in Table 1).

To investigate these, we set up an experiment where a client host in our home network testbed tried to access a folder on a server host in the same home network. On both the client and the server, we varied the firewall settings, and the domain or workgroup that the machine was joined to. On the server, we varied the access control list (ACL) of users allowed to access the folder, and on the client, we varied the identity of the user who tried to access the folder. In all, we gathered data for 313 different configurations.

Figure 6 shows the configuration tree generated by NetPrints. In a nutshell, the configuration tree tells us that file sharing works if (a) the server-side firewall allows file sharing, and (b.1) either the special user "everyone" is a member of the folder's ACL or the current user on the client is a member of the folder's ACL, or (b.2) the special user "guest" is a member of the server's ACL list and the current user on the client is *not* a local user on the server.

This last point, b.2, is interesting since it suggests that the special user "guest" includes all users *except* the local users on the host machine. This is counter-intuitive since it means that guest users can, depending on the policy, have greater access than local users. We confirmed with experts within Microsoft that this is indeed expected behavior.

9.5 Xbox Live

Xbox Live [20] is a service that allows Xbox users to play multi-player games, chat, and interact over the Internet. One issue was that we could not run the NetPrints client directly on the Xbox since the consumer Xboxes are not user-programmable. For the sake of our experiments, we emulated a NetPrints client on the Xbox by instead running the client on a PC that is able to monitor all of the Xbox's network communication.

For this experiment, we gathered data for the Netgear WGR614v5 and the Linksys WRT54G routers, as indicated in Table 5.

Figure 7 shows the configuration tree generated by NetPrints. NetPrints learned three configuration rules. First, to make the NAT open, the router needs to enable UPnP. Second, Xbox 360 requires the router MTU to be greater than 1300 to enable connectivity to Xbox Live. Third, the Xbox wireless adapter could not connect to a wireless network if the security mode used was WPA2.

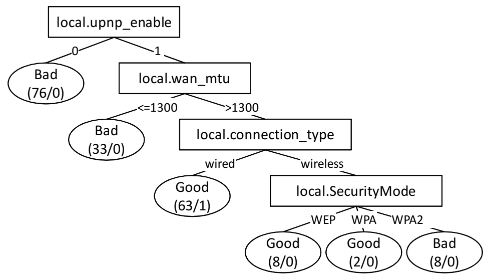


Figure 7: NetPrints configuration tree for Xbox Live.

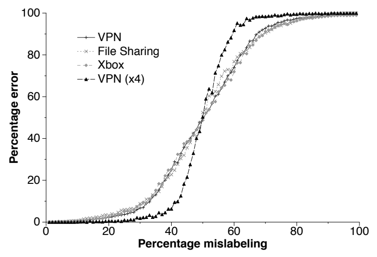


Figure 8: Sensitivity to mislabeled configuration data.

NetPrints' findings correspond to the suggested configuration fixes for #18 and #19 in Table 1, except for the MTU fix. We found out through support sites that Xbox Live requires the MTU to be set to 1365 bytes or larger. However, given that the data from our experiments, which formed the basis for NetPrints' learning, only had the MTU set to one of five values, the best inference we could make was that the MTU should be set to larger than 1300 bytes.

9.6 Robustness Tests

While our experiments have used clean and diverse data, in reality, configurations could be mislabeled and have limited diversity. Hence, we perform experiments to evaluate the robustness of the configuration trees to various conditions not found in our experimental data.

9.6.1 Mislabeled Configurations

In a deployed system, configurations uploaded to the server will not always be labeled correctly. Mislabeled configurations could potentially lead to troubleshooting a problem incorrectly, such as identifying a bad configuration as a good one. To evaluate the sensitivity of our configuration decision trees to mislabeling, we started with a known, correct set of labeled configurations and their associated decision trees. We then chose a random percentage p of those configurations and mislabeled them, flipping their labels from good to bad and vice versa. From this set with mislabeled configurations, we again generated decision trees and compared them with the original trees generated using correct labels.

Figure 8 shows the results of this experiment on the configurations for three applications: VPN (CM), File

Sharing and Xbox. The x-axis shows the percentage of mislabeling of configurations, and the y-axis shows the percentage of configurations incorrectly labeled in the decision tree based upon the mislabeled configurations. Each point represents the average across 100 trials. The VPN, File Sharing, and Xbox curves are similar and therefore difficult to distinguish. The VPN(x4) curve shows the effect of mislabeling for CM when the tree learning used four times as much data as from our testbed.

The results indicate that the applications are fairly resilient to mislabeling. While an insistence on no errors (0%) can only tolerate 2–4% mislabeling, allowing a 1% error (i.e., returning an incorrect configuration fix for up to 1 out of 100 diagnoses) allows tolerating 13–17% mislabeling. When more than 20% of configurations are mislabeled, though, the resulting decision trees overfit substantially, resulting in a high error rate. We also found that the effect of mislabeling diminishes significantly with a larger number of data points. For the VPN(x4) experiment, the tree tolerates 9% mislabeling (0% error) and 26% mislabeling (1% error), making it considerably more tolerant than the tree with the smaller configuration set.

Note that our methodology is not performing cross-validation on the data with training and testing sets. The reason is that we are not using the decision trees as classifiers. In other words, NetPrints does not use decision trees to classify or predict whether a configuration is good or bad — all configurations from the client already have labels ("good" or "bad") associated with them. The mislabeling experiment performs an extrinsic evaluation of the problem in terms of the utility of identifying an appropriate configuration mutation for a diagnosis in the face of incorrect labels.

9.6.2 Reduced Diversity

The configurations from our testbed experiments are roughly uniform in distribution in terms of the settings of the various parameters. In practice, the distribution is likely to be less diverse, with some settings much more prevalent than others (e.g., SPI might be disabled in 90% of configurations). In particular, the default configuration for a device, with an incorrect setting for a parameter, is likely to be prevalent, as is the resulting working configuration after correction.

Does low diversity further change the sensitivity of the decision trees to mislabeling? For each of the VPN, File Sharing and Xbox applications, we chose two configurations representing a default bad configuration and a default good configuration. We then introduced duplicates of those defaults to create low diversity. We varied the percentage of identical configurations from 0–95%, learnt the decision tree, and measured the extent of mislabeling similar to Section 9.6.1. For all of the applications, the effect of mislabeling was the same as with the original distribution of configurations.

10 Discussion

We now discuss a few broad challenges for NetPrints.

10.1 Reducing the Burden of Labeling

As noted in Section 5.2, NetPrints enlists the help of users to perform labeling of configurations (and also of network traffic traces). NetPrints employs several simple ideas to gather rich and accurate labeled data while minimizing the burden on users.

The labeling of "bad" configurations happens implicitly, as a by-product of a user invoking NetPrints for diagnosis when experiencing an application failure. Thus, it is only for having the "good" configurations labeled that the user's help must be enlisted explicitly.

However, prompting the user to label each run of an application as "good" or "bad" would likely be onerous and perhaps also provoke deliberately dishonest behaviour from an irritated user. So, in NetPrints, we only prompt each user for a small fraction of the application runs invoked by that user, with the expectation that, with a minimal burden placed on them, users would likely be honest while labeling. Given the participation of a large number of users, NetPrints is still able to accumulate a large volume of labeled configuration information, even while keeping the burden on any individual user low.

Furthermore, even the occassional prompting of a user is modulated so as to yield useful data with high likelihood. First, since the effective application of learning would require a mix of both "good" and "bad" data, users are prompted more frequently (with the hope of obtaining more data points labeled as "good") when the system is accumulating more "bad" data points because of users invoking NetPrints frequently to diagnose problems. Second, a user is more likely to be prompted when there has been a recent local configuration change. This policy increases the likelihood of novel information being fed into the learning process.

10.2 Preserving Privacy

Privacy is a key concern for NetPrints. Simply excluding privacy-sensitive configuration parameters such as usernames and passwords from the purview of NetPrints is not sufficient. Even the ability to tie back to the origin host (identified, say, by its IP address) configuration data uploaded to the NetPrints server could be problematic. For instance, knowledge of misconfigurations on a host could leave it vulnerable to attacks.

In ongoing work, we are working on a distributed aggregation system aimed at balancing two conflicting goals: enabling nodes to contribute data anonymously while still enforcing tight bounds on the ability of malicious nodes to pollute the aggregated data. Thus, if a majority of nodes is honest, the aggregated data would be mostly accurate. While the details of this aggregation system are out of the scope of the present paper, we believe that NetPrints could directly use such a system.

10.3 Bootstrapping NetPrints

A participatory system such as NetPrints faces interesting challenges in bootstrapping its deployment. There is a chicken-and-egg problem in that users are unlikely to participate unless the system is perceived as being valuable in terms of its ability to diagnose problems, which in turn depends on the contribution of data by the participating users' machines. Even if this dilemma were resolved, there is still the challenge that users might resort to greedy behaviour, installing and running NetPrints only when they need to diagnose a problem and turning it off at other times, thereby starving the system of the data it needs to perform diagnoses effectively.

One could devise incentive mechanisms to encourage user participation. A complementary mechanism, which we are pursuing, is to bootstrap NetPrints using information learned via experiments in a laboratory testbed. This is similar to the methodology used for the evaluation presented in Section 9. While the richness of the testbed data would have a direct bearing on NetPrints' learning and hence its ability to diagnose problems, such an approach could help bootstrap NetPrints to the point where users perceive enough value to start participating.

11 Conclusion

We have described the design and implementation of NetPrints, a system to automatically troubleshoot home networking problems caused by misconfigurations. NetPrints uses decision tree-based learning on labeled configuration information and traffic features from a population of clients to build a shared repository of knowledge on a per-application basis. We report experimental results for a few applications in a laboratory testbed and a small-scale deployment. Our ongoing work focuses on scaling up the deployment and addressing privacy issues.

Acknowledgments

The authors would like to thank Monojit Choudhury for fruitful discussions on machine learning techniques. We would also like to thank our shepherd Kobus van der Merwe and the anonymous reviewers for their time and insightful comments regarding NetPrints and this paper.

References

[1] GUI scripting using the AutoHotKey tool. http://www.autohotkey.com.

[2] Home Network Administration Protocol. http://hnap.org.

[3] Microsoft Internet Information Services (IIS). http://www.iis.net.

[4] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards Highly Reliable Enterprise Network Services Via Inference of Multi-level Dependencies. In *SIGCOMM*, 2007.

[5] P. Bahl, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. DAIR: A Framework for Managing Enterprise Wireless Networks Using Desktop Infrastructure. In *HotNets*, 2005.

[6] R. Chandra, V. N. Padmanabhan, and M. Zhang. WiFiProfiler: Cooperative Diagnosis in Wireless LANs. In *MobiSys*, 2006.

[7] Y. Cheng, P. B. John Bellardo, A. C. Snoeren, G. M. Voelker, and S. Savage. Jigsaw: Solving the Puzzle of Enterprise 802.11 Analysis. In *SIGCOMM*, 2006.

[8] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. In *OSDI*, 2004.

[9] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In *SOSP*, 2005.

[10] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, 2005.

[11] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren. IP Fault Localization via Risk Modeling. In *NSDI*, 2005.

[12] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson. User-level Internet Path Diagnosis. In *SOSP*, 2003.

[13] Microsoft Connection Manager. http://support.microsoft.com/kb/221119.

[14] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kauffman, 1993.

[15] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.

[16] Universal Plug and Play Internet Gateway Device Specification. http://www.upnp.org/standardizeddcps/igd.asp.

[17] P. E. Utgoff. Incremental Induction of Decision Trees. *Machine Learning*, 4:161–186, 1989.

[18] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *OSDI*, 2004.

[19] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H. J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *LISA*, 2003.

[20] The Xbox Live Service. http://www.xbox.com/en-us/live/.

[21] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated Known Problem Diagnosis with Event Traces. In *EuroSys*, 2006.

Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage

Yuvraj Agarwal[‡], *Steve Hodges*[†], *Ranveer Chandra*[†], *James Scott*[†], *Paramvir Bahl*[†], *Rajesh Gupta*[‡]

[†]*Microsoft Research,* [‡]*University of California, San Diego*
[‡]*yuvraj@cs.ucsd.edu,* [†]*{shodges, ranveer, jws, bahl}@microsoft.com,* [‡]*gupta@cs.ucsd.edu*

Abstract

Reducing the energy consumption of PCs is becoming increasingly important with rising energy costs and environmental concerns. Sleep states such as S3 (suspend to RAM) save energy, but are often not appropriate because ongoing networking tasks, such as accepting remote desktop logins or performing background file transfers, must be supported. In this paper we present *Somniloquy*, an architecture that augments network interfaces to allow PCs in S3 to be responsive to network traffic. We show that many applications, such as remote desktop and VoIP, can be supported without application-specific code in the augmented network interface by using application-level wakeup triggers. A further class of applications, such as instant messaging and peer-to-peer file sharing, can be supported with modest processing and memory resources in the network interface. Experiments using our prototype Somniloquy implementation, a USB-based network interface, demonstrates energy savings of 60% to 80% in most commonly occuring scenarios. This translates to significant cost savings for PC users.

1 Introduction

Many personal computers (PCs) remain switched on for much or all of the time, even when a user is not present [23], despite the existence of low power modes, such as sleep or suspend-to-RAM (ACPI state S3) and hibernate (ACPI state S4) [1]. The resulting electricity usage wastes money and has a negative impact on the environment.

PCs are left on for a variety of reasons (see Section 2), including ensuring remote access to local files, maintaining the reachability of users via incoming email, instant messaging (IM) or voice-over-IP (VoIP) clients, file sharing and content distribution, and so on. Unfortunately, these are all incompatible with current power-saving schemes such as S3 and S4, in which the PC does not respond to remote network events. Existing solutions for sleep-mode responsiveness such as Wake-On-LAN (WoL) [18] have not proven successful "in the wild" for a number of reasons, such as the need to modify application servers or configure network hardware. A few initial proposals suggest the use of network proxies [4, 7, 11] to perform lightweight protocol functionality, such as responding to ARPs. However, such a system too requires significant modifications to the network infrastructure, and to the best of our knowledge such a prototype has not been described in published form (see Section 6 for a full discussion).

In this paper, we present a system, called Somniloquy[1], that supports continuous operation of many network-facing applications, even while a PC is asleep. Somniloquy provides functionality that is not present in existing wake-up systems. In particular, it allows a PC to sleep while continuing to run some applications, such as BitTorrent and large web downloads, in the background. In existing systems, these applications would stop when the PC sleeps.

Somniloquy achieves the above functionality by embedding a low power secondary processor in the PC's network interface. This processor runs an embedded operating system and impersonates the sleeping PC to other hosts on the network. Many applications can be supported, either with or without application-specific code "stubs" on the secondary processor. Applications simply requiring the PC to be woken up on an event can be supported without stubs, while other applications require stubs but in return support greater levels of functionality during the sleep state.

We have prototyped Somniloquy using a USB-based low power network interface. Our system works for desktops and laptops, over wired and wireless networks, and is incrementally deployable on systems with an existing network interface. It does not require any changes to the operating system, to network hardware (e.g. routers), or to remote application servers. We have implemented support for applications including remote desktop access, SSH, telnet, VoIP, IM, web downloads

[1]somniloquy: the act or habit of talking in one's sleep.

and BitTorrent. Our system can also be extended to support other applications. We have evaluated Somniloquy in various settings, and in our testbed (Section 5) a PC in Somniloquy mode consumes 11x to 24x less power than a PC in idle state. For commonly occurring scenarios this translates to energy savings of 60% to 80%.

We make the following contributions in this paper:

- We present a new architecture to significantly reduce the energy consumption of a PC while maintaining network presence. This is accomplished without changes in the network infrastructure.
- We show that several applications — BitTorrent, web downloads, IM, remote desktop, etc. — can consume much less energy. This is achieved without modifying the remote application servers.
- We present and empirically validate a model to predict the energy savings of Somniloquy for various applications.
- We demonstrate the feasibility of Somniloquy via a prototype using commodity hardware. This prototype is incrementally deployable, and saves significant energy in a number of scenarios.

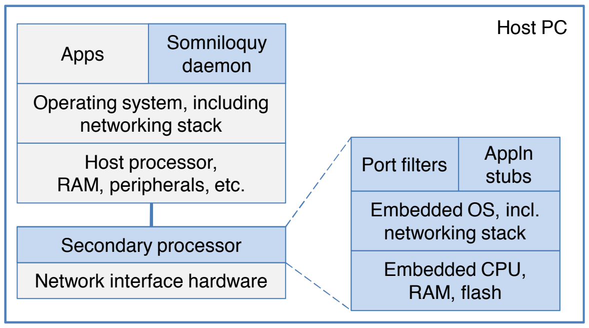
2 Motivation

Prior studies have shown that that users often leave their computer powered on, even when they are largely idle [4]. A study by Roberson et. al. [23] shows that in offices, 67% of desktop PCs remain powered on outside work hours, and only 4% use sleep mode. In home environments, Roth et. al. [24] show that average residential computer is on 34% of the time, but is not being actively used for more than half the time.

To uncover the reasons why people do not use sleep mode, we conducted an informal survey. We passed it among our contacts who in turn circulated it further. We had 107 respondents from various parts of the world, of which 58 worked in the IT sector. 30% of the respondents left at least one machine at home on all of the time, and 75% of the respondents left at least one work machine on even when no one was using it.

Among the people who left their home machine powered on, 29% did so for remote access, 45% for quick availability and 57% for applications running in the background, of which file sharing/downloading (40%) and IM/e-mail (37%) were most popular. In the office environment, 52% of respondents left their machines on for remote access, and 35% did so to support applications running in the background, of which e-mail and IM were most popular (47%).

Although this survey should not be regarded as representative of all users, and is not statistically significant, it does highlight two important points. First, a number of PCs don't go to sleep even when they are unused. Second, significant energy savings can be achieved if only a few applications — remote reachability, file sharing, file downloads, instant messaging, e-mail — can be handled when the PC is asleep.



Host PC
Apps
Somniloquy daemon
Operating system, including networking stack
Host processor, RAM, peripherals, etc.
Secondary processor
Network interface hardware
Port filters
Appln stubs
Embedded OS, incl. networking stack
Embedded CPU, RAM, flash

Figure 1: *Somniloquy augments the PC network interface with a low power secondary processor that runs an embedded OS and networking stack, network port filters and lightweight versions of certain applications (stubs). Shading indicates elements introduced by Somniloquy.*

3 The Somniloquy Architecture

Our primary aims during the development of Somniloquy were:

- to allow an unattended PC to be in low power S3 state while still being available and active for network-facing applications as if the PC were fully on;
- to do so without changing the user experience of the PC or requiring modification to the network infrastructure or remote application servers.

We accomplish these goals by augmenting the PC's network interface hardware with an always-on, low power embedded CPU, as shown in Figure 1. This *secondary processor* has a relatively small amount of memory and flash storage [2] which consumes much less power than if it were sharing the larger disk and memory of the host processor. It runs an embedded operating system with a full TCP/IP networking stack, such as embedded Linux or Windows CE. The flash storage is used as a temporary buffer to store data before the data is transferred in a larger chunk to the PC. A larger flash on the secondary processor allows the PC to sleep longer (Section 3.2. This architecture has a couple of useful properties. First, it does not require any changes to the host operating system, and second, it can be incrementally deployed on existing PCs using a peripheral network interface (Section 4).

[2]Our prototype had 64 MB DRAM and 2 GB of flash.

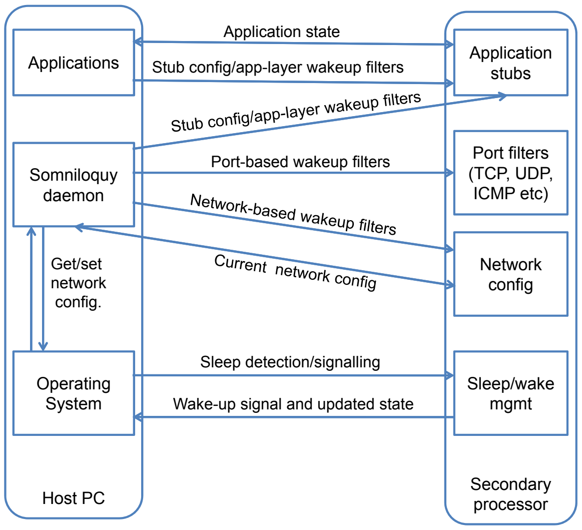
The software components of Somniloquy and their interactions are illustrated in Figure 2. The high-level operation of Somniloquy is as follows: When the host PC is powered on, the secondary processor does nothing; the network stack on the host processor communicates directly with the network interface hardware. When the PC initiates sleep, the Somniloquy daemon on the host processor captures the sleep event, and transfers the network state to the secondary processor. This state includes the ARP table entries, IP address, DHCP lease details, and associated SSID for wireless networks i.e. MAC- and IP-layer information. It also includes details of what events the host should be woken on, and application-specific details such as ongoing file downloads that should continue during sleep. Following the transfer of this information to the secondary processor, the host PC enters sleep.

Although the host processor is asleep, power to the network interface and the secondary processor is maintained [1]. To maintain transparent reachability to the host while it is asleep, the secondary processor impersonates the host by using the same MAC and IP addresses, host name, DHCP details, and for wireless, the same SSID. It also handles traffic at the link and network layers, such as ARP requests and pings – thereby maintaining basic presence on the network. New incoming connection requests for the host processor are now received and handled by the network stack running on the secondary processor. In this way the PC's transition into sleep is transparent to remote hosts on the network.

To ensure that the host PC is reachable by various applications, a process on the secondary processor monitors incoming packets. This process watches for patterns, such as requests on specific port numbers, which should trigger wake-up of the host processor. Although, this simple architecture [4, 7, 11] supports several applications with minimal complexity, Somniloquy can get much greater energy savings for some applications by not waking up the host processor for simple tasks, for example, to send instant messenger presence updates. To perform these tasks on the secondary processor, we require the application writer to add a small amount of application specific code ("stubs") on the host and secondary processor. In the rest of this section we describe in more detail how we handle various applications – with and without application stubs.

3.1 Somniloquy without Application Stubs

The Somniloquy daemon on the host processor specifies packet filters, i.e. patterns on incoming packets, on which the secondary processor should wake up the host processor from sleep state. The Somniloquy daemon creates filters at various layers of the network stack. At the link layer and network layer, the secondary processor can be told to wake the computer when it detects a particular packet, analogously to the magic packets used by Wake on LAN, though not requiring the MAC address to be known by the remote host (see further discussion in Section 6). Trigger conditions at the transport layer may also be specified, for example, wake on TCP port 23 for telnet requests. Similarly, Somniloquy also supports wake-ups on patterns in the application payload.



Applications
Application state
Stub config/app-layer wakeup filters
Application stubs
Stub config/app-layer wakeup filters
Somniloquy daemon
Port-based wakeup filters
Port filters (TCP, UDP, ICMP etc)
Network-based wakeup filters
Current network config
Network config
Get/set network config.
Operating System
Sleep detection/signalling
Wake-up signal and updated state
Sleep/wake mgmt
Host PC
Secondary processor

Figure 2: *Somniloquy software components on the host PC and the secondary processor, and their interactions.*

Although the host PC will wake up within a few seconds, it will not receive the packet(s) that triggered the wake-up. One way to solve this problem is to buffer the packet on the secondary processor and replay it on the network stack of the host processor once it has woken up. However, since the time to wake up is just a few seconds, most sources can be relied upon to retry the connection request. For example, any protocol using TCP as the transport layer will automatically retransmit the initial SYN packet. Even UDP-based applications that are designed for Internet use are designed to cope with packet loss using automatic retransmissions.

This simple packet filter based approach to triggering wake-ups has the advantage that application-specific code does not need to be executed on the secondary processor. Nonetheless, it is sufficient to support many applications that get triggered on remote connection requests, such as remote file access, remote desktop access, telnet and ssh requests to name a few.

3.2 Application-specific Extensions

Several applications maintain active state on the PC even when it is idle, and hence prevent a PC from going to sleep. For example, a movie download client on a home

PC (e.g. from Netflix) will require the host PC to be awake for a few hours while downloading the movie. An instant messenger (IM) client will require the PC to be on in order for the user to stay "online" (reachable) to their contacts.

Somniloquy provides a way for these applications to consume significantly less power. By performing lightweight operations on the secondary processor, it can opportunistically put the host processor to sleep. For example, the secondary processor can send and receive presence updates to/from the IM server while the host processor is asleep. During a large download, the secondary processor can download portions of the file, putting the host processor to sleep in the meantime.

The key to supporting these applications is the use of stubs that run on the host and the secondary processor. We have implemented stubs for three popular applications – IM (MSN, AOL, ICQ), BitTorrent, and web download. Here, we will describe the general guidelines for writing these stubs, and describe the specific implementations for the three applications in Section 4.

Writing application stubs: When designing an application stub, the first step is to understand the subset of the application's functionality that needs to run when the PC is asleep. This is implemented as a stub on the secondary processor. For example, for an IM stub, the functionality to send and receive presence updates is essential to maintain IM reachability. However, the stub need not include any UI-related code – such as opening a chat window.

We note that it is not feasible for the stub to reuse the entire original application code from the host PC. The application code might depend on drivers (display, disk, etc.) that are absent on the secondary processor. Furthermore, running the entire application might overload the secondary processor. Therefore, only the essential components of the application are implemented as part of the application stub.

Another step in designing application stubs is to decide when to wake up the host processor. Triggers can be user-defined, for example waking up on an incoming call from a specific IM contact. Triggers may also occur when the secondary's processor's resources are insufficient, for example when the flash is full or more CPU resources are needed. In all of these cases, the stub wakes up the host processor.

To interface with the application on the host PC and the Somniloquy daemon, the application stub needs to have a component on the host processor. This component registers two callback functions with the Somniloquy daemon — one that is called just before the PC goes to sleep and the other just after it has woken up. The first function transfers the application state to the stub on the secondary processor, and also sets the trigger conditions on which to wake the host processor. These values depend on the application being handled by the stub. The second callback function, which is called when the host resumes from sleep, checks the event that caused the wakeup — whether it was caused by a trigger condition on the secondary processor or due to user activity. It handles these events differently. If the wakeup was caused by user activity, the stub transfers state from the secondary processor, and disables it. However, if the wakeup was caused by a trigger condition on the secondary processor, the application stub handles it as defined by the user. For example, for an incoming VoIP call, the stub engages the incoming call functionality of the VoIP application.

Having determined what functionality needs to be supported by the application stub and host-based callbacks, and what state must pass between them, the final step is to implement this. We have used two manual approaches to doing this. For the download stub, we built all the functionality ourselves based on detailed knowledge of the application protocols, and for the BitTorrent and IM stubs, we trimmed down existing application code to reduce memory and CPU footprint. An alternative could be to automatically learn protocol behavior to build these application stubs. However, we believe that this is an extremely difficult problem. There are parts of the application that are difficult to infer, and any inaccuracy in the application stub will make it unusable. For example, knowledge of how BitTorrent hashes the file blocks is necessary for the stub to successfully share a file with peers. We are unaware of any automatic tool that can learn such application behavior. Therefore, we believe that the best (although perhaps not the most elegant) approach to building these stubs is to modify application source code and remove functionality that is not required by the secondary processor. In the future, with a greater incentive to save energy, we expect that application developers will compete for energy consumption, and hence provide stubs for their applications using the guidelines described in this section.

We realize that partial application stubs might be created using tools such as the Generic Application-Level Protocol Analyzer [6] and Discoverer [8], which automatically learn the behavior and message formats for a range of protocols. As part of future work, we plan to explore how the knowledge of the protocol can be augmented with application-specific behavior to ease the development of application stubs.

When to use application stubs? Not all applications are conducive to low-power operation via application stubs. A CPU intensive application, such as a compilation job, will be very slow on the secondary processor since it has a less powerful CPU and low memory. Similarly, an I/O intensive application, such as a disk indexer, will need to read the disk very often and will therefore need the PC to be awake. Download and file sharing applications are an interesting exception, because portions of a file can be transferred by the secondary processor whilst the host sleeps. We will discuss this approach in more detail in Section 4.4.

Even for an application stub that saves energy for a given application, it is not always useful to offload the application to the secondary processor when the host PC is going to sleep. Several other applications may also want to run their application stubs on the secondary processor. This might overload the CPU of the (weaker, low power) secondary processor. In this case, it might be beneficial to keep the host PC awake.

One way to solve this problem is to modify the Somniloquy daemon to predict the CPU utilization of the stubs for all applications that are willing to be offloaded to the secondary processor. However, making this prediction is extremely difficult. There might be little correlation between the CPU utilization of the application on the host PC, and the stub on the secondary processor, because of different processor architectures, and varying application demands. Instead, we take a systems approach. We monitor the CPU utilization of the secondary processor; if it remains at more than 90% continuously($>$30 seconds), we wake up the PC, and resume all applications on the host processor. If the CPU utilization of these applications decreases by more than 10% on the host processor, we repeat the same procedure — offload to the secondary processor and stay there if CPU utilization is less than 90%. In our Somniloquy deployment the need to move applications arose when running multiple application stubs on the secondary processor, such as two concurrent 8 Mbps web downloads and two concurrent BitTorrent downloads of Section 5.3.2.

Incremental Deployment: We realize that Somniloquy may never be universally deployed, and that getting software vendors to try for incremental deployment requires a low-effort mechanism to ensure that their Somniloquy-enhanced software is compatible with machines and platforms that do not have Somniloquy support. The Somniloquy daemon queries the OS to determine the presence of a secondary processor, and the supported application stubs. Applications then need to query the Somniloquy daemon, and invoke the application stubs only if the OS supports Somniloquy, and the corresponding stubs are implemented on the secondary processor.

3.3 Quantifying Energy Savings

The amount of energy saved through adoption of Somniloquy is quite easy to predict; it depends on the relative power consumption of the awake and sleep states, and the proportion of time that a machine can be kept asleep when it would previously have been awake. For applications without stubs, this proportion is largely dependent on the actions of a remote user - how frequently a remote ssh session is initiated for example, and for how long. On the other hand, for applications with stubs the secondary processor may regularly wake up the host to perform some task or other. We quantify the energy savings for an application with different wake-up intervals in Section 5.4.4.

More formally, suppose the host is woken up once every T_{sleep} seconds, whereupon it stays awake for T_{awake} seconds. T_{awake} includes the time it takes to transfer data between the PC and the secondary processor. Also assume that d is sum of the time to wake up the host plus the time to transition to sleep. Suppose:

- P_a is the power consumption of the PC when it is awake (in W)
- P_s is power consumed in sleep mode (in W), and
- P_e is power consumed by the secondary (embedded) processor (in W)

The energy (E) consumed during Somniloquy operation is given by:

$$\begin{aligned} E_{somniloquy} &= E_{PCinSleepMode} + E_{PCinAwakeMode} \\ &\quad + E_{SecondaryProcessor} \\ &= T_{sleep} * P_s + (T_{awake} + d) * P_a \\ &\quad + (T_{awake} + d + T_{sleep}) * P_e \; Joules \end{aligned}$$

In the absence of Somniloquy, the amount of energy consumed by the host PC in the same time is $E_{host} = P_a * (T_{awake} + T_{sleep})$ Joules. Therefore, the ratio of energy consumed by Somniloquy compared to the host PC being always on is given by:

$$\frac{E_{somniloquy}}{E_{host}} = \frac{T_{sleep}*(P_e+P_s)+T_{awake}*(P_a+P_e)+d*(P_a+P_s)}{P_a*(T_{awake}+T_{sleep})}$$

Typically, as we show in Section 5, P_e and P_s are two orders of magnitude less than P_a for a desktop computer, and d is around 10 seconds (to wake up the host, and put it back to sleep). Therefore, for most energy savings, we would want T_{awake} to be much less than T_{sleep}, i.e. if $T_{awake} \ll T_{sleep}$, then the ratio $E_{somniloquy}/E_{host}$ is approximately $(P_e + P_s)/P_a$. We will present the approximate energy savings for different applications in Section 4.

Of course, Somniloquy could save more energy by disabling the secondary processor when the PC is awake. This would require the PC to enable the secondary processor before going to sleep, and disable it when the PC has woken up. We were unable to fully implement this functionality in our prototype, but we expect this to be a minor fix in a production system.

3.4 Discussion

Security: A common requirement of corporate IT departments is that all PCs should be up to date with the latest OS and application patches. Somniloquy can ensure that this constraint is met even when PCs are asleep. This is achieved using a port-based trigger to wake up the host PC when the SMS (Systems Management Server) contacts the host PC to install updates.

Somniloquy ensures that the secondary processor is secure by patching its OS whenever security updates become available. Also, it prevents attackers from replacing the secondary processor by requiring that it be a physically part of the PC (as part of the network interface). In some cases however, the functionality that Somniloquy provides could be misused to conduct attacks that spuriously wake up the PC and waste energy. This kind of denial-of-service attack would be particularly effective for mobile devices where a drained battery might result. One way to address this issue is to disable port triggers, and instead exclusively use application stubs which ensure that only authenticated remote hosts are allowed to trigger wakeup.

Another concern is that application stubs, and hence the use of extra code, increases the PC's attack surface. To mitigate the impact of this vulnerability we use a few techniques. First, the secondary processor only listens on ports that have been opened by applications on the host PC. Second, we require the PC and the secondary processor to be on the same administrative domain.

We also note that modern processors have additional security features built in, for example an execute-disable bit, used by some applications to prevent executing arbitrary code and preventing buffer overflows. We realize that a low power processor may not currently support this advanced functionality, although we expect that in the future low-power chips will also be available with these features.

Alternative Design: With the increasing prevalence of multi-core PCs, one idea to alleviate the need for the additional secondary processor introduced by Somniloquy would be to use one of the cores of the host CPU instead. Running just one core at the lowest possible clock frequency would minimize energy consumption and obviate the need for a separate low power processor in the NIC.

However, it turns out that such an approach is not useful without significant modification to today's PC architecture. Our measurements (see Section 5.1) show that the power consumption of a multi-core PC with only one core active, running at the lowest permissible clock speed is still approximately 50 times that of our low power secondary processor, even with all other peripherals in their lowest power modes – e.g. disk spun down. This is because of the lack of truly fine-grained power control of PC components such as the Northbridge, Southbridge, memory buses, parts of the storage hierarchy and various peripherals. Even if fine-grained control were available, the base power consumption of individual components (NIC, hard drive) is significant (see Table 2). One way to reduce this base power draw would be to have a separate and relatively simple core with a small amount of associated memory running from a separate power domain so that it can function without powering on other components. Such an architecture is very similar to Somniloquy, and most of our design principles can easily be adopted.

4 Prototype Implementation

We have prototyped Somniloquy using *gumstix*, a low power modular embedded processor platform manufactured by Gumstix Inc that support a wide variety of peripherals.

4.1 Hardware and Software Overview

An important goal when prototyping Somniloquy was to have it work with existing unmodified desktops and laptops, and for both wired and wireless networks. Furthermore, we required the platform to be low power, have a small form factor, and be well supported for development. The gumstix platform served all these design requirements well. The specific components we use for Somniloquy include a connex-200xm processor board, an etherstix network interface card (NIC) (for wired Ethernet), a wifistix NIC (for Wi-Fi), and a thumbstix combined USB interface/breakout board. The connex-200xm employs a low power 200 MHz PXA255 XScale processor, with 16 MB of non-volatile flash and 64 MB of RAM. The etherstix provides a 10/100BaseT wired Ethernet interface plus an SD memory slot to which we have attached a 2GB SD card. The thumbstix provides a USB connector, serial connections and general purpose input and output (GPIO) connections from the XScale.

To enable Somniloquy we needed mechanisms to wake-up the host PC, and also to detect its state (awake or in S3). To achieve this we added a custom designed circuit board that incorporates a single chip — the FT232RL from FTDI. The FT232RL is a USB-to-Serial converter chip supporting functionality such as sending a resume signal to the host and detecting the state of the host, both over the USB bus. This board is attached to the computer via a second USB port and to the thumbstix module (and thence to the XScale processor) via a two-wire serial (RS232) interface plus two GPIO lines. One GPIO line is connected to the FT232RL's 'ring indicator' input to wake up the computer. The second GPIO

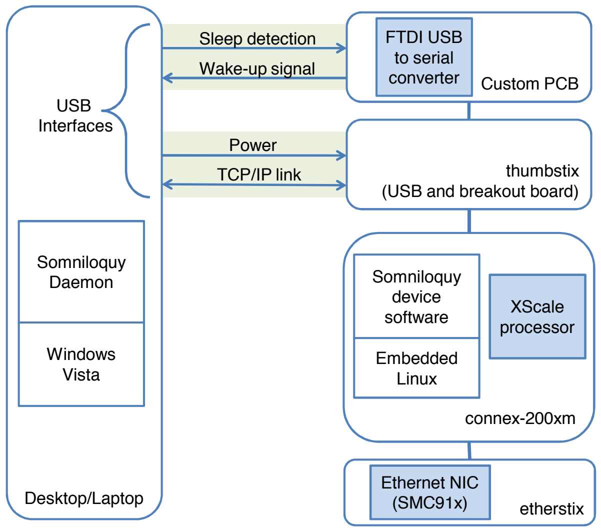


Figure 3: *Block diagram of the Somniloquy prototype system - Wired-1NIC version. The figure shows various components of the gumstix and the USB interfaces to the host laptop.*

line is connected to the FT232RL's 'sleep' output which can be polled by the gumstix to detect whether the host PC is active or in S3.

As mentioned above (and shown in Figure 3), the computer is connected to the secondary processor via two USB connections. One of these provides power and two-way communications between the two processors. It is configured to appear as a point-to-point network interface ("USBNet"), over which the gumstix and the host computer communicate using TCP/IP. The second USB interface provides sleep and wake-up signaling, and a serial port for debugging purposes. The use of two USB interfaces is not a fundamental requirement, it is simply for ease of prototyping.

Since we use standard USB ports for interfacing with the host and for sleep signaling, our prototype works on any recent desktop or laptop that supports USB. We run an embedded distribution of Linux on the gumstix that supports a full TCP/IP stack, DHCP, configurable routing tables, a configurable firewall, SSH and serial port communication. This provides a flexible prototyping platform for Somniloquy with very low power operation.

We have implemented the Somniloquy host software on Windows Vista. The Somniloquy daemon detects transition to S3 sleep state, and before this is allowed to occur we transfer the network state (MAC address, IP address, and in the case of the wireless prototype, the SSID of the AP) and other information about the wakeup triggers as discussed in Section 3.

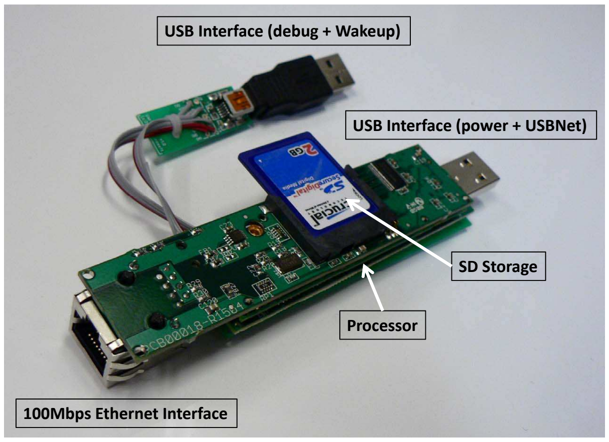

USB Interface (debug + Wakeup)
USB Interface (power + USBNet)
SD Storage
Processor
100Mbps Ethernet Interface

Figure 4: *Photograph of the gumstix-based Somniloquy prototype - Wired-1NIC version.*

4.2 Three different prototypes

We have prototyped three different Somniloquy designs to explore different aspects of operation. The first uses the gumstix as an augmented Ethernet interface, as described in Section 3. However, in our prototype this has some performance limitations so we have also implemented a second design which uses the gumstix in co-operation with an existing high-speed Ethernet interface. Finally, we have a Wi-Fi version. All three prototypes are described in further detail below:

Augmented Network Interface: We call this implementation the *Wired-1NIC* version. The architecture is shown in Figure 3, with a photograph of the prototype shown in Figure 4. In this prototype, we disable the NIC of the host, and configure the PC to use the USBNet interface (USB connection between the gumstix and the host) as its only NIC. The gumstix is connected to the network using its Ethernet connection. To enable the host PC to be on the network, we set up a transparent layer-2 software bridge between the USBnet interface to the host and the Ethernet interface of the gumstix. This bridge is active when the host is awake. When the host transitions to sleep, the gumstix disables the bridge, and resets the MAC address of its Ethernet interface to that of the USBNet interface of the host. The gumstix thus appears to the rest of the network as the host itself, since it has the same network parameters (IP, MAC address). When the host wakes up, the gumstix resets its MAC address to its original value and starts bridging traffic to the host again.

Although our *Wired-1NIC* prototype hardware supports a 100 Mbps Ethernet interface, we are limited to a throughput of 5 Mbps due to the bandwidth supported by the USBNet interface driver. There is also a slight overhead of bridging traffic on the gumstix. Although this limits bandwidth to the host significantly in our prototype, we note that in a final integrated version, this overhead of bridging can be avoided by allowing both the host and the low power secondary processor to access the NIC directly.

Using Existing Network Interface: Somniloquy can coexist with an existing NIC. On such systems, the overhead of bridging is avoided by using the existing Ethernet interface on the host PC for data transfer when it is awake, with the gumstix using its own Ethernet interface (while still impersonating the host PC) when the host is asleep. We have built this version where the gumstix does not perform Layer-2 bridging, and call it the *Wired-2NIC* prototype.

Using Wi-Fi: We have also implemented a wireless version of Somniloquy. We were unable to implement a one-NIC version since the Marvell 88W8385 802.11 b/g chipset present on the wifistix does not currently support layer 2 bridging. We have however implemented a *Wireless-2NIC* version.

4.3 Applications Without Stubs

We have implemented a flexible packet filter on the gumstix using the BSD raw socket interface to support applications that do not require stubs, e.g. RDP, SSH, telnet and SMB connections. Every application in this class provides a regular expression matched against incoming packets to decide whether to trigger host wakeup. For example, handling incoming remote desktop requests requires the host to be woken up when the gumstix receives a TCP packet with destination port 3389.

We note that waking up the host computer is not enough; the incoming connection request must somehow be conveyed to the host. We accomplish this by using the `iptables` firewall on the gumstix to filter any response to TCP or UDP packets that the gumstix does not handle itself. Thus trigger packets are not acknowledged by the gumstix and the remote client sends retries. After the host has resumed, one of the retries will reach it (since it is still using the same IP and MAC addresses) and it will respond directly. Using port-based filtering, we have implemented wake-up triggers for four applications: remote desktop requests (RDP), remote secure shell (SSH), file access requests (SMB), and Voice over IP calls (SIP/VoIP).

4.4 Applications Using Stubs

To demonstrate how modest application stubs can enable significant sleep-mode operation in Somniloquy, we have also implemented application stubs for three applications that were popular in our informal survey: background web download, peer to peer content distribution using BitTorrent, and instant messaging. For all these applications, we did not have to modify the operating system or the existing applications on the PC, which were only available to us in binaries. To capture the state of the application for the respective stub, we wrote wrappers around the binaries.

Background Web Downloads: We developed the web download stub for `wget` which works as follows: When the host PC transitions to sleep, the status of active downloads is sent to the stub running on the gumstix. The status includes the download URL, the offset of how much download has taken place, the buffer space available, and the credentials (if required for the download). Most popular web servers (e.g. IIS and Apache) allow these byte ranges to be specified using the HTTP 'Accept-Ranges' primitives [22]. The web download stub then resumes the downloads from the respective offsets of the files, and stores the data on the flash storage of the gumstix. If the flash memory fills up before the downloads complete, the stub wakes up the host PC and transfers the downloaded files from flash storage to the host PC, thereby freeing up space. The host PC then goes back to sleep while the stub continues the downloads. At the end of a download, the gumstix wakes up the host PC, and transfers the remaining part of the file.

The download stub consumes significantly less energy to download a file than keeping the PC awake to download it. The overhead is a slight increase in latency. We can quantify the savings and overhead using the model described in Section 3.3. If flash storage is F MB and the download bandwidth is B MBps, then the host PC is woken up every F/B seconds, and it is awake for F/T seconds, where T is the transfer rate between the host and the gumstix. Therefore, using the formula in Section 3.3, Somniloquy gives most energy savings at low B and high T. We empirically validate this observation in Section 5.4.4. When T is of the same order as B, Somniloquy might not save much energy. This can happen if the NIC supports very high rates (e.g. 1 Gbps), while the secondary processor can only support lower data rates (up to 100 Mbps) or if the transfer rate T is limited. However, we anticipate the download stub to be primarily used in scenarios where the download speeds are limited by the last mile connection of at most a few tens of Mbps – here, this stub is nearly always beneficial.

BitTorrent: For the BitTorrent stub we customized a console-based client, *ctorrent*, to run on the gumstix with a low CPU utilization and memory footprint. Prior to suspending to S3, the host computer transfers the '.torrent' file and the portion of the file that has already been downloaded to the gumstix. The BitTorrent stub on the gumstix then resumes download of the torrent file and stores it temporarily on the SD flash memory of the gumstix. When the download completes, the stub wakes up the host and transfers the file.

When only downloading content, the energy saved by using this stub is similar to that of the web download stub, i.e., frequency of waking up the PC and the duration for which it is woken up depends on the download bandwidth B, the transfer speed T and the flash size F. However, when uploading/sharing (which is key to altruistic P2P applications), the energy savings are much more. The same file chunk can be uploaded to many peers, and hence the PC can sleep for much longer – implying more energy savings using the formula in Section 3.3.

Instant Messaging: For the IM stub, we used a console-only IM client called *finch* that supports many IM protocols such as MSN, AOL, ICQ, etc. On the PC, we used the corresponding GUI version of the IM client. To ensure our goal of a low memory and CPU footprint we customized finch to include only the features salient to our aim of waking up the host processor when an incoming chat message arrives. This only requires authentication, presence updates and notifications; we disabled other functionality. The host processor transfers over the authentication credentials for relevant IM accounts before going to S3. The gumstix then logs into the relevant IM servers, and when an incoming message arrives it triggers wakeup. The energy saved by the IM stub is thus similar to applications that are handled using packet filters (e.g. SSH/RDP), where the duration for which a host can sleep depends on the frequency of occurrence of wake-up triggers.

5 System Evaluation

We present the benefits of Somniloquy in four steps. First, we show that gumstix consumes much less power than a PC by profiling standalone desktops, laptops and the gumstix in different power states. Second, we measure the energy saved (and latency introduced) by Somniloquy when used on an "idle" host processor. Third, we show how Somniloquy affects the performance of various applications, with and without application stubs. Finally, we quantify Somniloquy's energy savings — monetary and environmental cost for an enterprise and battery lifetime increase for laptops.

Methodology: To measure the power consumption of laptops and desktop PCs, we used a commercially available mains power meter, *Watts-Up* [3]. To measure the power consumption of the standalone gumstix, we built a USB extension cable with a 100 mΩ 0.1% sense resistor, which was inserted in series with the +5 V supply line, and we used this cable to connect the gumstix to the computer. We calculated the power draw of the gumstix by measuring the voltage drop across the sense resistor. All power numbers presented in this section are averaged across at least five runs.

[3]`http://www.wattsupmeters.com/`

Condition	Optiplex 745	Dimension 4600
Normal idle state	102.1 W	72.7 W
Lowest CPU frequency	97.4 W	N/A
Disable multiple cores	93.1 W	N/A
'Base power'	93.1 W	72.7 W
Suspend state (S3)	1.2 W	3.6 W
Time to enter S3	9.4 s	5.8 s
Time to resume from S3	4.4 s	6.2 s

Table 1: *Power consumption and S3 suspend/resume time for two desktops under various operating conditions. In all cases the processor is idle and the hard disk is spun down. The power consumed by other peripherals such as displays is not included.*

Condition	Lenovo X60	Toshiba M400	Lenovo T60
Normal idle state	16.0 W	27.4 W	29.7 W
Backlight minimum	13.8 W	22.4 W	24.7 W
Screen turned off	11 W	18.3 W	21.3 W
'Base power'	11 W	18.3 W	21.3 W
Suspend state (S3)	0.74 W	1.15 W	0.55 W
Battery capacity	65 Wh	50 Wh	85 Wh
Base lifetime	5.9 h	2.7 h	4.0 h
Suspend lifetime	88 h	43 h	155 h
Time to enter S3	8.7 s	5.5 s	4.9 s
Time to resume from S3	3.0 s	3.6 s	4.8 s

Table 2: *Power consumption and battery lifetime of three laptops under various operating conditions, and the time to change power states.*

5.1 Microbenchmarks – Power, Latency

Desktops: Table 1 presents the average power consumption for two Dell desktop machines: an Intel dual core (2.4 GHz Core2Duo) OptiPlex 745 with 2 GB RAM running Windows Vista, and a 2.4 GHz Pentium 4 Dimension 4600 with 512 MB RAM running Windows XP. The display is turned off in these experiments, and only the essential system processes are left running. The power consumption of the desktop in S3 is two orders of magnitude less than when it is awake. This is consistent with prior published data on the power consumption of modern PCs [7]. We use the term 'base power' to indicate the lowest power mode that a PC can be in and still be responsive to network traffic (without using Somniloquy). To get this number, we further scaled down the CPU to the lowest permissible frequency on these desktops. Furthermore, we disabled the multi-core functionality using the system BIOS to effectively use only one core and verified that the system was actually doing so by using a processor ID utility supplied by Intel. The time taken for the desktops to resume from S3 and reconnect to the network is of the order of a few seconds (Table 1).

	Gumstix state	Power
	Wired version	
1	gumstix only - no Ethernet	210 mW
2	gumstix + Ethernet idle	1073 mW
3	gumstix + Ethernet bridging	1131 mW
4	gumstix + Ethernet + write to flash	1675 mW
5	gumstix + Ethernet broadcast storm	1695 mW
6	gumstix + Ethernet unicast storm	1162 mW
	Wireless version	
7	gumstix only – no Wi-Fi	210 mW
8	gumstix + Wi-Fi associated (PSM)	290 mW
9	gumstix + Wi-Fi associated (CAM)	1300 mW
10	gumstix + Wi-Fi broadcast storm	1350 mW
11	gumstix + Wi-Fi unicast storm	1600 mW

Table 3: *Power consumption for the gumstix platform in various states of operation.*

Laptops: Table 2 presents the average power consumption of three popular laptops: a Lenovo X60 tablet PC with 2 GB RAM running Windows Vista, a Toshiba laptop with 1 GB RAM running Windows XP, and a Lenovo T60 laptop with 1 GB RAM running Windows Vista. For all power measurements, the processor is set to the lowest speed and is idle, the hard disk is spun down and the wireless network interface is powered on. The base power is between 11 W and 22 W, resulting in a battery lifetime of around 4 to 5 hours with the batteries that are present on these laptops. Using the sleep/S3 state can dramatically extend the battery lifetime, to between 40 and 150 hours for the laptops we tested, although the laptop is unreachable in this state.

Gumstix: Table 3 shows the average power consumed by the gumstix (with both etherstix and wifistix) in various states of operation. The gumstix has a base power of approximately 210 mW when no network interface is present (row 1). A gumstix with an active network interface typically consumes approximately 1070-1300 mW (rows 2 and 9), however with an associated Wi-Fi interface in power save mode it consumes only 290 mW (row 8). The power consumption of the gumstix when its network interface is active and the downloaded data is being written to flash is around 1675 mW (row 4). Broadcast and unicast 'storms' (continuous traffic) increase the power consumption by a few hundred milliwatts[4]. Importantly, the power consumption of the gumstix is approximately one tenth that of an awake laptop in the lowest power state, and approximately 50 times less than an idle desktop.

[4]Wi-Fi broadcasts are sent at 6 Mbps while unicasts are sent at 54 Mbps in our setup. Consequently a unicast storm consumes more power than a broadcast storm.

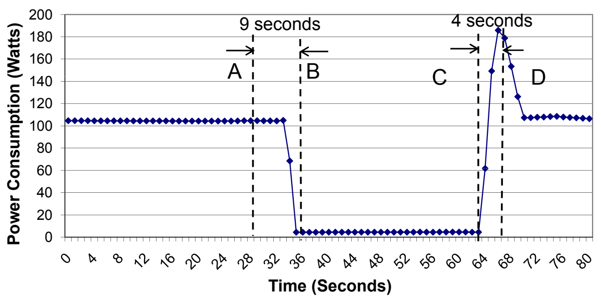


Figure 5: *Power consumption and state transitions for our desktop testbed.*

5.2 Somniloquy in Operation

We now report the power consumption of Somniloquy in operation. For these measurements we use two testbed systems: a desktop (Dell OptiPlex 745 with 2 GB RAM running Windows Vista) with the Wired-1NIC prototype of Somniloquy, and a laptop (Lenovo X60 tablet PC running Windows Vista) with the Wireless-2NIC version of Somniloquy. Thus, our tests span both Ethernet and Wi-Fi networks, and both the integrated single network interface, and the higher performance versions which uses the existing internal network interface. The test traffic is generated using a standard desktop machine running on the same (wireless or wired) LAN subnet as the testbed machine.

Figure 5 shows the power consumption of our desktop testbed. Initially the desktop's host processor is awake and uses the gumstix for bridging, and the whole system draws 104 W of power. At time 'A' a state change to S3 is initiated by the user. This request completes at time 'B' after which the power draw of the system is approximately 4.4 W, i.e. 24x less. This power is split between the gumstix, the DRAM of the PC, and other power chain elements in the PC. Subsequently at time 'C' the gumstix, which has been actively monitoring the network interface, wakes up the host in response to a network event. This request completes at time 'D' when the host system has fully resumed. As the figures illustrate this resume event takes about 4 seconds. We do not show the laptop figure for space reasons; the trace looks very similar with a starting power of 16 W with the screen on (which drops to 11 W if the screen is turned off), a power draw of 1 W when using Somniloquy (11x less than the screen-off case) and a resume time of 3 seconds.

5.3 Application Performance

As described earlier there are two classes of applications that are supported by Somniloquy: first, a large class of applications that do not require application stubs, and second a smaller class of applications that can be sup-

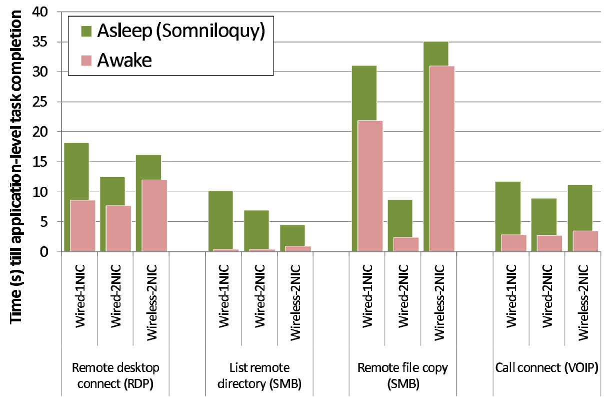


Figure 6: *Application-layer latency for three Somniloquy testbeds and four application types.*

ported using application stubs running on the gumstix. We performed a number of experiments to evaluate the performance of both these classes of applications.

5.3.1 Applications without stubs

We now quantify the end-to-end latency (as perceived by users) incurred by the applications that are handled by Somniloquy without using application stubs. For these experiments, we use the same two testbeds as above, with the addition of a third testbed based on the Wired-2NIC prototype (using same desktop machine as the Wired-1NIC case), providing a direct comparison between the 1NIC and 2NIC cases. In each case the latency reported is the mean over five test runs.

Figure 6 reports the time taken to satisfy an incoming application-layer request for four sample applications. For each application, we show the latency for "awake" operation (i.e. when the host is on and directly responds to the request) and when the host is in S3 and Somniloquy prototype receives the incoming packet and triggers wake-up of the host.

The four applications we tested were:

Remote desktop access (RDP): Here we used a stopwatch to measure the latency between initiating a remote desktop session to the host and the remote desktop being displayed. A stopwatch was used to ensure that true user-perceived latency was measured. The gumstix was configured to wakeup the main processor on detecting TCP traffic on port 3389 (the RDP port).

Remote directory listing (SMB): A directory listing from the Somniloquy testbed was requested by the tester machine (via Windows file sharing, which is based on the SMB protocol). The time between the request being initiated and the listing being returned was measured using a simple script. The secondary processor was configured to initiate wake-up on detection of traffic on either of the TCP ports used by SMB,i.e. ports 137 and 445.

Remote file copy (SMB): The SMB protocol was used again, but this time to transfer a 17 MB file from the Somniloquy testbed to the tester machine.

VoIP call (SIP): A Voice-over-IP call was placed to a user who had been running a SIP client on the Somniloquy laptop before it had entered S3. On receipt of the incoming call the SIP server responded with a TCP connection to the testbed, causing the gumstix to trigger wakeup. A similar procedure was used in [2]. Once again, the latencies were measured using a stopwatch to measure true user-perceived delay.

As Figure 6 shows, Somniloquy adds between 4-10 s latency in all cases. As described in Section 5.2 earlier, part of this latency is attributed to resuming from S3, i.e. 4-5 s for the desktop and 2-3 s for the laptop, and is independent of Somniloquy. Further latency is due to the delay for TCP to retransmit the request, and for the host to respond to the request (which may take longer since it has just resumed). Note that the Wired-1NIC prototype shows higher latency than the Wired-2NIC prototype. This is purely an artifact of our prototype caused by the overhead of MAC bridging and largely the slower speed of the USBNet IP link between the gumstix and the host. The latter is particularly obvious in the file copy test, where the file copy time with the Wired-2NIC case is much faster than for Wired-1NIC (although the Wired-1NIC speed is still faster than Wireless-2NIC). While Somniloquy does result in 4-10 s additional application-layer latency, these delays are acceptable for real usage (including VoIP [2]) in exchange for the substantial benefit of 20x-50x power savings.

5.3.2 Applications Requiring Stubs

In this section we present evaluations for applications that require stub support on the gumstix, primarily looking at the overhead in terms of memory consumption and processing capabilities that they impose on the gumstix. We have implemented application stubs for three common applications — background downloads using the http protocol, P2P file sharing using BitTorrent, and maintaining presence on IM networks — as described in Section 4.

To study the overhead of IM clients, we run the corresponding application stub using up to three different IM protocols simultaneously — MSN Messenger, AOL Messenger and ICQ Chat. Table 4 shows the processor utilization and memory footprint of the Wired-1NIC prototype when running these IM clients. Since the behavior of the IM stub is such that it maintains presence of the user on various networks and on receipt of an appropriate trigger (IM from someone) wakes up the host, the latency values are similar to those of the VoIP application

Accounts	Processor 95th percentile	Memory 95th percentile
None	0.0%	5.9 MB
MSN only	10.0%	6.5 MB
MSN+AOL	21.6%	6.7 MB
MSN+AOL+ICQ	26.0%	6.9 MB

Table 4: *Processor and memory utilization for the IM stub for various configurations. Total memory for the gumstix is 64 MB.*

Configuration	Processor 95th percentile	Memory 95th percentile
Single download		
4MB cache	16.0%	6.5 MB
8MB cache	16.0%	10.6 MB
16MB cache	16.1%	18.9 MB
Two simultaneous downloads (4 MB cache)		
1st download	16%	6.5 MB
2nd download	24%	7.0 MB

Table 5: *Processor and memory utilization for the BitTorent stub for various configurations. Total memory for the gumstix is 64 MB.*

as reported in Figure 6. For our Wired-1NIC prototype the additional latency for the IM stub when using Somniloquy is around seven seconds.

To evaluate the overhead of P2P file sharing using the BitTorrent stub on the gumstix, we initiated downloads using a torrent from a remote website[5] into the 2 GB SD card of the Wired-1NIC gumstix. We varied the memory cache available to the stub while conducting a single download, and then tested two simultaneous downloads. The results in Table 5 show that the memory footprint of the stub increases proportionally to the cache size as expected, while the processor utilization remains constant. When there are two simultaneous downloads, each instance of the stub uses memory proportional to its specified 4 MB cache.

Finally, to evaluate the web-download stub on the gumstix we initiate download of a large (300 MB) file from a local web server. We varied the throughput of the downloads and measured the processor utilization and the memory consumption of the gumstix, and experimented with two simultaneous downloads. As shown in Table 6, the processor utilization increases as the download rate increases although the memory footprint for each download remains constant.

The above results show that using application stubs, we can support fairly complex tasks and applications, including background web downloads and P2P file sharing using relatively modest resources on the gumstix. It is important to note that the power consumption of the gumstix did not exceed 2 W in all of these experiments.

[5]http://www.legaltorrents.com/

Configuration	Processor 95th percentile	Memory 95th percentile
Single download		
2Mbps	9.2%	1.8 MB
4Mbps	21%	1.8 MB
8Mbps	50%	1.8 MB
Two simultaneous downloads (4 Mbps each)		
1st download	31%	1.8 MB
2nd download	26.3%	1.8 MB

Table 6: *Processor and memory utilization for the web download stub for various configurations. Total memory for the gumstix is 64 MB.*

5.4 Energy Savings using Somniloquy

In addition to evaluating the operating performance of our Somniloquy prototypes, it's also important to assess the higher level goal of this work, namely the impact on PC energy consumption. In this section we present some data which demonstrates the potential of Somniloquy to reduce both desktop and laptop energy usage in general terms. We also verify the energy saving model presented in Section 3.3, which allows the specific savings in a given application scenario to be calculated. Unless otherwise noted, we are using the Wired-1NIC version of our prototype for the desktop energy measurements and the Wireless-2NIC version for the laptop energy measurements.

5.4.1 Reducing Desktop Energy Consumption

Our testbed desktop PC consumes 102 W in normal operation and <5 W in S3 with Somniloquy. Somniloquy therefore saves around 97 W. On this basis, if Somniloquy were to be deployed in an environment where a PC is actively used for an average of 45 hours each week (i.e. 27% of the time), this would result in 620 kWh of savings per computer in a year. Assuming 0.61 kg CO_2/kWH[6] and US$ 0.09/kWH[7], this means an annual saving of 378 kg of CO_2 (to put it in perspective, the average US residents annual CO_2 emissions are 20 metric tonnes as compared to a worldwide average of 4 metric tonnes per person[8]) and US$ 56 per computer. We

[6]`http://www.eia.doe.gov/cneaf/electricity/page/co2_report/co2report.html`

[7]`http://www.eia.doe.gov/cneaf/electricity/epa/epa_sum.html`

[8]`http://www.sciencedaily.com/releases/2008/04/080428120658.htm`

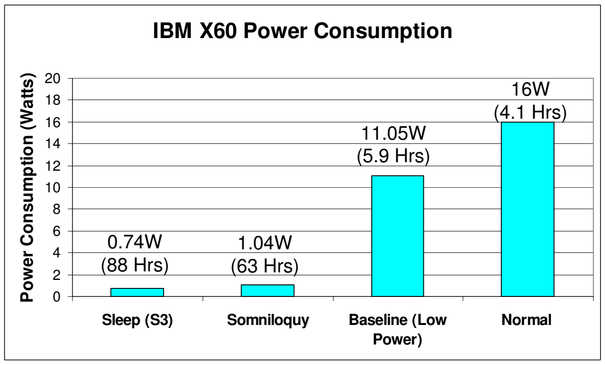


Figure 7: *Power consumption and the resulting estimated battery lifetime of a Lenovo X60 using Somniloquy. The lifetime is calculated using the standard 65 Watt hour battery of the laptop.*

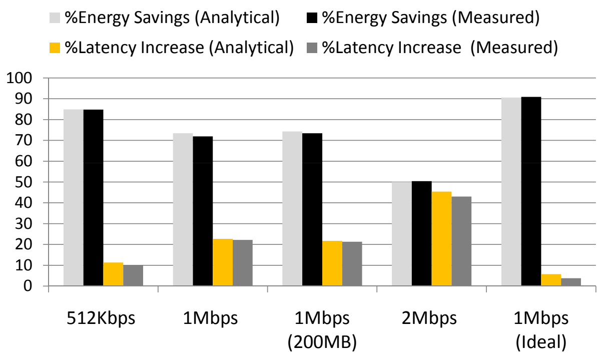


Figure 8: *Comparing the analytical results with the measured values for the web-download stub. The flash storage available on the gumstix is set to 100 MB, unless stated otherwise.*

believe this is significantly higher than the bill of materials cost of the components required to implement a commoditized Somniloquy-enabled network card. In this case, deployments of Somniloquy-enabled devices would pay for themselves within a year.

5.4.2 Desktop Energy Savings for Real Workloads

We now estimate the energy savings enabled by Somniloquy under realistic workloads. We use the data provided by [20], relating to the use patterns of twenty two distinct desktop PCs; each of which is classified as being either idle, active, sleep or turned off. We then compute the energy consumed by each of the PCs with and without Somniloquy using the formula of Section 3.3. For ease of exposition, we bin the data into three different categories: PCs that are idle for $<$25% of the time (7 machines), idle for 25%-75% of the time (6 machines) and finally those that are idle for $>$75% of the time (9 machines). The average energy savings for these twenty two PCs when using Somniloquy is 65%, as compared to normal operation without Somniloquy. The average energy savings for the PCs in the individual categories are 38%, 68% and 85% respectively. As expected, the most energy savings are for the PCs with larger idle times since they have more opportunity to use Somniloquy.

5.4.3 Increasing Laptop Battery Lifetime

Figure 7 shows the average power consumption of the laptop testbed when operating normally (i.e. no power saving mechanisms), with standard power saving mechanisms in place (the baseline power), when Somniloquy (Wireless-2NIC) is operational, and in the standard S3 mode (without the gumstix attached). Somniloquy adds a relatively low overhead of 300 mW to S3 mode, resulting in a total power consumption which is close to just 1 W, as compared to the 11 W of the idle laptop. This means that when the laptop needs to be attached to the network and available for remote applications but is otherwise idle, it can be put into Somniloquy mode to enable an order of magnitude decrease in power consumption and a resulting increase in battery lifetime from 5.9 hours to 63 hours (using the standard 65 Watt-Hour battery).

5.4.4 Energy Savings for Specific Applications

The basic analysis of energy consumption and battery lifetime presented above is very generic; for a given usage scenario it should be possible to use the energy saving model presented in Section 3.3 to predict savings much more accurately. In order to validate this model we ran experiments downloading content from a remote web server, and measured both energy consumption and latency so as to compare them with their corresponding analytical values. Note that we only measure the energy consumption for the duration of the application.

The web download stub was chosen since it was relatively easy to change the duty-cycle of the host, i.e. the duration for which the host can sleep (T_{sleep}) after which it needs to be woken up to transfer data from the gumstix (T_{awake}). As discussed in Section 3.3, T_{sleep} depends on the download bandwidth and the amount of flash storage on the gumstix, while T_{awake} depends on the amount of flash storage on the gumstix and the transfer rate between the gumstix and the host. We downloaded a 300 MB file at various link bandwidths ranging from 512 Kbps to 2 Mbps, and used two different flash storage sizes at the gumstix - 100 MB and 200 MB, effectively varying T_{sleep} from approximately 1600 seconds down to 400 seconds. We measured the power consumed during the download using the methodology described in the beginning of this section. In Figure 8, we present the measured energy savings and the corresponding predicted values using our model for four different data points. As we can see from the figure, the predicted energy savings and the increased latency closely match the measured values (within 1.5%). The values do not exactly match since the actual measured power values vary over time, and the time taken to suspend and resume also varies across runs. We used a fixed value for these in the formula.

Figure 8 also illustrates that increasing the bandwidth from 512 Kbps to 2 Mbps reduces the energy savings from 85% to 50%, and increases the latency from 11% to 43%, although a larger amount of flash storage improves the energy saving and latency. As explained earlier this is due to the limited transfer speed of the USBnet interface in our prototype (<5 Mbps), because of which the PC is awake for longer periods of time while transferring the data from the gumstix (T_{awake}= 181 seconds to transfer 100 MB of data). In Figure 8 we have also plotted an ideal case (1 Mbps-ideal) where the host can read the flash storage of the gumstix directly. For the ideal case the duration for which the host needs to stay awake to transfer data from the gumstix reduces considerably (T_{awake}= 23 seconds). This improves energy savings to 91% and limits the increase in latency when using Somniloquy to less than 5%.

6 Related Work

There have been several proposals to reduce the energy consumption of desktop PCs and laptops. Prior work can largely be grouped in three categories: reducing the active power consumption of devices (when awake) [3, 5, 9, 10, 16, 17], reducing the power consumption of the network infrastructure (e.g. routers and switches) [11, 12, 21], and opportunistically putting the devices to sleep. Somniloquy falls in the third category. Since a machine in sleep state consumes significantly less power than in lowest power active state [11, 27] (verified by us in Section 5), significant energy savings are possible by putting the machine to sleep whenever possible.

For opportunistic-sleep systems, the biggest challenge is to ensure connectivity when the host is asleep. Prior techniques to solve this problem either use advanced functionality in the NIC [18] or use extra network interfaces [26, 27]. We now compare and contrast Somniloquy to both these classes of work.

Among schemes that do not use an extra network interface, the most well-known are Wake-on-LAN (WoL) [18] and its wireless equivalent, Wake-on-WLAN (WoWLAN). In both these schemes, the NIC parses incoming packets when the host is asleep. It wakes up the host PC whenever an incoming "magic" packet is received. According to the specification [18], the magic packet payload must include 6 characters of a wakeup pattern that is set by the host PC, followed by 8 copies of the NIC's MAC address. In WoWLAN, the only difference is that this packet is sent over the Wireless LAN. Although most modern NICs implement WoL functionality, few deployed systems actually use this functionality, due to four main reasons. First, the remote host must know that the PC is asleep and that it must wake it up before pursuing application functionality. Second, the remote host must have a way of sending a packet to the sleeping PC through any firewalls/NAT boxes, which typically do not allow incoming connections without special configuration. Third, the remote host must know the MAC address of the sleeping PC. Fourth, WoWLAN does not work when laptops change their subnet because of mobility. In contrast, Somniloquy does not require the extra configuration of firewalls/NAT boxes, and is transparent to remote application servers. It can handle mobility across subnets since the secondary processor can re-associate with services such as Dynamic DNS (to redirect a permanent host name to the PC's new IP address), and re-log-in to servers such as IM servers. In addition to these differences, Somniloquy also allows applications to be offloaded to the low power processor. There is no such concept in WoL, which instead wakes up the host when any pattern is matched.

Intel recently announced its Remote-Wake' [14] chipset technology (RWT) that claims to extend WoL on new motherboards by allowing VoIP calls to wake up a system, although its general applicability to other applications is not known. The details of this technology are not published. In contrast, Somniloquy goes beyond just WoL or RWT. It allows low power operation for various applications other than VoIP. Furthermore, Somniloquy does not require modifications to application end points or servers. RWT requires applications to first contact a server, which then sends a special packet to the PC to signal a wake up.

Another approach is to use additional "low-power" network interfaces to maintain connectivity to the PC that is asleep. This approach has been proposed for use with mobile devices. For example, Wake-on-Wireless [26] wakes up the host PC on receiving a special packet on the low power network interface. Turducken [27] uses several tiers of network interfaces and processors with different power characteristics, and wakes up the upper tier when the lower tier cannot handle a task. In contrast to these schemes, Somniloquy requires only a single network interface, and presents the paradigm of a single PC to users rather than a multi-tiered system, preserving the current user experience and therefore requiring less training to use. Somniloquy also gives the impression to remote application servers that a device remains awake all the time even though it is actually asleep, since the same MAC and IP addresses are used. This level of transparency is not provided either by Wake-on-Wireless or Turducken. Finally, we have gone into more detail than previous work on ways of supporting applications that require interactions among the secondary and the host processor to perform offload – such as IM, BitTorrent and web downloads.

To reduce the power consumed by desktop PCs, some early proposals have suggested the use of proxies on the subnet that function on behalf of the desktop PC when it is asleep [4, 7, 11]. The proxy monitors incoming packets for the PC, and wakes it up using WoL when the PC needs to handle the packet. We are not aware of any published prototype implementations of such systems. Recently, Sabhanatarajan et. al. [25] propose a smart NIC that can act as proxy for a host to save power. However, the authors focus primarily on the design of a high speed packet classifier for such an interface. In comparison, Somniloquy has much wider applicability than the above schemes. It can be used in homes and small offices where it might be infeasible to deploy a dedicated server to handle processing for another PC.

A contemporaneous effort to Somniloquy is the idea of a Network Connection Proxy (NCP) [15, 20], which is a network entity that maintains the presence of a sleeping PC. In [15], the authors define the requirements of an NCP and propose modifications to the socket layer (similar to Split TCP) for keeping TCP connections alive through a PC's sleep transitions. In [20], the authors extend these APIs to support other protocols as well. Somniloquy is similar in spirit to NCP, and NCP's socket APIs can reduce Somniloquy's overhead when waking up from sleep (Section 3.1). Furthermore, to the best of our knowledge, Somniloquy is the first published prototype of any proxying system.

We note that the concept of adding more processing to the network interface is not new. Existing products offload processing to the NIC to improve performance (TCP offload [19]) and remote manageability (Intel AMT [13]). Somniloquy uses a similar offloading paradigm, but to conserve energy instead of improving performance or manageability.

7 Conclusions

We have presented Somniloquy, a system that augments network interfaces to allow PCs to be put into low-power sleep states opportunistically, without sacrificing functionality. Somniloquy enables several new energy saving opportunities. First, PCs can be put to sleep while maintaining network reachability, without special network infrastructure as needed by previous solutions (e.g. WoL). Second, some applications can be run in sleep mode thereby requiring much less power. In this paper, we have shown the feasibility for three such applications to be run in sleep mode: BitTorrent, instant messaging, and web downloads.

Somniloquy achieves these energy savings without requiring any modifications to network, to remote application servers, or to the user experience of the PC. Furthermore, Somniloquy can be incrementally deployed on legacy network interfaces, and does not rely on changes to the CPU scheduler or the memory manager to implement this functionality, thus it is compatible with a wide class of machines and operating systems.

Our prototype implementation, based on a USB peripheral, includes support for waking up the PC on network events such as incoming file copy requests, VoIP calls, instant messages and remote desktop connections, and we have also demonstrated that file sharing/content distribution systems (e.g. BitTorrent, web downloads) can run in the augmented network interface, allowing for file downloads to progress without the PC being awake. Our tests show power savings of 24x are possible for desktop PCs left on when idle, or 11x for laptops. For PCs that are left idle most of the time, this translates to energy savings of 60% to 80%. The electricity savings made are such that deploying a productized version of Somniloquy could pay for itself within a year.

Acknowledgements

We would like to thank John Dunagan, Gunjan Gupta, Srikanth Kandula, Jitu Padhye, Patrick Verkaik, Kashi Vishwanath and the anonymous reviewers for their comments on various versions of this paper. We would also like to acknowledge the feedback received from Alex Snoeren, Stefan Savage and Geoff Voelker. Finally, we are grateful to our shepherd, Jeffrey Mogul for meticulously guiding us towards the final version of the paper. His help was invaluable.

References

[1] ACPI. Advanced Configuration and Power Interface Specification, Revision 3.0b. `http://www.acpi.info`.

[2] Y. Agarwal, R. Chandra, A. Wolman, P. Bahl, K. Chin, and R. Gupta. Wireless Wakeups Revisited: Energy Management for VoIP over Wi-Fi Smartphones. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 179–191, New York, NY, USA, 2007. ACM.

[3] Y. Agarwal, T. Pering, R. Want, and R. Gupta. "SwitchR: Reducing System Power Consumption in a Multi-Clients, Multi-Radio Environment". In *Proceedings of IEEE International Symposium on Wearable Computing (ISWC)*, 2008.

[4] M. Allman, K. Christensen, B. Nordman, and V. Paxon. Enabling an Energy-Efficient Future Internet Through Selectively Connected End Systems. In *6th ACM Workshop on Hot Topics in Networks (HotNets)*. ACM, November 2007.

[5] M. Anand, E. B. Nightingale, and J. Flinn. Self-tuning Wireless Network Power Management. In *MobiCom '03: Proceedings of the 9th annual international conference on Mobile computing and networking*, pages 176–189, New York, NY, USA, 2003. ACM Press.

[6] N. Borisov, D. Brumley, H. J. Wang, J. Dunagan, P. Joshi, and C. Guo. A generic application-level protocol analyzer and its language. In *Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS)*, 2007.

[7] K. Christensen, C. Gunaratne, and B. Nordman. The Next Frontier for Communication Networks: Power Management. *Computer Communications*, 27(18):1758–1770, 2004.

[8] W. Cui, J. Kannan, and H. J. Wang. Discoverer : Automatic Protocol Reverse Engineering from Network Traces. In *Proceedings of the USENIX Security Symposium*, 2007.

[9] K. Flautner, S. K. Reinhardt, and T. N. Mudge. Automatic Performance Setting for Dynamic Voltage Scaling. In *MobiCom '01: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 260–271, 2001.

[10] J. Flinn and M. Satyanarayanan. Managing Battery Lifetime with Energy-Aware Adaptation. *ACM Trans. Comput. Syst.*, 22(2):137–179, 2004.

[11] C. Gunaratne, K. Christensen, and B. Nordman. Managing Energy Consumption Costs in Desktop PCs and LAN Switches with Proxying, Split TCP Connections, and Scaling of Link Speed. *Int. J. Netw. Manag.*, 15(5):297–310, 2005.

[12] M. Gupta and S. Singh. Greening of the Internet. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 19–26, New York, NY, USA, 2003. ACM.

[13] Intel. Intel Active Management Technology (AMT). `http://www.intel.com/technology/platform-technology/intel-amt/`.

[14] Intel. Intel Remote Wake Technology. `http://www.intel.com/support/chipsets/rwt/`.

[15] M. Jimeno, K. Christensen, and B. Nordman. A Network Connection Proxy to Enable Hosts to Sleep and Save Energy. In *IEEE International Performance Computing and Communications Conference*, 2008.

[16] R. Kravets and P. Krishnan. Application-driven Power Management for Mobile Communication. *Wireless Networks*, 6(4):263–277, 2000.

[17] X. Li, R. Gupta, S. V. Adve, and Y. Zhou. Cross-Component Energy Management: Joint Adaptation of Processor and Memory. *ACM Trans. Archit. Code Optim.*, 4(3):14, 2007.

[18] P. Lieberman. Wake-on-LAN technology. `http://www.liebsoft.com/index.cfm/whitepapers/Wake_On_LAN`.

[19] J. C. Mogul. TCP Offload Is a Dumb Idea Whose Time Has Come. In *HotOS*, pages 25–30, 2003.

[20] S. Nedevschi, J. Chandrashekar, B. Nordman, S. Ratnasamy, and N. Taft. Skilled in the art of being idle: reducing energy waste in networked systems. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[21] S. Nedevschi, L. Popa, G. Iannaccone, S. Ratnasamy, and D. Wetherall. Reducing Network Energy Consumption via Sleeping and Rate-Adaptation. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 323–336. USENIX Association Berkeley, CA, USA, 2008.

[22] R.Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and T. Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, June 1999.

[23] J. Roberson, C. Webber, M. McWhinney, R. Brown, M. Pinckard, and J. Busch. After-hours Power Status of Office Equipment and Energy use of Miscellaneous Plug-load Equipment. *Lawrence Berkeley National Laboratory, Berkeley, California. Report# LBNL-53729-Revised*, 2004.

[24] K. Roth and K. McKenney. Energy Consumption by Consumer Electronics in US Residences. *Final Report to the Consumer Electronics Association (CEA)*, 2007.

[25] K. Sabhanatarajan, A. G.-R. M. Oden, M. Navada, and A. George. Smart-NICs: Power Proxying for Reduced Power Consumption in Network Edge Devices. In *ISVLSI '08*, 2008.

[26] E. Shih, P. Bahl, and M. J. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *MobiCom '02: Proceedings of the 8th annual international conference on Mobile computing and networking*, pages 160–171, New York, NY, USA, 2002. ACM Press.

[27] J. Sorber, N. Banerjee, M. D. Corner, and S. Rollins. Turducken: Hierarchical Power Management for Mobile Devices. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

Skilled in the Art of Being Idle: Reducing Energy Waste in Networked Systems

Sergiu Nedevschi[*] [†] Jaideep Chandrashekar[†] Junda Liu[‡] [*] Bruce Nordman[§]
Sylvia Ratnasamy[†] Nina Taft[†]

Abstract

Networked end-systems such as desktops and set-top boxes are often left powered-on, but idle, leading to wasted energy consumption. An alternative would be for these idle systems to enter low-power sleep modes. Unfortunately, today, a sleeping system sees degraded functionality: first, a sleeping device loses its network "presence" which is problematic to users and applications that expect to maintain access to a remote machine and, second, sleeping can prevent running tasks scheduled during times of low utilization (*e.g.,* network backups). Various solutions to these problems have been proposed over the years including wake-on-lan (WoL) mechanisms that wake hosts when specific packets arrive, and the use of a proxy that handles idle-time traffic on behalf of a sleeping host. As of yet, however, an in-depth evaluation of the potential for energy savings, and the effectiveness of proposed solutions has not been carried out. To remedy this, in this paper, we collect data directly from 250 enterprise users on their end-host machines capturing network traffic patterns and user presence indicators. With this data, we answer several questions: what is the potential value of proxying or using magic packets? which protocols and applications require proxying? how comprehensive does proxying need to be for energy benefits to be compelling? and so on.

We find that, although there is indeed much potential for energy savings, trivial approaches are not effective. We also find that achieving substantial savings requires a careful consideration of the tradeoffs between the proxy complexity and the idle-time functionality available to users, and that these tradeoffs vary with user environment. Based on our findings, we propose and evaluate a proxy architecture that exposes a minimal set of APIs to support different forms of idle-time behavior.

[*]International Computer Science Institute
[†]Intel Research
[‡]University of California, Berkeley
[§]Lawrence Berkeley National Laboratories

1 Introduction

Recent years have seen rising concern over the energy consumption of our computing infrastructure. A recent study [19] estimates that, in the U.S. alone, energy consumption for networked systems approaches 150 TWh, with an associated cost of around 15 billion dollars. About 75% of this consumption can be attributed to homes and enterprises, and the remaining 25% to networks and data centers. Our focus in this paper is on reducing the 75% consumed in homes and enterprises. To put this in perspective, this energy (112 TWh) is roughly equivalent to the yearly output of 6 nuclear plants [14]. Of equal concern is that this consumption has grown – and continues to grow – at a rapid pace.

In response to these energy concerns, computer vendors have developed sophisticated power management techniques that offer various options by which to reduce computer power consumption. Broadly, these techniques all build on hardware support for *sleep* (S-states), and frequency/voltage scaling [21] (processor P-states [4]). The former is intended to reduce power consumption during idle times, by powering down sub-components to different extents, while the latter reduces power consumption while active, by lowering processor operating frequency and voltage during active periods of low system utilization.

Of these, sleep modes offer the greatest reduction in the power draw of machines that are *idle*. For example, a typical sleeping desktop draws no more than 5W [2], as compared to at least 50W [2] when on, but idle – an order of magnitude reduction. It is thus unfortunate that sleep modes are not taken advantage of to anywhere close to their fullest potential. Surveys of office buildings have shown that about two thirds of desktops are fully on at night [20], with only 4% asleep. Our own measurements (Section 3) reveal that enterprise desktops remain idle for an average of 12 hours/day – time that could, in theory, be spent mostly sleeping.

Relative to an idle machine, the only loss of functionality to a sleeping machine is twofold. First, since a sleeping computer cannot receive or transmit network messages, it effectively loses its "presence" on the network.

This can lead to broken connections and sessions when the machine resumes (*e.g.,* a sleeping machine does not renew its DHCP lease and hence loses its IP address) and also prevents remote access to a sleeping computer. This loss of functionality is problematic in an increasingly networked world. For example, a user at home might want to access files on his desktop at work, an on-the-road user might want to download files from his home machine to his handheld, system administrators might desire access to enterprise machines for software updates, security checks and so forth. In fact, some enterprises, *require* that users not power off their desktops to ensure administrators can access machines at all times [6]. The second problematic scenario is when users or administrators deliberately want to schedule tasks to run during idle times – *e.g.,* network backups that run at night, critical software updates, and so on. Unfortunately, these drawbacks cause users to forego the use of sleep modes leading to wasteful energy consumption.

The above observations are not new, having been repeatedly articulated (also by some of the authors) in both the technical literature and popular press [13, 16, 19, 10, 7, 15]. Likewise, there have been two long-standing proposals on how to tackle the problem. The first is to generalize the old technology of Wake-on-LAN (WoL), an Ethernet computer networking standard that allows a machine to be turned on or woken up remotely by a special "magic packet". A second, more heavyweight, proposal has been to use a *proxy* that handles idle-time traffic on behalf of a sleeping host(s), waking the sleeping host when appropriate. Thus both problem (wasted energy consumption by idle computers) and proposed solutions (wake-up packets and/or proxies for sleeping machines) have existed for a while now. In fact, the technology for WoL has been implemented and deployed although not widely used (we explore possible causes for this later in the paper). However the recent focus on energy consumption has led to renewed interest in the topic with calls for research [7, 13], calls for standardization [12], and even some commercial prototypes [15]. As yet however, there has been little systematic and in-depth evaluation of the problem or its solutions – what savings might such solutions enable? what is the broader design space for solutions? what, if any, might be the role of standardization? are these the right long-term solutions? *etc.*

In this paper, we explore these questions by studying user behavior and network traffic in an enterprise environment. Specifically, we focus on answering the following questions:

Q1: Is the problem worth solving? Just how much energy is squandered due to poor computer sleeping habits? This will tell us the potential energy savings these solutions stand to enable and hence the complexity they warrant. Also, is proxying really needed to realize these potential savings or can we hope that WoL suffices to maintain network presence while still sleeping usefully?

Q2: What network traffic do idle machines see? Understanding this will shed light on how this idle-time traffic might be dealt with and, consequently, what protocols and applications might trigger wake-up packets and/or require proxying. On the face of it, it would seem like an idle machine ought not to be engaged in much useful activity and hence, ideally, one might hope that a small number of wake-up events are required and/or that a relatively small set of protocols must be proxied to realize useful savings.

Q3: What is the design space for a proxy? In general, the space appears large. Different proxy implementations might vary in the complexity they undertake in terms of what work is handled by the proxy *vs.* waking the machine to do so. In some cases, one might opt for a relatively simple proxy that (for example) only responds to certain protocols such as ARP (specified by the DMTF ASF2.0 standard[1]) and NetBios. But more complex proxies are also conceivable. For example, a proxy might take on application-specific processing such as initiating/completing BitTorrent downloads during idle times and so forth. Likewise, there are many conceivable deployment options – a proxy might run at a network middlebox (*e.g.,* firewall, NAT, *etc.*), at a separate machine on each subnet, or even at individual machines (*e.g.,* on its NIC, on a motherboard controller, or on a USB-attached lightweight microengine). Given this breadth of options, we are interested in whether one can identify a minimal proxy architecture that exposes a set of open APIs that would accommodate a spectrum of design choices and deployment models. Doing so appears important because a proxy potentially interacts with a diversity of system components and even vendors (hardware power management, operating systems, higher-layer applications, network switches, NICs, *etc.*) and hence identifying a core set of open APIs would allow different vendors to co-exist and yet innovate independently. For example, an application developer should be able to define the manner in which his application interacts with the proxy with no concern for whether the proxy is deployed at a firewall, a separate machine or a NIC.

Q4: What implications does proxying have for future protocol and system design? The need for a proxy arises largely because network protocols and applications were never designed with energy efficiency in mind nor to usefully exploit, or even co-exist with, power management in modern PCs and operating systems. While proxies offer a pragmatic approach to dealing with this mismatch for currently deployed protocols and software, one might also take a longer-term view of the problem and ask how we might redesign protocols, applications or even hardware power management to eventually obviate the need for such proxying altogether.

In this paper, we study the network-related behavior of 250 users and machines in enterprise and home environments, and evaluate each of the above questions in Sections 3 to 6 respectively.

2 Measurement data and methodology

We collected network and user-level activity traces from approximately 250 client machines belonging to Intel corporation employees, for a period of approximately 5 weeks. The machines, running Windows XP, include both desktops and notebooks—approximately 10% are desktops and the rest, notebooks.

Our trace collection software was run at the individual end-hosts themselves and hence, in the case of notebooks, trace collection continued uninterrupted as the user moved between *enterprise* and *home*, enabling us to analyze traffic from both of these environments.

Our packet traces were collected using Windump. To capture user activity, we developed an application that sampled a number of user activity indicators at one second intervals. The user activity indicators we collected included keyboard activity and mouse movements and clicks. Noticeable gaps in the traces occur when the host was turned off, put to sleep, or in hibernation. Thus each end-host is associated with a trace of its network and user activity. We then used BRO [9] to reassemble connection-level information from each packet-level trace.

Thus, for the 5 week duration of our measurement study, we have the following information for each end-host:

- a packet-level (pcap) trace capturing packet headers for the entire duration
- per-second indicators of user presence at the machine
- the set of all connections—incoming and outgoing—as reconstructed by BRO from the packet traces

The result is a 500GB repository of trace data. To process this, we developed a custom tool that extends the publicly available WIRESHARK [3] network protocol analyzer with different function callbacks implementing the additional processing required for our study.

3 Low Power Proxying: Potential and Need

In this section, we estimate the energy wasted by home and office computers that remain powered on even when idle, i.e., even when there is no human interacting with the computer. Subsequently, we investigate whether very simple approaches — *e.g.,* the computer is woken up to process every network packet and then returns to sleep immediately after—would suffice in allowing hosts to sleep more while preserving their network "presence".

How much energy is squandered by not sleeping?

Virtually all modern computers support advanced sleep states, S1 - S4 as defined in the ACPI specification [5].

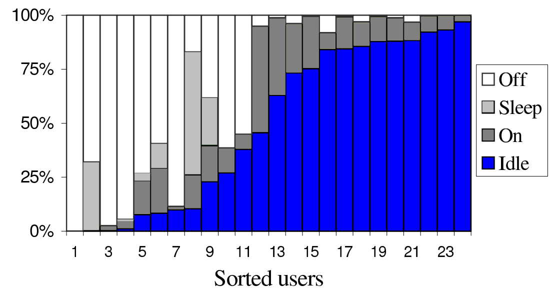


Figure 1: Distribution of the split among off, idle and active periods across users.

These states vary in their characteristics—whether the CPU is powered off, how much memory state is lost, which buses are clocked and so on. However, common to all states, is that the CPU stops executing instructions and hence the computer appears to be powered down. Thus although these sleep states conserve energy, the undesirable side-effect is that a sleeping computer effectively "falls off" the network—making it unavailable for remote access and unable to perform routine tasks that may have been scheduled at particular times. This leads many users to disable power management altogether and instead leave machines running 24/7. For example, studies have shown that approximately 60% of the PCs in office buildings remain powered on overnight and almost all of these have power management disabled [20].

To more carefully quantify the amount of wasted energy (and hence potential savings), we analyzed the trace data collected at our enterprise machines. To determine whether a machine has a locally present and active user, we examine the recorded mouse and keyboard activity for the machine: if no such activity is recorded for 15 minutes, we say that the machine is *idle*. We use 15 minutes because it is the default timeout recommended by EnergyStar for putting machines to sleep, and because it represents a simple (and fairly liberal) approximation for the notion of *idle-ness*, for which a standard definition does not exist. We maintain this definition of idle-ness for the remainder of the paper.

At any point in time, we classify a machine as being in one of four possible states: (a) on, and actively used, we call this *active*; (b) on, but not used, *idle*; (c) in a *sleep* state such as S3 or S4, and (d) powered down, *off*. Note that this notion of "idle" refers here to the *user*, and not the machine, being inactive.

In Figure 1 we present this data for our enterprise desktops. We focus here on the desktops since this represents the potential energy savings an enterprise could garner. Because the bulk of our traces come from mobile users, we have a limited number of desktops. We see that the fraction of time when these machines are active is quite low, falling below 10% on average. Moreover, the aver-

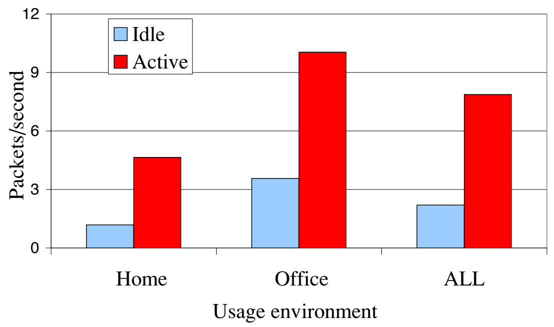


Figure 2: Average number of directed and broadcast/multicast packets received on average by a network host at home and in the office.

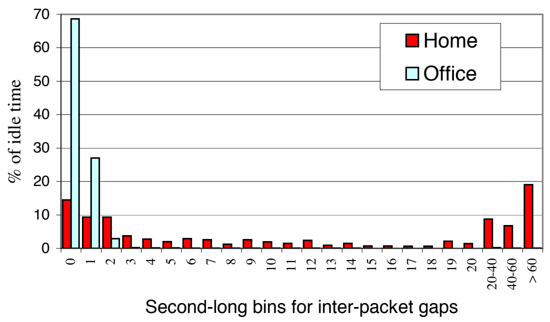


Figure 3: Histogram of the fraction of the idle time made up of inter-packet gaps of different size.

age fraction of time when machines are idle is high – about 50%. Similar to other studies, we note that a small fraction of our desktops (only 5 out 24) use sleep mode at all. Overall, this indicates that there is a tremendous opportunity for energy savings on enterprise desktops. The opportunity on our corporate laptops exists too, but is moderate because we found that our laptop users were more likely to employ aggressive sleeping configurations that come pre-configured on laptops.

While the sample of the desktop machines in our experiments is small, the results are consistent with existing studies [20]. We therefore use these measured idle times to extrapolate the energy that could be saved by sleeping instead of remaining idle. There are estimated to be about 170 million desktop PCs in the US (data summarized in [23]). Assuming an 80W power consumption of an idle PC, and assuming these machines are idle for 50% of the time, this amounts to roughly 60 TWh/year of wasted electricity (or 6 billion dollars, at US$0.10 per kWh).

Is low-power proxying needed? Before developing new solutions to reducing host idle times, we investigate whether very simple approaches like waking up for every packet can deliver these savings while maintaining full network presence. In this approach, which we denote (WoP – wake on packet), the machine is woken up for every packet it needs to receive (directed or broadcast), and put back to sleep after the packet is served. The performance of such an approach depends on whether the inter-packet gap (IPG) is smaller or comparable to the time it takes to transition in and out of sleep. If it isn't then there is no gain over simply leaving the machine in an idle state.

To examine the traffic during idle times, we used both our desktop and laptop machines. We consider both types (even though we're primarily interested in desktops) because this gives us a significantly larger set of samples. We separate the idle time traffic into two categories, office and home. In Figure 2 we plot the average number of packets/sec for idle traffic both in the office and at home. In the office environment, the average number of packets per second is roughly 3, while at home it is roughly 1. This indicates a fairly constant level of background chatter on the network, independent of the user's activity. Because this number is an average, we need to understand if these packets occur in bursts or not. If the packets are bursty most of the time, then there may still be opportunities to sleep as the host can be woken up to service a burst of packets and then be put to sleep for some reasonable period of time (certainly more than a few seconds). If these packets occur fairly evenly spaced, then it is not worth going to sleep unless the time to transition in and out of sleep is very small (on the order of 1 to 3 seconds).

To quantify the burstiness level of our traffic, we group inter-packet gaps into second-long bins (*i.e.,* 0-1s, 1-2s, *etc*.). We then compute the sum of the inter-packet gaps in each of these bins, and finally compute the fraction of total idle time represented by each bin. We present these results in Figure 3, for both home and office environments. In the office, over 90% of the time, the IPG is less than 2 seconds. Although the distribution is more uniformly spread for the home environment, we still see that roughly 70% of the time, the IPG is less than 20 seconds. Overall we observe that: (a) neither of the environments enjoys many long periods of quiet time; (b) we find this distribution to be very different for the two environments. In home networks the distribution has a much heavier tail, the traffic is burstier, and we do see longer periods of quiet time.

We now translate these observations into actual sleep time. In order to perform this computation, we must consider a representative value for the time interval it takes the host to wake up, process the packet and then go to sleep again—we call this the transition time, denoted t_s. Today, typical machines take 3 – 8 seconds to enter S3 sleep, and 3 – 5 seconds to fully resume from S3, as measured in a recent study [6]. Therefore, it is reasonable to assume an average transition time t_s of $10s$.

When a packet arrives, the machine is woken up to serve the packet. After processing a packet, the machine only goes to sleep again if it knows the next packet will not arrive before it transitions to sleep. This idealized test

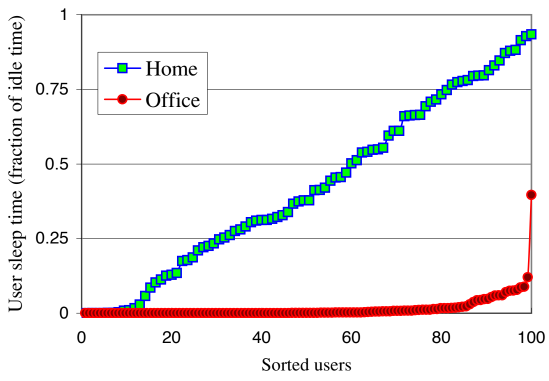


Figure 4: The fraction of idle time users can sleep if they wake up for every packet, across different environments for a transition time $t_s = 10$seconds.

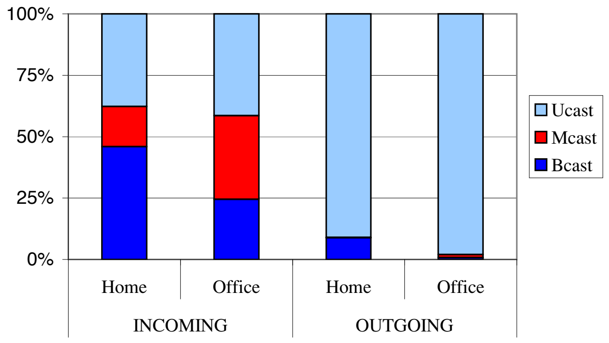


Figure 5: Composition of incoming and outgoing traffic during idle times, for home and office environments, based on communication paradigms

thus assumes that the host knows the future incoming packet stream and captures the best the machine could do in terms of energy savings.

Figure 4 presents the fraction of idle time for which users can sleep, assuming the policy described above. The results are rather dramatically different for across environments. In the office, there is almost no opportunity to sleep for the majority of the users. This indicates that the magic packet-like approach will not succeed in saving any energy for machines in a typical corporate office environment. For the home environment, we see that roughly half the users can sleep for over 50% of their idle times. Thus in these environments, a $10s$ transition time coupled with a WoP type policy can be somewhat effective. However, these estimates assume perfect knowledge of future traffic arrivals and also frequent transitions in and out of sleep—in practice, we expect the achievable savings would be somewhat lower. Nonetheless, this does suggest that efforts to reduce system transition times in future hardware could obviate the need for more complex power-saving strategies in certain environments.

We conclude that while significant opportunity for sleep exists, capitalizing on this opportunity requires solutions that go beyond merely waking the host to handle network traffic; we thus consider solutions based on proxying idle-time traffic in the following sections.

4 Deconstructing traffic

In the previous section we saw that, by just waking up to handle all packets, our ability to increase a machine's sleep time is limited. In particular, we see virtually no energy savings in the dominant office environments. This suggests that we need an approach that is more discriminating in choosing when to wake hosts. This leads us to an alternate solution to the WoL which is to employ a network proxy whose job is to handle idle-time traffic on behalf of one or more sleeping hosts. Packets destined for a sleeping host are intercepted by (or routed to, depending on the proxy deployment model) its proxy. At this point, the proxy must know what to do with this intercepted traffic; broadly, the proxy must choose between three reactions: a) ignore/drop the packet; b) respond to the packet on behalf of the machine; or c) wake up the machine to service it. To make a judicious choice, the proxy must have some knowledge of network traffic—what traffic is safely ignorable, what applications do packets belong to, which applications are essential, and so forth. In this section, we do a top-down deconstruction of the idle-time traffic traces aimed at learning the answers to these questions.

4.1 Traffic Classes by Communication Paradigm

To begin, we look at all packets exchanged during idle periods, and classify each packet as either being a broadcast, multicast or unicast packet. Within these broad traffic classes, we further partition the traffic by whether the packets are incoming or outgoing, for both the home and office environments. We separate incoming and outgoing traffic because we expect them to look different in terms of the proportion of each class in different directions (*e.g.,* most end-hosts ought to send little broadcast traffic). Similarly, we look at different usage environments because it is intuitive that the dominant protocols and applications used in each environment may differ. Since we expect these differences, we treat them as such to avoid mischaracterizations. The breakdown of our traffic according to all these partitions in depicted in Fig. 5.

We note that outgoing traffic is dominated by unicast traffic since, as expected, each host generates little broadcast or multicast traffic. We also find that incoming traffic at a host sees significant proportions of *all* three classes of traffic, and this is true in both enterprise and home environments. This suggests that a power-saving proxy might have to tackle all three traffic classes to see significant savings.

So far, we looked at traffic volumes as indicative of the need to proxy the corresponding traffic type. We now directly evaluate the opportunity for sleep represented by each traffic type. To understand the maximum sleeping opportunities, we consider for a moment an idealized scenario in which we use our proxy to ignore *all* incoming packets from either or both of the broadcast and multicast traffic classes. A machine always wakes up for unicast packets. Fig. 6 shows the sleep potential in four scenarios: a) ignore only broadcast and wake for the rest; c) ignore only multicast and wake for the rest; c) ignore both broadcast and multicast. For comparison purposes we also include the results for a scenario d) in which we wake up for all packets. This comparison allows us to compare the benefits derived from these four different proxy policies. For each user, we computed the fraction of its idle time that could have been spent sleeping under the scenario in question. We use a transition time of $t_s = 10s$ and the results are averaged over 250 users for both home and office environments.

We make the following observations:

(i) Broadcast and multicast are largely responsible for poor sleep. If we can proxy these, then we can recuperate over 80% of the idle time in home environments. And in the office, where previously sleep was barely possible, we can now sleep for over 50% of the idle time.

(ii) Doing away with only one of either broadcast or multicast is not very effective (we suspect this is due to the periodicity of multicast and broadcast protocols, and evaluate this in later sections).

More generally, the graph clearly indicates a valuable conclusion—if we're looking to narrow the set of traffic classes to proxy, then multicast and broadcast traffic appear to be clear low-hanging fruit and should be our primary candidates for proxying. That said, proxying unicast traffic appears key to achieving higher savings (beyond 50%) in the enterprise and hence should not be dismissed either. We thus continue, for now, to study all three traffic types.

Of course, whether these potential savings can actually be realized depends on whether a particular traffic type can indeed be handled by a proxy without waking the host. This depends on the specific protocols and applications within that class and hence, in the remainder of this section, we proceed in turn to deconstruct each of broadcast (§4.2), multicast (§4.3) and unicast (§4.4) traffic.

4.2 Deconstructing Broadcast

Our goal in this section is to evaluate individual broadcast protocols, looking for: (1) which of these protocols are the main offenders in terms of preventing hosts from sleeping and, (2) what purpose do these protocols serve and how might a proxy handle them. Answering the first question requires a measure of protocol "badness" with respect to preventing hosts from sleeping. We use two metrics for our evaluation. The first is simply the total **volume of traffic** due to the protocol in question. While high-volume traffic often makes sleep harder, this is an

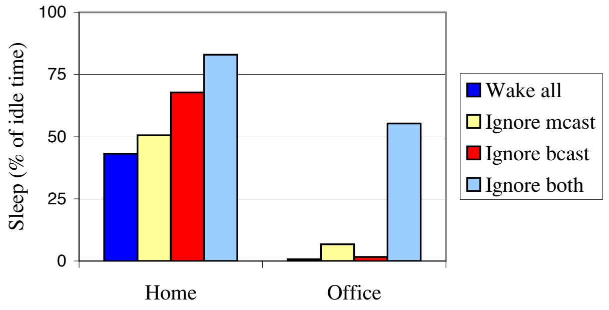


Figure 6: Average sleep opportunity when ignoring multicast and/or broadcast traffic, for different environments

imperfect metric since the (in)ability to sleep depends as much on the precise temporal packet arrival pattern due to the protocol as on packet volumes. Nonetheless, we retain traffic-volume as an intuitive, although indirect measure of protocol badness. Our second metric—which we term the **half-sleep time**, denoted `ts_50` – more directly measures a protocol's role in preventing sleep.

We define the half-sleep time for a protocol (or traffic type) P as the largest host transition time that would be required to allow the host to sleep for at least 50% of its idle time, under the scenario where the machine wakes up for all packets of type P and ignores all other traffic. In effect, `ts_50` quantifies the intuition that, if we ignore all traffic other than that due to the protocol of interest, then a protocol whose packets arrive spaced far enough apart in time is more conducive to sleep since the host has sufficient time to transition in and out of sleep.

In more detail, `ts_50` is computed from our traces as follows. We measure the total time a given host can sleep assuming it wakes up for all the packets of the protocol under consideration and ignores all others. We compute this number for all hosts and take the average. This gives us an upper bound on achievable sleep if the protocol is handled by waking the host. We estimate this sleep duration for different values of the host transition time t_s ranging from 0 seconds (ideal) to 15 minutes. The largest of these transition times t_s that allows the host to sleep for over 50% of its idle time is the protocol's `ts_50` .

Intuitively, `ts_50` indicates the extent to which a protocol is "sleep friendly" since protocols with large values of `ts_50` could simply be handled by allowing the machine to wake up; whereas those with low values of `ts_50` imply that (to achieve useful sleep) the proxy must handle such traffic without waking the host.

For our evaluation, we classify each packet by protocol and rank them by both metrics: traffic volume and the half-sleep time. We begin by measuring traffic volume, we then establish the top ranking protocols by volume, and use these as candidates for our second metric, the half-sleep time. When presenting the top ranking protocols by each of the metrics, we consider : (1) the protocols whose traffic volumes represents more than 1% of

Office		*Home*	
Protocol	**% of traffic**	**Protocol**	**% of traffic**
ARP	46.13	ARP	42.56
NBNS	22.89	SSDP	19.63
IPX	10.12	NBNS	9.48
NBDGM	5.91	CUPS	5.6
LLC	3.28	LLC	4.4
ANS	2.85	UNISTIM	4.07
RPC	2.46	IPX	3.8
BOOTP	2.01	NBDGM	2.3
NTP	1.13	BOOTP	1.02
Other	3.22	Other	7.14
Total	**100**	**Total**	**100**

Figure 7: Protocol composition of incoming broadcast traffic, in both office and home environments, ranked by per-protocol traffic volumes.

Office		*Home*	
All Bcast	**1-2s**	**All Bcast**	**10-20s**
ARP	2-3s	ARP	1-2min
NBDGM	10-20s	NBDGM	2-4min
NBNS	2-4min	NBNS	4-8min
IPX	4-8min		

Figure 8: Protocol composition for broadcast protocols ranked by `ts_50` .

the total traffic at the host and (2) the protocols with a half-sleep time of less than 15 minutes. Table 7 and 8 present our results for broadcast traffic. For completeness, we also present the value of `ts_50` when considering all broadcast traffic together.

In terms of traffic volumes, we see that the bulk of broadcast traffic is in the cause of address resolution and various service discovery protocols (*e.g.,* ARP, Netbios Name Service – NBNS, the Simple Service Discovery Protocol used by UPnP devices – SSDP). These protocols are well represented in both home and office LANs. A second well-represented category of traffic is from router-specific protocols (*e.g.,* routing protocols implemented on top of the IPX).

In terms of the half-sleep time, we see that broadcast as a whole allows very little sleep in the office: achieving 50% sleep would require very fast transitions (between 1 and 2 seconds), not feasible with today's hardware support. The situation in home LANs is significantly better (`ts_50` $= 10s$). In terms of protocols, we see that the greatest offenders are similar to those from our traffic-volume analysis, namely: ARP, Netbios Datagrams (NBDGM) and Name Queries (NBNS), and IPX.

On closer examination, we find that most of these offending protocols could be easily handled by a proxy: for example, IPX is safely ignorable, ARP traffic that is not destined to the machine in question is likewise safely ignorable; for ARP queries destined to the machine, it would be fairly straightforward for a proxy to automatically construct and generate the requisite response without having to wake the host.

Office		*Home*	
Protocol	**% of traffic**	**Protocol**	**% of traffic**
HSRP	59.58	SSDP	94.4
SSDP	24.91	HSRP	2.31
PIM	6.04	IGMP	1.84
IGMP	5.05		
EIGRP	1.88		
Other	2.54	Other	1.45
Total	**100**	**Total**	**100**

Figure 9: Protocol composition for incoming multicast traffic, in both office and home enviroments, ranked by per-protocol traffic volumes.

Office		*Home*	
All Mcast	**0-1s**	**All Mcast**	**1-2min**
HSRP	0-1s	SSDP	4-8min
PIM	8-9s	HSRP	>15min
IGMP	20-30s	IGMP	>15min
SSDP	20-30s		

Figure 10: Protocol composition for incoming multicast traffic, in both office and home environments, ranked by `ts_50` .

4.3 Deconstructing Multicast

Table 9 and 10 present our protocol rankings for multicast traffic. Again, we also present the value of `ts_50` when considering all multicast traffic taken together. We see that, multicast traffic (as a whole) can be a bad offender in enterprise environments with an `ts_50` $= 0-1s$. It turns out that this is largely caused by router traffic—the Hot Standby Router Protocol (HSRP), Protocol Independent Multicast (PIM), EIGRP, *etc.*

This traffic is either absent (*e.g.,* PIM) or greatly reduced (*e.g.,* HSRP) in home environments which explains why multicast is much less problematic in homes, with an `ts_50` $= 1-2$ minutes (compared to $10-20s$ for broadcast).

The good news is that all router traffic (HSRP, PIM, IGRP) is safely ignorable. In fact, many modern Ethernet cards already include a hardware multicast filter that discards most unwanted multicast traffic.

As with broadcast traffic, we also see significant traffic contributed by service discovery protocols: in this case SSDP, the Simple Service Discovery Protocol used by UPnP devices. Once again, for protocols such as SSDP and IGMP, it is fairly straightforward for a proxy to automatically respond to incoming traffic without waking the host; doing so would require some amount of state at the proxy such as the list of multicast groups the interface belongs to and the services running on the machine.

4.4 Deconstructing Unicast

Finally, we present our protocol ranking for unicast traffic in Tables 11 and 12. Because much of unicast traffic is either TCP or UDP, and this level of classification is unlikely to be informative, we further break each

Transport Protocol	Session Protocol	% of traffic	
TCP		94.73	
	DCE/RPC		24.91
	NBSS		14.85
	HTTP		12.31
	TPKT		3.82
	SSL		2.68
	VNC		2.45
	Other		33.71
UDP		3.75	
	DNS		1
	Other		2.75
ICMP		1.29	1.29
Other		0.23	0.23
Total		**100**	**100**

Figure 11: Protocol composition of incoming unicast traffic in office enviroments, ranked by per-protocol traffic volumes.

Office		*Home*	
All Ucast	**10-20s**	**All Ucast**	**50-60s**
TCP	10-20s	UDP	1-2min
UDP	1-2min	DNS	1-2min
DCE/RPC	1-2min	TCP	8-15min
DNS	2-4min		
SMB	4-8min		
NBNS	4-8min		
HTTP	8-15min		

Figure 12: Protocol composition of incoming unicast traffic in office environments, ranked by `ts_50` .

Port	App	ts_50
TCP keep alives	many	1-2min
UDP 53	DNS	2-4min
TCP 1025	DCE/RPC	2-4min
TCP 445	SMB/CIFS	4-8min
TCP 63422	Bigfix	4-8min
TCP 53	DNS	4-8min
TCP 80	HTTP	8-15min
UDP 63422	Bigfix	8-15min
TCP SYNs	many	> 15min

Figure 13: Protocol composition for unicast traffic based on TCP and UDP ports, ranked by `ts_50`.

down by session-layer protocol with an additional mapping from ports in Table 13. Unfortunately, unlike the case of broadcast and multicast, with unicast, it is harder to deduce the ultimate purpose for much of this traffic since even the session or application-level protocol identifiers are fairly generic. (One exception is the "BigFix" application listed in Fig. 13. BigFix is an enterprise software patching service that checks security compliance of enterprise machines; based on the frequency and volume of BigFix traffic we see, it appears to have been configured by an over-zealous system administrator.)

Stymied in our attempts to deconstruct unicast traffic based on whether and how it might be proxied, we try

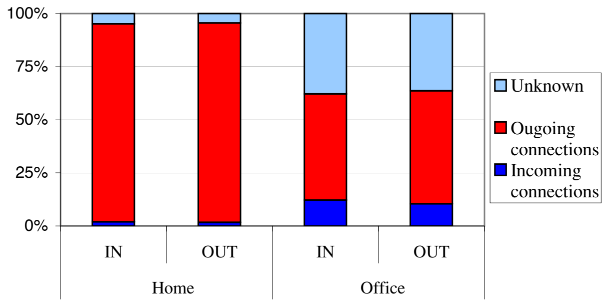


Figure 14: Fraction of packets generated by incoming vs. outgoing connections. For home and office, both received and transmitted packets.

an alternate strategy. We classify TCP and UDP packets based on the connections they belong to and categorize connections as incoming vs. outgoing. Our interest in this classification is because we suspect that a large portion of packets are likely to belong to outgoing connections. And while a host might wake for incoming connections, waking for outgoing connections might well be avoidable (for reasons discussed below). From the results in Fig. 14, we see that outgoing connections do indeed dominate. Now for a sleeping machine, there are three possibilities for these outgoing connections: (1) the connection was initiated by the host before the idle period—in this case, such traffic might not be ignorable if the host/proxy wants to maintain this connection, hence we hope this percentage of traffic is small, (2) the connection was initiated but failed (3) the connection was initiated by the host after the start of the idle period; for a sleeping host, these connections would either simply never have been initiated (if the connection were deemed unncessary) or, the host would be deliberately woken to initiate these connections (if the connection were deemed necessary, as for services scheduled to run during idle times). For the former, the traffic can simply be ignored from our accounting and, in the latter case, such scheduled processing is easily batched and hence needn't disrupt sleep. Hence for all but the first case, waking the machine might be avoidable. We plot this breakdown of outgoing connections in Figure 15. We see that only a relatively small percentage of outgoing connections – always less than 25% – belong to the first category which might require waking the host. Based on this, we speculate that, it might be possible to eliminate or ignore much of even unicast traffic.

Early in this section, we asked whether one might identify a small set of of protocols or proxy behaviors that could yield significant savings. We find that, the answer is positive in the case of multicast and broadcast but less clear for unicast traffic. In the next section we consider the implications of our traffic analysis for proxy design.

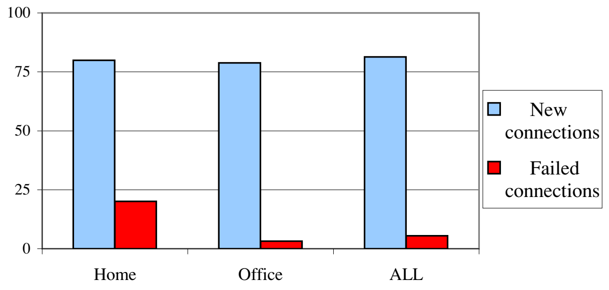


Figure 15: For outgoing connections: the fraction of incoming packets that belong to new connections and failed connection attempts.

5 A Measurement-driven Approach to Proxy Design

Having studied the nature of idle-time traffic, we now apply our findings to the design of a practical power-saving proxy. We start in Section 5.1 by extracting the high-level design implications of our traffic analysis from the previous section. Building on this, in Section 5.2, we illustrate the space of design tradeoffs by considering four specific examples of proxies. In Section 5.3, we distill our findings into a proposal for a core proxy architecture that offers a single framework capable of supporting the broad design space we identify.

5.1 Design Implications

At minimum, a power-saving proxy should: (a) allow the host to sleep for a significant fraction of the time, and (b) maintain the basic network presence of the host by ensuring remote entities can still address and reach the machine and the services it supports. Beyond this, we have a significant margin of freedom in choosing how a proxy might handle the remaining idle-time traffic and applications. Viewed through this lens, our results from Section 4 lead us to differentiate idle-time traffic along two different dimensions. The first classifies traffic based on the need to proxy the traffic in question:

(1) don't-wake protocols: these are protocols that generate sustained and periodic traffic and hence, ideally, would be dealt with (by a proxy) *without* waking the host since otherwise the host would enjoy little sleep. Examples of such protocols identified in the previous section include IGMP, PIM, ARP. Table 1 lists a set of protocols we classify as don't-wake.

(2) don't-ignore protocols: these are protocols that require attention to ensure the correct operation of higher-layer protocols and applications. For example, we must ensure the DHCP lease on an IP address must be maintained and that a machine must respond to NetBIOS name queries to ensure the services it runs over NetBIOS remain addressable. The protocols we identified as don't-ignore are listed in Table 1. Note that the list of don't-wake and don't-ignore protocols need not be mutually

Don't wake	HSRP, ARP, PIM, NBDGM, ICMP, IGMP, SSDP
Don't ignore	ARP (for me), NBNS, DHCP (for me)

Table 1: Protocols that shouldn't cause a wake up (too expensive in terms of sleep), and protocols that should not be ignored (for correctness).

Ignorable	HSRP, PIM, ARP (for others), IPX, LLC, EIGRP, DHCP	
Mechanical Response	Protocol	State
	ARP	IP address
	NBNS	NB names of machine and local services
	SSDP	Names of local plug-n-play services
	IGMP	Multicast groups the interface belongs to
	ICMP	IP address
	NBDGM	NB names of machine and local services. Ignores pkts. not destined to host, wakes host for rest

Table 2: Protocols that can be handled by ignoring or by mechanical response. We classify DHCP as ignorable because we choose to schedule the machine to wake up and issue DHCP requests to renew the IP lease – an infrequent event.

exclusive; for example, ARP traffic is both frequent and critical and hence falls under both categories.

(3) policy-dependent traffic: for the remainder of traffic, the choice of whether and how a proxy should handle the traffic is a matter of the tradeoff the user (or software designer) is seeking to achieve between the sophistication of idle-time functionality, the complexity of the proxy implementation and energy savings. We shall explore these tradeoffs in the context of concrete proxy implementations in Section 5.2.

A complementary dimension along which we can classify traffic is based on the complexity required to proxy the traffic in question:

(A) ignorable (drop): this is traffic that can safely be ignored. Section 4 identified several such protocols and the top ranked. of these are listed in Table 2. Comparing Tables 1 and 2, we see that (fortunately) there is a significant overlap between `don't-wake` and ignorable protocols. Policy-dependent traffic/applications that are deemed unimportant to maintain during idle times could likewise be ignored while `don't-ignore` protocols obviously cannot be.

(B) handled via mechanical responses: this includes incoming (outgoing) protocol traffic for which it is easy to construct the required response (request) using little-to-no state transferred from the sleeping ho.nction is somewhat subjective, based For example, a proxy can easily respond to NetBIOS Name Queries asking about local NetBIOS services, once these services are known by the proxy. Table 2 lists key protocols that can be dealt with through mechanical responses.

(C) require specialized processing: this covers protocol traffic that, if proxied, would require more complex state maintenance (transfer, creation, processing and update) between the proxy and host. For example, consider a proxy that takes on the role of completing ongoing p2p downloads on behalf of a sleeping host – this requires that the proxy learn the status of ongoing and scheduled downloads, the addresses of peers, *etc.* and moreover that the proxy appropriately update/transfer state at the host once it resumes. In theory, specialized processing would be attractive for `policy-dependent` traffic that is both important and frequently-occurring (since otherwise we could simply drop unimportant traffic and wake the host to process infrequent traffic).

Of course, in addition to the the above (classes A-C), for traffic that a proxy doesn't ignore but doesn't want/know to handle a proxy always has the option of waking the host. Essentially the decision of whether to handle desired traffic in the proxy versus waking the host represents a tradeoff between the complexity of a proxy implementation and the sleep time of hosts.

5.2 Example Proxies

We now present four concrete proxy designs derived from the distinctions drawn above. We select these proxies to be illustrative of the design tradeoffs possible but also representative of practical and useful proxy designs.

proxy_1 We start with a very simple proxy that: (1) ignores all traffic listed as ignorable in Table 2 and (2) wakes the machine to handle all other incoming traffic. Besides clearly ignorable protocols, we choose to also ignore traffic generated by the Bigfix application (TCP port 63422) , which we previously identified (Section 4) to be one of the *big offenders*. We do so because this traffic is a) not representative for non-Intel machines, and b) the application is very badly configured – sending very large amounts of traffic for little offered functionality – making sleep almost impossible.

This proxy is simple – it requires no mechanical or specialized processing. At the same time, because it makes the conservative choice of waking the host for all traffic not known to be safely ignorable, this proxy is fully *transparent* to users and applications, in the sense that the effective behavior of the sleeping machine is never different from had it been idle (except for the performance penalties due to the additional wake-up time).

proxy_2 Our second proxy is also fully transparent, but takes on greater complexity in order to reduce the frequency with which the machine must be woken. This proxy: (1) ignores all traffic listed as ignorable in Table 2, and (2) issues responses for protocol traffic listed in the same table as to be handled with mechanical responses and (3) wakes the machine for all other incoming traffic. Since this proxy needs more state to generate mechanical responses (*e.g.,* the NetBIOS Names of local services, needed to answer NBNS queries), it can also use this extra information to selectively ignore more packets than `proxy_2` (*e.g.,* ignore all NetBIOS datagrams not destined for local services).

proxy_3 Our third proxy generates even deeper savings by only maintaining a small set of applications, (chosen by the user) operable during idle times, while ignoring all other traffic. We use telnet, ssh, VNC, SMB file-sharing and NetBIOS as our applications of interest. This proxy performs the same actions (1) and (2) as implemented by `proxy_2` (ignore and responds to the same set of protocols), but it (3) wakes up for all traffic belonging to any of telnet, ssh, VNC, SMB file-sharing and NetBIOS and (4) drops any other incoming traffic. Relative to our previous example, `proxy_2` is less transparent in that the machine appears not to be sleeping for some select remote applications, but is inaccessible to all others.

proxy_4 All the above proxies implement functionality related to handling incoming packets. In our final proxy, we also consider waking up for scheduled tasks initiated locally. This proxy behaves identically to `proxy_3` with respect to incoming packet, but supports an additional action: (5) wake up for the following tasks (for which we assume that the system is configured to wake up in order to perform them): regular network backups, anti-virus (McAfee) software updates, FTP traffic for automatic software updates, and Intel specific updates.

Evaluating tradeoffs In the following we compare the sleep achievable by our 4 proposed proxies, and compare it with the baseline WoP case. We perform this evaluation for both office and home environments, and in each case we evaluate 3 possible values for transition times ts: 5, 10, and 60 seconds. The first of these (5s) is a very optimistic transition time, not achievable today using S3 sleep states, but foreseeable in the near future (today, Microsoft Vista specifications require computers to resume from S3 sleep in under 2s [18]). The second (10s) is representative of the shortest transitions achievable today [6], and the last (1min) is representative of a setting that allows almost a minute for processing subsequent relevant network packets before going to sleep again. The advantage of using a very short timer before going to sleep is the increased achievable sleep. The disadvantage is that the delay penalty for waking the host will be incurred at more packets. In the extreme case of very short sleep timers, this could make remote applications sluggish and un-responsive. For the wake events generated by scheduled tasks, we use a longer transition time (and thus a longer sleep timer value) of 1min, since such tasks usually take longer time to complete.

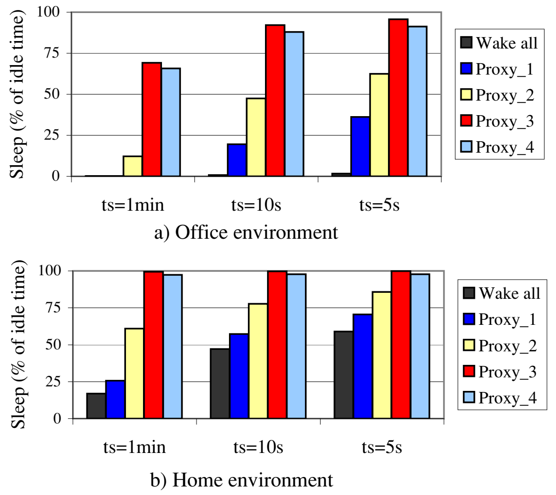


Figure 16: Savings achieved by different proxies in home and office environments.

Examining the performance of our proxies, we make the following high-level observations: *a)* At one end of the spectrum, `proxy_1`(the simplest) is inadequate in office environments, and borderline adequate in home environments. *b)* At the other end of the spectrum we have `proxy_3`, which only handles a select number of applications, but in return achieves good sleep in all scenarios – more than 70% of idle time even in the office and with a transition time of 1minute. *c)* The efficiency of proxy 2 depends heavily on environment. While the additional complexity (compared to `proxy_1`) makes it a good fit in home environments (sleeping close to 60% even for $ts = 1min$), having to handle all traffic makes it a worse fit for the office (sleeping $\approx 12\%$ for the same transition time). This shows that, unless they support a large number of rules, transparent proxies are a better fit for home, but not the office. *d)* The best tradeoff between functionality and savings, and therefore the appropriate proxy configuration, depends on the operating environment. *e)* Since scheduled wake-ups are typically infrequent, the impact they have on sleep is minimal – in our case, `proxy_4` sleeps almost as much as `proxy_3` in all considered scenarios.

5.3 A strawman proxy architecture

Our study leads us to propose a simple proxy architecture that offers a unified framework within which we can accommodate the multiplicity of design options identified above. The proposal we present is a high-level one since our intent here is merely to provide an initial sketch of an architecture that could serve as the starting point for future discussion on standardization efforts.

The core of our proposal is a table—the power-proxy table (PPT)—that stores a list of *rules*. Each rule describes the manner in which a specified traffic type should be handled by the proxy when idle. A rule consists of a *trigger*, an *action* and a *timeout*.

Triggers are either timer events or regular expressions describing some network traffic of interest. When a trigger's timer event fires or if an incoming packet matches a trigger's regular expression, the proxy executes the corresponding action. If the action involves waking the host, the timeout value specifies the minimum period of time for which the host must stay awake before contemplating sleep again. To resolve multiple matching rules, standard techniques such as ordering the rules by specificity, policy, *etc.* can be used. The proxy table must also include a *default* rule that determines the treatment of packets that do not match on any of the explicitly enumerated rules.

We propose the following actions:

- drop: the incoming packet is dropped.
- wake: the proxy wakes the host and forwards the packets to it. Other packets buffered while waiting for the wake will be forwarded as well.
- respond(`template, state`): the proxy uses the specified *template* to craft a response based on the incoming packet and some *state* stored by the proxy. This action is used to generate mechanical responses as described below.
- redirect(`handle`): the proxy forwards the packet to a destination specified by the `handle` parameter. This is used to accommodate specialized processing as described below.

A response template is a *function* that computes the mechanical response based on the incoming packet and one or more *immutable* pieces of state. This means that our function does not maintain or change any state. There is no state carried over between successive incoming packets (such as sequence numbers), and no state transfer between the proxy and the host upon wake-up. We choose to support this functionality because a) it is relatively simple to implement in practice and b) it covers most of the non-application specific traffic, as shown in Section 4, and illustrated in our proxy examples.

To accommodate more specialized processing, we assume developers will write application-specific stubs and then enter a redirect rule into the proxy's PPT, where the `handle` specifies the location to which the proxy should send the packet. Such stubs can run on machine accessible over the network (*e.g.,* a server dedicated to proxying for many sleeping machines in a corporate LAN), or on a low-power micro-engine supported on the local host (*e.g.,* a controller on the motherboard, or a USB-connected gumstick). In all these cases, the handle would be specified by its address, for example a (IP address, port) combination. The redirect abstraction thus allows us to accommodate specialized processing without embedding application-specific knowledge into the core proxy architecture.

The external API to this proxy is twofold: (1) APIs to

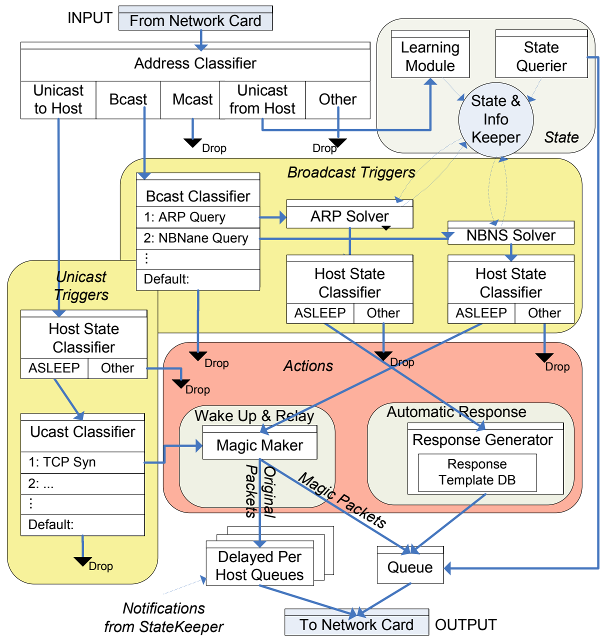


Figure 17: Example Click implementation.

activate/deactivate the proxy as the host enters/exits sleep and (2) APIs to insert and delete rules. The process by which to install and execute stubs is outside of the core proxy specification which only provides the mechanism to register and invoke such stubs. The architecture is agnostic to where the proxy runs allowing implementations in hardware (e.g., at host NICs), in PC software (e.g., a proxy server running on the same LAN) or in network equipment (e.g., a firewall, NAT box).

Finally, the use of timer events to wake a host already exists today. Our contribution here is merely to integrate the mechanism into a unified proxy architecture.

5.4 Proxy Prototype Implementation

To illustrate the feasibility of our architecture, we build a simple proxy prototype using the Click modular router [17]. We choose to deploy the proxying functionality in a standalone machine responsible for maintaining the network presence of several hosts on the same LAN. To allow our proxy (let us call it P) to sniff the traffic for each host, we ensure that P shares the same broadcast domain with these hosts. This can be achieved either by connecting the proxy and the machines to a common network HUB, or by configuring the LAN switch to forward all traffic to the port that serves P.

In our initial design, we don't implement proxies that involve transferring state between the host and the proxy. Instead, P learns the pieces of state required (e.g. the IP address and the Netbios name for each host) by sniffing host traffic and extracting the state exchanged (e.g. ARP and NBNS exchanges). This design circumvent the need for any end-host modifications, and support proxying for machines with different hardware platforms (new and old) and operating systems. The proxy requires minimal configuration (a list of the MAC addresses of the hosts that need to be proxied), and can be incrementally deployed as a low-power stand-alone network box. Once low-power proxying standards are developed [12], the design can be extended to support state transfer, and achieve even deeper energy savings.

Our prototype implements very basic proxying functionality, but the software architecture (presented in Figure 17) can be easily extended to more protocols and use cases. Currently, we support three types of actions: *wake*, *respond* and *drop*. The proxy awakes its hosts for TCP connection requests (incoming TCP SYN packets) and incoming Netbios Name Queries for the host's NB name. If such a "wake packet" for a sleeping host arrives, P buffers the request, sends a magic packet to wake the host, and relays the buffered packet once the host becomes available. The proxy responds automatically to incoming ARP requests, and drops all other incoming packets. In relation to the examples discussed in Section 5.2, this prototype has a *simple* and *non-transparent* design. To determine whether a host is awake, the proxy sends periodic ARP queries to each host; if these queries receive no response, the host is assumed to be asleep. When the proxy attempts to wake a host and fails repeatedly, the host is assumed to be off, rather than just asleep, and the proxy ceases to maintain its network presence.

Figure 17 presents the software architecture of our Click proxy, and highlights the mapping between Click modules and the generic categories of *triggers*, *actions* and *state*, discussed in the strawman proxy architecture.

We test our Click-based proxy implementation by installing it on one of our enterprise desktops, and configuring the proxy to maintain the network presence of several IBM ThinkPad laptops. We use this deployment to measure the delays experienced by applications waking a sleeping host, and find these to be surprisingly low: $2.4s$ on average, and $4s$ at maximum – much lower than the $30s$ TCP SYN timeout. These delays includes the host wake-up delay ($\approx 1.4s$), and the additional time required for the proxy to detect the state change and relay the buffered packet causing the wake ($\approx 1s$). We defer a comprehensive deployment-based evaluation to future work.

6 Power-Aware System Redesign

In this section we consider approaches that might assist in reducing idle-time energy consumption by either simplifying the implementation of proxies or altogether obviating the need for proxying.

6.1 Software Redesign

Our idle traffic analysis shows that solutions relying on Wake-on-LAN functionality face the following challenges: (i) It is difficult to decide if various packets and protocols warrant a machine wake-up.(ii) Hosts receive many packets even when idle (3 per second on average). (iii) Many protocols exchange packets periodically, preventing long quiet periods when hosts could sleep. These challenges could be dealt with at both application and protocol level:

Power-aware application configuration Today, applications and services are typically designed or configured without taking into account their potential impact on the power management at end-systems. For example, in Section 4.4 we discussed a tool called Bigfix, that checks if network hosts conform to Intel's corporate security specifications. This application was configured to perform these checks very aggressively, continuously generating large amounts of traffic. Under a WoL approach, this application alone would have made prolonged sleep virtually impossible.

This is a perfect example of the behaviour that could be avoided by configuring applications to be more power-aware, and perform periodic tasks less frequently, reducing the volume of network traffic seen by hosts.

Protocol Specification The decision to ignore or wake on a packet can be difficult, and involves protocol parsing, maintaiing a long set of filters and rules, and for some protocols host or application-specific state.

To eliminate the complexity of this decision, and allow hosts to sleep longer even when using very simple rules for waking, protocols could be augmented to carry explicit power-related information in their packets. An example of such information would be a simple bit indicating whether a packet can be ignored.

Protocol Redesign We believe these principles should be followed when designing power-aware protocols.

Consideration when using broadcast and multicast: We saw earlier that broadcast and multicast are mainly responsible for keeping hosts awake. This type of traffic could be substantially reduced by redesigning protocols to use broadcasts sparingly. Some protocols are particularly inefficient in this respect. For example, all NetBIOS datagrams are always sent over Ethernet broadcast frames. These frames are received by all hosts on the LAN, and then discarded by most of them. This ranks NBDGM as one of the top "offenders", yet this could be easily avoided by using unicast transmissions when possible. Another approach is based on the observation that many service discovery protocols have redundant functionality. This redundant functionality could conceivable be replaced by a single service that can be shared by a multiplicity of applications.

Synchronization of periodic traffic: One way to increase the number of long periods of network quiescence would be to identify protocols that use periodic updates/message exchanges, and try to synchronize, or bulk these exchanges together. This would allow machines to periodically wake up, process all notifications and request, and resume sleep.

Complementing soft state: Many protocols (*e.g.,* SSDP, NetBIOS, etc.) maintain and update state using periodic broadcast notifications/ For such protocols (and for similar applicatios), it would be essential to make them disconnection-tolerant, by providing complementary *state query* mechanisms that could be used quickly build up-to-date copies of the soft state upon waking. This would enable ignoring any soft state notifications. Today, such query mechanisms exist only for some of these protocols, and they are often inefficient.

6.2 Hardware Redesign

A general goal of energy saving mechanisms, especially hardware designs, is to lead the industry towards energy proportional computing [8]. If energy consumption of a machine would accurately reflect its level of utilization, the energy would be zero when idle. Sleep states are a step in this direction, P-states (low power active operation) are another. Related to this, it would be very desirable to expose power saving states (S states) that feature better transition times, even if they offer smaller savings. Given the small inter-packet gaps, these states will come in handier than the deep-sleep ones.

7 Related Work

The notion that internetworked systems waste energy due to *idle* periods has been frequently reiterated[14, 13, 16, 19, 10, 7, 15]. Network presence proxying for the purpose of saving energy in end devices was first proposed over ten years ago by Christensen *et al.*; in follow-up work [11] the authors quantify the potential savings using traffic traces from a single dormitory access point and in [13] examine the traffic received at a single idle machine to identify dominant protocols and discuss whether these can be safely ignored. Our work draws inspiration from this early work extending it with a large-scale and more in-depth evaluation of idle-time traffic in enterprise and home environments. A more recent proposal [7]. postulates the notion of "selective connectivity", whereby a host can dictate or manage its network connectivity, going to sleep when it does not want to respond to traffic.

There is an extensive literature on energy saving techniques for individual PC platforms. Broadly, these aim for reduced power draws at the hardware level and faster transition times at the system level. These offer a complementary approach to reducing the power draw of idle machines; if and when these techniques lead us to perfectly "energy-proportional" computers, the idle-time consumption will be less problematic and proxying will fade in importance. So far however, achieving such energy proportionality has proved challenging.

In parallel work [6], the authors build a prototype proxy supporting BIT-TORRENT and IM as example applications. Our work considers a broader proxy design space, evaluating the tradeoffs between design options and the resultant energy savings informed by detailed analysis of network traffic. In relation to our design space, their proxy supports BT and IM using application stubs.

8 Conclusions

In general, the question of how a proxy should handle the user-idle time traffic presents a complex tradeoff between balancing the complexity of the proxy, the amount of energy saved, and the sophistication of idle-time functionality. Through the use of an unusual dataset, collected directly on endhosts, we explored the potential savings, requirements and effectiveness of technologies that aim to put endhost machines to sleep when users are idle. For the first time here, we dissect the different categories of traffic that are present during idle times, and quantify which of them have traffic arrival patterns that prevent periods of deep sleep. We see that broadcast and multicast traffic constitute a substantial amount of the background chatter due to service discovery and routing protocols. Our data also revealed a significant amount of outgoing connections, generated in part by enterprise applications. We tried to identify which traffic can be ignored and found that most of the broadcast and multicast traffic, as well as roughly 75% of outgoing connections, appears safely ignorable. Handling unicast traffic is more involved because it harder to infer the intent of such traffic, and often needs some state information to be maintained on the proxy.

After having studied our traffic and the sleep potential those patterns contain, we discuss the design space for proxies, and evaluate the savings offered by 4 sample proxy designs. These cases reveal the tradeoffs between design complexity, available functionality and energy savings, and discuss the appropriateness of various design points in different use environments, such as home and office.

Finally, we present a general and flexible strawman proxy architecture, and we build an extensible Click-based proxy that exemplifies one way in which this architecture can be implemented.

Aknowledgments

We thank Robert Hays, Ken Christensen, Gianluca Iannaccone, Eric Mann, Rabin Patra and Kevin Fall for their suggestions, and Eve Schooler for her help collecting trace data. We also thank the anonymous reviewers and our shepherd Yuanyuan Zhou for their useful feedback.

References

[1] Alert Standard Format (ASF) Specification, v2.0, DSP0136, Distributed Management Task Force. `http://www.dmtf.org/standards/asf`.

[2] ENERGY STAR Program Requirement for Computers. Version 4.0. `http://www.eu-energystar.org`.

[3] The Wireshark Network Protocol Analyzer. `http://www.wireshark.org/`.

[4] Power and Thermal Management in the Intel Core Duo Processor. In *Intel Technology Review, Vol.10*, 2006.

[5] Advanced configuration and power interface. `http://www.acpi.org`.

[6] Y. Agarwal, S. Hodges, J. Scott, R. Chandra, P. Bahl, and R. Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *NSDI*, 2009.

[7] M. Allman, K. Christensen, B. Nordman, and V. Paxson. Enabling an energy-efficient future internet through selectively connected end systems. In *HotNets*, 2007.

[8] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[9] BRO IDS. `http://www.bro-ids.org`.

[10] K. J. Christensen and F. B. Gulledge. Enabling power management for network-attached computers. *International Journal of Network Management*, 1998.

[11] K. J. Christensen, C. Gunaratne, B. Nordman, and A. D. George. The next frontier for communications networks: power management. *Computer Communications*, 27(18):1758–1770, 2004.

[12] ECMA International. TC32-TG21 – Proxying Support for Sleep Modes. `http://www.ecma-international.org/memento/TC32-TG21-M.htm`.

[13] C. Gunaratne, K. Christensen, and B. Nordman. Managing Energy Consumption Costs in Desktop PCs and LAN Switches with Proxying, Split TCP Connections, and Scaling of Link Speed. *International Journal of Network Management*, October 2005.

[14] M. Gupta and S. Singh. Greening of the internet. In *ACM SIGCOMM, Karlsruhe, Germany*, August 2003.

[15] Intel remote wake technology. `http://support.intel.com/support/chipsets/rwt/`.

[16] J. Klamra, M. Olsson, K. Christensen, and B. Nordman. Design and implementation of a power management proxy for universal plug and play. *Proceedings of the Swedish National Computer Networking Workshop (SNCW)*, Sep 2005.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[18] Microsoft Window Vista Logo Program Requirements and Policies. `http://www.microsoft.com/whdc/winlogo/hwrequirements.mspx`.

[19] B. Nordman. Networks, Energy, and Energy Efficiency. *Cisco Green Research Symposium*, March 2008.

[20] C. Webber, J. Roberson, M. McWhinney, R. Brown, M. Pinckard, and J. Busch. After-hours power status of office equipment in the usa. *Energy*, 31(14):2823–2838, Nov 2006.

[21] B. Zhai, D. Blaauw, D. Sylvester, and K. Flautner. Theoretical and practical limits of dynamic voltage scaling. In *DAC*, 2004.

Wishbone: Profile-based Partitioning for Sensornet Applications

Ryan Newton, Sivan Toledo, Lewis Girod, Hari Balakrishnan, and Samuel Madden
MIT CSAIL

Abstract

The ability to partition sensor network application code across sensor nodes and backend servers is important for running complex, data-intensive applications on sensor platforms that have CPU, energy, and bandwidth limitations. This paper presents Wishbone, a system that takes a dataflow graph of operators and produces an optimal partitioning. With Wishbone, users can run the same program on a range of sensor platforms, including TinyOS motes, smartphones running JavaME, and the iPhone. The resulting program partitioning will in general be different in each case, reflecting the different node capabilities. Wishbone uses *profiling* to determine how each operator in the dataflow graph will actually perform on sample data, without requiring cumbersome user annotations. Its partitioning algorithm models the problem as an integer linear program that minimizes a linear combination of network bandwidth and CPU load and uses program structure to solve the problem efficiently in practice. Our results on a speech detection application show that the system can quickly identify good trade-offs given limitations in CPU and network capacity.

1 Introduction

An important class of sensor computing applications are data-intensive, involving multiple embedded sensors each sampling data at tens or hundreds of kilohertz and generating many megabytes per second in aggregate. Examples include acoustic localization of animals, gunshots, or speakers; structural monitoring and vibration analysis of bridges, buildings, and pipes; object tracking in video streams, etc. Over the past few years, impressive advances in sensor networking hardware and software have made it possible to prototype these applications. However, two challenges confront the developer who wants to deploy and sustain these applications:

- **Heterogeneity:** Thanks to hardware advances, one can run these applications on a variety of embedded devices, including "motes", smartphones (which themselves are varied), embedded Linux devices (e.g., Gumstix, WiFi access points), etc. This richness of hardware and software is good because it allows the developer to pick the right platforms for a task and evolve the infrastructure with time. On the other hand, it poses a software nightmare because it requires code to be developed multiple times, or ported to different platforms.
- **Decomposition:** A simple way of designing such systems would deliver all the gathered data to a central server, with all the computation running there. This approach may consume an excessive amount of bandwidth and energy. A different approach is to run all of the computation "in the sensor network", but often the computational capabilities of the sensor nodes are insufficient. The question is: how best to partition an application between the server(s) and the embedded nodes? Improper partitioning can lose important data, waste energy, and may cause applications to simply not work as desired.

No current solution addresses both of these challenges. To support heterogeneity, one might be able to write programs in a language like Java. Unfortunately, some platforms do not support Java, or may not support it in its full generality; in addition, Java virtual machines for embedded devices are of uneven quality. More importantly, it is difficult to partition such a program in a way that will perform well on any given platform without a significant amount of tuning and manual optimization. That, in turn, limits the ability to swap out the underlying hardware platform, or even to move computation between the embedded nodes and servers.

We have developed Wishbone, a system that allows developers to achieve both goals for applications that satisfy two conditions:

- Streaming dataflow model: The application should be written as a stream-oriented collection of operators configured as a dataflow graph.
- Predictable input rates and patterns: The input data rates at the sensors gathering data don't change in unpredictable ways.

To use Wishbone, the developer writes a program in a high-level stream-processing language, WaveScript [16], which has a common runtime for both embedded nodes and servers. We have extended our open-source WaveScript compiler to produce efficient code for several embedded platforms: TinyOS 2.0, smartphones running Java J2ME, the iPhone, Nokia tablets, various WiFi access points, and any POSIX compliant platform supporting GCC. These platforms are sufficiently diverse that generating high-performance native code from a shared high-level language is itself a challenge. Fortunately, we have an advantage in WaveScript's domain-specificity: the compiler has additional information that it can use to optimize programs for specific streaming workloads.

We have used WaveScript in several applications, including: locating wild animals with microphone arrays, locating leaks in water pipelines, and detecting potholes in sensor-equipped taxis. For the purposes of this paper, we chose to focus on two applications that highlight the program partitioning features of Wishbone: a speech detector that identifies when a person is speaking in a room and a 22-channel EEG application. Each is based on an application currently in use by our group (EEG) or by other groups (speaker detection). Both were ported[1] to WaveScript for the evaluation in this paper.

The key function of Wishbone is, given a WaveScript-produced dataflow graph of stream operators, to partition it into in-network and server-side components. It uses a *profile-driven approach*, where the compiler executes each operator against programmer-supplied sample data, using real embedded hardware or a cycle-accurate simulation. After profiling, we are able to estimate the CPU and communication requirements of every operator on every platform. Wishbone depends on this sample data being representative of the actual input the sensor will see during deployment; we believe this is a valid assumption and justify it in our experiments.

Determining a good partitioning is difficult even after one uses a profiler to determine the computational and network load imposed by each operator. Wishbone models the partitioning problem as an integer linear program (ILP), seeking to minimize a combination of network bandwidth and CPU consumption subject to hard upper bounds on those resources. With these criteria, our ILP formulation will find optimal solutions—and although ILP is an NP-hard problem, in practice our implementation can partition dataflow graphs containing over a thousand operators in a few seconds.

Our results show that the system can quickly identify the optimal partition given constraints on CPU and network capacity. And picking the right partition matters. In our evaluation, our weakest platform got 0% of speaker detection results through the network successfully when doing all work on the server, and 0.5% when doing all work at the node. We can do 20× better by picking the right intermediate partition. Because the optimal partitioning changes depending on the hardware platform and the number of nodes in the network, manual partitioning is likely to be tedious at best. For larger graphs (such as our 1412 node electroencephalography (EEG) application), doing the partitioning by hand with any degree of confidence becomes extremely difficult.

Finally, we note that we do not intend that Wishbone be used only as a completely automated partitioning tool, but also as a part of an interactive design process with the programmer in the loop. In addition to recommending partitions, Wishbone can find situations in which there is no feasible partitioning of a program; e.g., because the bandwidth requirements will always exceed available network bandwidth, or because there are insufficient CPU resources to place bandwidth-reducing portions of the program inside the sensor network. In these cases, the programmer will have to either switch to a more powerful node platform, reduce the sampling rates or the number of sensors, or be willing to run the network in an overload situation where some samples are lost. In the overload case, Wishbone can compute how much the data rates need to be reduced to achieve a viable partition.

[1]WaveScript is an imperative language with a C-like syntax. An initial port of an application from C/C++ is very quick: cut, paste, and clean it up. Refactoring to expose the parallel/streaming structure of the application may be more involved.

```
fun FIRFilter(coeffs, strm) {
   N = Array:length(coeffs);
   fifo = FIFO:make(N);
   for i = 1 to N-1 { FIFO:enqueue(fifo, 0) };
   iterate x in strm {
       FIFO:enqueue(fifo, x);
       sum = 0;
       for i = 0 to N-1 {
         sum += coeffs[i] * FIFO:peek(fifo, i);
       };
       FIFO:dequeue(fifo);
       emit sum;
   }
 }
fun LowFreqFilter(strm) {
  evenSignal = GetEven(strm);
  oddSignal  = GetOdd (strm);
  // even samples go to one filter, odds the other:
  lowFreqEven = FIRFilter(hLow_Even, evenSignal);
  lowFreqOdd  = FIRFilter(hLow_Odd,  oddSignal);
  // now recombine them
  AddOddAndEven(lowFreqEven, lowFreqOdd)
 }
fun GetChannelFeatures(strm) {
  low1 = LowFreqFilter(strm);
  low2 = LowFreqFilter(low1);
  low3 = LowFreqFilter(low2);

  high4 = HighFreqFilter(low3); // we need this
  low4  = LowFreqFilter(low3);
  level4 = MagWithScale(filterGains[3], high4);

  high5 = HighFreqFilter(low4); // and this one
  low5  = LowFreqFilter(low4);
  level5 = MagWithScale(filterGains[4], high5);

  high6 = HighFreqFilter(low5); // and this one
  level6 = MagWithScale(filterGains[5], high6);
  zipN([level4, level5, level6]);
 }
```

Figure 1: Excerpts from running code in EEG-application. The "low level" FIRFilter function constructs new dataflow operators using `iterate`. FIRFilter is stateful because it maintains and modifies `fifo`. Higher level functions such as LowFreqFilter and GetChannelFeatures wire together a larger graph.

2 Language and front-end compiler

The developer writes a program in WaveScript that constructs a dataflow graph of stream operators. Each operator consists of a work function and optional private state. The job of the WaveScript front-end compiler is to partially evaluate the program to create the dataflow graph, whereas the WaveScript backend performs graph optimizations and reduces work functions to an intermediate language that can be fed to a number of backend code generators. Each work function contains an imperative routine that processes a single stream element, updates the private state for that dataflow operator, and produces elements on output streams. (Later, we will single out *stateless* operators that maintain no mutable state between invocations.)

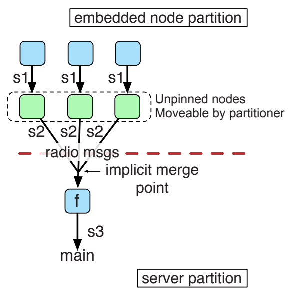
A WaveScript source program can manipulate streams as values and thereby wire together operator graphs, as seen in Figure 1. The example in Figure 1 contains psuedocode that wires together the cascading filters found in one of the 22-channels of our EEG application. The evaluation of the `iterate` form creates a new dataflow operator and provides its work function. The return value of an `iterate` is its output stream. For example, the function `FIRFilter` in Figure 1 takes a stream as one of its inputs and returns a stream. Within the body of the `iterate` the `emit` keyword produces elements on the output stream. The equal (`=`) operator introduces new variables and the last expression in a `{...}` block is its return value. Type annotations are unnecessary.

2.1 Program Distribution

Thus far, our description applies to WaveScript programs that run on a single node. To support distributed execution, we extended the language to allow developers to specify which part of the dataflow graph should be replicated on all embedded nodes. This specification is *logical* rather than *physical*; the physical locations of operators are computed by Wishbone's partitioner using the programmer's annotations and profiler data.

To create the logical specification in Wishbone, the user places a subset of the program's top-level stream bindings in a `Node{}` namespace. All operators in the `Node{}` namespace are replicated once per embedded node. This separation is particularly important for stateful operators, because stateful operators in the `Node` partition have an instance of their state for every node in the network. Stateful operators on the server side are instantiated only once.

As an example, consider the code snippet in Figure 2, which shows a node/server program that samples data from the microphone and filters it. The operator `readMic`, producing the stream `s1`, must reside on each node, as it samples data from hardware only available on the embedded node. Because the `filtAudio` call producing `s2` is in the `Node` partition, its operators will be replicated once per node, but can be physically placed either on the embedded node or the server, depending on what the partitioner determines would be best. If `filtAudio` creates stateful operators, their state will need to replicated once per node, regardless of where they are placed. This example illustrates the basic repartitioning model, and shows that, while the system is free to move some operators, there are certain *relocation constraints* the partitioner must respect, discussed in the next section.



```
namespace Node {
  s1 = readMic(...)
  s2 = filtAudio(s1)
}
s3 = f(s2)
main = s3
```

Figure 2: A program skeleton specifying a replicated stream computation across all embedded nodes.

2.1.1 Relocation Constraints

Operators are classified as *movable* or *pinned* as follows. First, operators with side-effects—for example, OS-specific foreign calls to sample sensors and blink LEDs—are pinned to their partition. Likewise, operators on the server that print output to the user or to a file are pinned. Stateless operators without side-effects are not pinned and are always moveable, allowing them to be moved into the other partition if the system determines that to be advantageous. Finally, stateful operators are treated differently for the node and server partitions. It is not generally possible to move stateful server operators into the network—they have a serial execution semantics and a single state instance. However, it is possible to move stateful operators from the node partition to the server. The state of the operator is duplicated in a table indexed by node ID. Thus, a single server operator can emulate many instances running within the network.

Relocating stateful operators in this way raises a different issue—message loss on wireless links. Operators in the node partition may safely assume that all edges between the raw sensors and themselves are *lossless*. Relocating an operator to the server means putting potential data loss upstream of it that was not there previously. Stateless operators are insensitive to this kind of loss because they process each element without any memory of preceding elements, but stateful operators may perform erratically in the face of unexpected missing data, unless they have been intentionally engineered to tolerate it.

Because tolerance to data loss in stateful operators is an application-specific issue, Wishbone supports two operational modes that can be specified by the programmer at compile time. In *conservative* mode it will not relocate stateful operators onto the server, refusing to add lossiness to a previously lossless edge. In *permissive* mode, the system will automatically perform these relocations. In the future, it would be possible to extend the system to make many finer distinctions, such as labeling individual edges as loss-tolerant, or grouping operators together in blocks that cannot be divided by a lossy edge.

2.1.2 Restrictions

The system we present in this paper targets a restricted domain: first, because we focus on a specific dataflow model and, second, because of limitations of our current implementation. (Section 9 will discuss generalizing and extending the model.) Presently, our implementation requires that any path through the operator graph connecting a data source on the node to a data sink on the server may only cross the network once. The graph partitioning algorithm in Section 4 does, however, support back-and-forth communication. The reason for the restriction is that we haven't yet implemented arbitrary communication for all of our platforms. Note that this does not rule out all communication from the server to the nodes, it is still possible, for example, to have configuration parameters sent from a server to in-network operators.

We make the best of this restriction by leveraging it in a number of ways. As we will see, it enables a simplified version of the partitioning algorithm. It can also further filter the set of moveable operators as described in Section 2.1.1, because pinning an operator pins all up- or down-stream operators (can't cross back).

3 Profile & Partition

The WaveScript compiler, implemented in the Scheme language, can profile stream graphs by executing them directly within Scheme during compilation (using sample input traces). This produces platform-independent data rates, but cannot determine execution time on embedded platforms. For this purpose, we employ a separate profiling phase on the device itself, or on a cycle-accurate simulator for its microprocessor.

First, the partitioner determines what operators might possibly run on the embedded platform, discounting those that are pinned to the server, but including movable operators together with those that are pinned to the node. The code generator emits code for this partition, inserting timing statements at the beginning and end of each operator's work function, and at emit statements, which represent yield points or control transfers downstream.

The partition is then executed on simulated or real hardware. The inserted timing statements print output to a debug channel read by the compiler. For example, we execute instrumented TinyOS programs either on TMote Sky motes or by using the MSPsim simulator[2]. In either case, timestamps are sent through a real or virtual USB serial port, where they are collected by the compiler.

For most platforms, the above timestamping method is sufficient. That is, the only relevant information for partitioning is how long each operator takes to execute on that platform (and therefore, given an input data rate, the percent CPU consumed by the operator). For TinyOS, some additional profiling is necessary. To support subdividing tasks into smaller pieces, we must be able to perform a reverse mapping between points in time (during an operator's execution) and points in the operator's code. Ideally, for operator splitting purposes, we would recover a full execution trace, annotating each atomic instruction with a clock cycle. Such information, however, would be prohibitively expensive to collect. We have found it is sufficient to instead simply time stamp the beginning and end of each `for` or `while` loop, and count loop iterations. As most time is spent within loops, and loops generally perform identical computations repeatedly, this enables us to roughly subdivide execution of an operator into a specified number of slices.

After profiling, control transfers to the partitioner. The movable subgraph of operators has already been determined. Next, the partitioner formulates the partitioning problem in terms of this subgraph, and invokes an external solver (described in Section 4) to identify the optimal partition. The program graph is repartitioned along the new boundary, and code generation proceeds, including generating communication code for cut edges (e.g., code to marshal and unmarshal data structures). Also, after profiling and partitioning, the compiler generates a visualization summarizing the results for the user. The visualization, produced using the well-known GraphViz tool from AT&T Research, uses colorization to represent profiling results (cool to hot) and shapes to indicate which operators were assigned to the node partition.

4 Partitioning Algorithms

In this section, we describe Wishbone's algorithms to partition the dataflow graph. We consider a directed acyclic graph (DAG) whose vertices are stream operators and whose edges are streams, with edge weights representing bandwidth and vertex weights representing CPU utilization or memory footprint. We only include vertices that can move across the node-server partition; i.e., the movable subset. The server is assumed to have infinite computational power compared to the embedded nodes, which is a close approximation of reality.

The partitioning problem is to find a cut of the graph such that vertices on one side of the cut reside on the nodes and vertices on the other side reside on the server. The bandwidth of a given cut is measured as the sum of the bandwidths of the edges in the cut. An example problem is shown in Figure 3.

Unfortunately, existing tools for graph partitioning are not a good fit for this problem. Tools like METIS [12] or Zoltan [7] are designed for partitioning large scientific codes for parallel simulation. These are heuristic solutions that generally seek to create a fixed number of

[2]We also tried Simics and msp430-gdb for simulation, but MSPsim was the easiest to use. Note that TOSSIM is not appropriate for performance modeling.

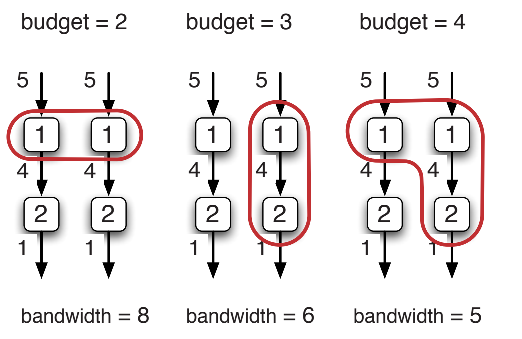


Figure 3: Simple motivating example. Vertices are labeled with CPU consumed, edges with bandwidth. The optimal mote partition is selected in red. This partitioning can change unpredictably, for example between a horizontal and vertical partitioning, with only a small change in the CPU budget.

balanced graph partitions while minimizing cut edges. Newer tools like Zoltan support unbalanced partitions, but with a specified ratios, not allowing unlimited and unspecified capacity to the server partition. Further, they expect a single weight on each edge and each vertex. They cannot support a situation where the cost of a vertex *changes* depending on the partition is it placed in. This is the situation we're faced with: diverse hardware platforms that not only have varying capacities, but for which the *relative* cost of operators varies (for example, due to a missing floating point unit).

We may also consider traditional task scheduling algorithms as a candidate solution to our partitioning problem. These algorithms assign a directed graph of tasks to processors, attempting to minimize the total execution time. The most popular heuristics for this class of problem are variants of *list scheduling*, where tasks are prioritized according to some metric and then added one at a time to the working schedule. But there are three major differences between this classic problem and our own. First, task-scheduling does not directly fit the nondeterministic dataflow model, as no conditional control flow is allowed at the task level—all tasks execute exactly once. Second, task-scheduling is not designed for vastly unequal node capabilities. Finally, schedule length is not the appropriate metric for streaming systems. Schedule length would optimize for *latency*: how fast can the system process one data element. Rather, we wish to optimize for throughput, which is akin to scheduling for a task-graph repeated ad infinitum.

Thus we have developed a different approach. Our technique first preprocesses the graph to reduce the partition search space. Then it constructs a problem formulation based on the desired objective function and calls an external ILP solver. By default, Wishbone currently uses the minimum-cost cut subject to not exceeding the CPU resources of the embedded node or the network capacity of the channel. Cost here is defined as a linear combination of CPU and network usage, $\alpha \cdot CPU + \beta \cdot Net$ (which can be a proxy for energy usage). Therefore we set four numbers for each platform: the CPU/Network resource limits, and coefficients α, β. The user may override these quantities to direct the optimization process.

4.1 Preprocessing

The graph preprocessing step precedes the actual partitioning step. The goal of the preprocessing step is to eliminate edges that could never be viable cut-points. Consider an operator u that feeds another operator v such that the bandwidth from v is the same or higher than the bandwidth on the output stream from u. A partition with a cut-point on the v's output stream can always be improved by moving the cut-point to the stream $u \rightarrow v$; the bandwidth does not increase, but the load on the embedded node decreases (v moves to the server). Thus, any operator that is data-expanding or data-neutral may be merged with its downstream operator(s) for the purposes of the partitioning algorithm, reducing the search space without eliminating optimal solutions.

4.2 Optimal Partitionings

It is well-known that optimal graph partitioning is NP-complete [8]. Despite the intrinsic difficulty of the problem, the problem proves tractable for the graphs seen in realistic applications. Our pre-processing heuristic reduces the problem size enough to allow an ILP solver to solve it exactly within a few seconds to minutes.

4.2.1 Integer Linear Programming (ILP)

Let $G = (V, E)$ be the directed acyclic graph (DAG) of stream operators. For all $v \in V$, the compute cost on the node is given by $c_v > 0$ and the communication (radio) cost is given by r_{uv} for all edges $(u, v) \in E$. One might think of the compute cost in units of MHz (megahertz of CPU required to process a sample and keep up with the sampling rate), and the bandwidth cost in kilobits/s consumed by the data going over the radio. Adding additional constraints for RAM usage (assuming static allocation) or code storage is straightforward in this formulation, but we do not do it here. For each of these costs we can use either mean or peak load (profiling computes both). Because our applications have predictable rates, we use mean load here. Peak loads might be more appropriate in applications characterized by "bursty" rates.

The DAG G contains a set of terminal *source* vertices S, and *sink* vertices T, that have no inward and outward edges, respectively, and where $S, T \subset V$. As noted above, we construct G from the original operator graph such that these boundary vertices are pinned—all the sources must remain on the embedded node; all sinks on the server. Recall that the partitioning problem is to find a single cut of G that assigns vertices to the nodes and server. We can think of the graph G as corresponding to the server and a single node, but vertices assigned to the node partition are instantiated on all physical nodes in the system.

We encode a partitioning using a set of indicator variables $f_v \in \{0,1\}$ for all v in V. If $f_v = 1$, then operator v resides on the node; otherwise, it resides on the server. The pinning constraints are:

$$\begin{array}{l} (\forall u \in S)\ f_u = 1 \\ (\forall v \in T)\ f_v = 0 \\ (\forall v)\ f_v \in \{0,1\}\ . \end{array} \tag{1}$$

Next, we constrain the sum of node CPU costs to be less than some total budget C.

$$cpu \leq C \quad \texttt{where} \quad cpu = \sum_{v \in V} f_v c_v \tag{2}$$

A simple expression for the total cut bandwidth is $\sum_{(u,v)\in E}(f_u - f_v)^2 r_{uv}$. (Because $f_v \in \{0,1\}$, the square evaluates to 1 when the edge (u,v) is cut and to 0 if it is not; $|f_u - f_v|$ gives the same values.) However, we prefer to formulate the integer programming problem as one with a linear rather than quadratic objective function, so that standard ILP techniques can be used.

We can convert the quadratic objective function to a linear one by introducing two variables per edge, e_{uv} and e'_{uv}, which are subject to the following constraints:

$$\begin{array}{l} (\forall (u,v) \in E)\ e_{uv} \geq 0 \\ (\forall (u,v) \in E)\ e'_{uv} \geq 0 \\ (\forall (u,v) \in E)\ f_u - f_v + e_{uv} \geq 0 \\ (\forall (u,v) \in E)\ f_v - f_u + e'_{uv} \geq 0\ . \end{array} \tag{3}$$

The intuition here is that when the edge (u,v) is not cut (i.e., u and v are in the same partition), we would like e_{uv} and e'_{uv} to both be zero. When u and v are in different partitions, we would like a non-zero cost to be associated with that edge; the constraints above ensure that the cost is at least 1 unit, because $f_u - f_v$ is -1 when u is on the server and v on the embedded node. These observations allow us to formulate the bandwidth of the cut, cap that bandwidth, and define the objective function in terms of both CPU and network load.

$$net < N \quad \texttt{where} \quad net = \left(\sum_{(u,v)\in E} (e_{uv} + e'_{uv}) r_{uv} \right) \tag{4}$$

$$\texttt{objective}: \quad \min(\alpha \ \ cpu + \beta \ \ net) \tag{5}$$

Any optimal solution of (5) subject to (1), (2), (3), and (4) will have $e_{uv} + e'_{uv}$ equal to 1 if the edge is cut and to 0 otherwise. Thus, we have shown how to express our partitioning problem as an integer programming problem with a linear objective function, $2|E| + |V|$ variables (only $|V|$ of which are explicitly constrained to be integers), and at most $4|E| + |V| + 1$ equality or inequality constraints.

We could use a standard ILP solver on the formulation described above, but a further improvement is possible if we restrict the data flow to not cross back and forth between node and server, as described in Section 2.1.2. On the positive side, the restriction reduces the size of the partitioning problem, which speeds up its solution.

With the above restriction, we can then flip all edges going from server to node for the purpose of partitioning (the communication cost would be the same under our model). With all edges pointed towards the server, and only one crossing of the network allowed, another set of constraints now apply:

$$(\forall (u,v) \in E)\ f_u - f_v \geq 0 \tag{6}$$

With (6) the network load quantity simplifies:

$$net = \left(\sum_{(u,v)\in E} (f_u - f_v) r_{uv} \right)\ . \tag{7}$$

This formulation eliminates the e_{uv} and e'_{uv} variables, simplifying the optimization problem. We now have only $|V|$ variables and at most $|E| + |V| + 1$ constraints. We have chosen this restricted formulation for our current, prototype implementation, primarily because the per-platform code generators don't yet support arbitrary back-and-forth communication between node and server. We use an off-the-shelf integer programming solver, `lp_solve`[3], to minimize (7) subject to (1) and (2).

We note that the restriction of unidirectional data flow does preclude cases when sinks are pinned to embedded nodes (e.g., actuators or feedback in the signal processing). It also prevents a good partition when a high-bandwidth stream is merged with a heavily-processed stream. In the latter case, the merging must be done on the node due to the high-bandwidth stream, but the expensive processing of the other stream should be performed on the server. In our applications so far, we have found our restriction to be a good compromise between provable optimality and speed of finding a partition.

4.3 Data Rate as a Free Variable

It is possible that the partitioning algorithm will not be able to find a cut that satisfies all of the constraints (i.e., there may be no way to "fit" the program on the embedded nodes.) In this situation we wish to find the maximum data rates for input sources that *will* support a viable partitioning. The algorithm given above cannot directly treat data rate as a free variable. Even if CPU and

[3]`lp_solve` was developed by Michel Berkelaar, Kjell Eikland, and Peter Notebaert. It uses branch-and-bound to solve integer-constrained problems, like ours, and the Simplex algorithm to solve linear programming problems.

network load varied linearly with data rate, the resulting optimization problem would be non-linear. However, it turns out to be inexpensive to perform the search over data-rates as an *outer loop* that on each iteration calls the partitioning algorithm.

This is because in most applications, CPU and network load increase monotonically with input data rate. If there is a viable partition when scaling input data rates by a factor X, then any factor $Y < X$ will also have a viable partitioning. Thus Wishbone simply does a binary search over data rates to find the maximum rate at which the partitioning algorithm returns a valid partition. As long as we are not over-saturating the network such that sending fewer packets actually result in more data being successfully received, this maximum sustainable rate will be the best rate to pick to maximize outputs (throughput) of the data flow graph. We will re-examine this assumption in Section 7.

5 Wishbone Platform Backends

In this section, we describe three new WaveScript code generators we built for Wishbone, which are described here for the first time. These support ANSI C, NesC/TinyOS and JavaME.

5.1 Code Generation: ANSI C and JavaME

In contrast with the original WaveScript C++ backend (and XStream runtime engine), our current C code-generator produces simple, single threaded code in which each operator becomes a function definition. Passing data via `emit` becomes a function call, and the system does a depth-first traversal of the stream graph. The generated code requires virtually no runtime and is easily portable. This C backend is used to execute the server-side portion of a partitioned program, as well as the node-side portion on Unix-like embedded platforms that run C, such as the iPhone (jailbroken), Gumstix, or Meraki.

Generating code for JavaME also straightforward, as Java provides a high level programming environment that abstracts hardware management. The basic mapping between the languages is the same as in the C backend. Operators become functions, and an entire graph traversal is a chain of function calls. Some minor problems arise due to Java's limited set of numeric types.

5.2 Code Generation: TinyOS 2.0

Supporting TinyOS 2.0 is much more challenging. The difficulties are both due to the extreme resource constraints of TinyOS motes (typically less than 10 KB of RAM and 100 KB of ROM), and to the restricted concurrency model of TinyOS (tasks must be be relatively short-lived and non blocking; all IO must be performed with split-phase asynchronous calls). Also, program objects be serialized and split into small network packets. Wishbone's support for TinyOS demonstrates its ability to use platforms with severe resource restrictions and unusual concurrency models.

Our prototype does not currently support WaveScript's dynamic memory management in code running on motes. We may support it in the future, but it remains to be seen whether this style of programming can be made effective for extremely resource constrained devices. Instead, we enforce that all operators assigned to motes use only statically allocated storage in our applications.

The most difficult issue in mapping a high-level language onto TinyOS is handling the TinyOS concurrency model. All code executes in either *task* or *interrupt* context, with only a single, non-preemptive task running at a time. Wishbone simply maps each operator onto a task. Each data element that arrives on a source operator, for example a sensor sample or an array of samples, will result in a depth-first traversal of the operator graph (executed as a series of posted tasks). This graph traversal is not re-entrant. Instead, the runtime buffers data at the source operators until the current graph traversal finishes.

This simple design raises several issues. First, generated TinyOS tasks must be neither too short nor too long. Tasks with very short durations incur unnecessary overhead, and tasks that run too long degrade system performance by starving important system tasks (for example, sending network messages). Second, the best method for transferring data items between operators is no longer obvious. In the basic C backend, we simply issue a function call to the downstream operator, wait for it to complete, and then continue computation. We cannot use this method under TinyOS, where it would force us to perform an entire traversal of the graph in a single very long task execution. But the obvious alternative also presents problems: executing an operator in its entirety before any downstream operators would require a queue to buffer all output elements of the current operator.

The full details of TinyOS code generation are beyond the scope of this paper. In short, the WaveScript compiler can convert programs programs into a cooperative multi-tasking form (via a CPS conversion). This serves two purposes: every call to `emit` can serve as a yield point, causing the task to yield to its downstream operator in a depth-first fashion (with no queues), which in turn will re-post the upstream operator upon completing the traversal. Second, based on profiling data, additional yield points can be inserted to "split" tasks to adjust granularity for system health.

6 Applications

We evaluate Wishbone in terms of two experimental applications: acoustic speech detection and EEG-based seizure onset detection. Both of these applications exercise Wishbone's capability to automatically partition a

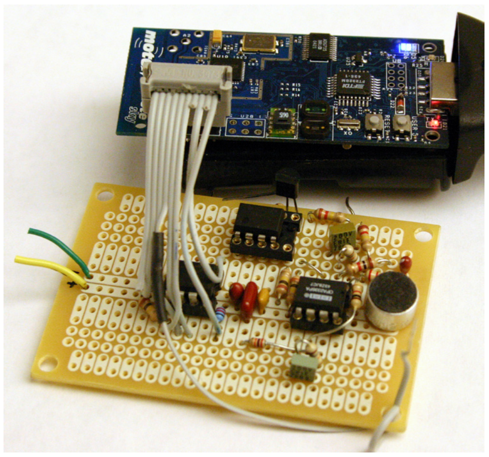


Figure 4: Custom audio board attached to a TMote Sky.

single high-level program into components that run over a network containing sensor nodes and a server or “base station”. Neither of these applications is in itself novel. In both cases we ported existing implementations from Matlab and C to Wishbone and verified that the results matched the original implementations.

6.1 Application: Seizure Onset Detection

We used Wishbone to implement a patient-specific seizure onset detection algorithm [20]. The application was previously implemented in C++, but by porting it to Wishbone/WaveScript we enabled its embedded/distributed operation, while reducing the amount of code by a factor of four without loss of performance.

The algorithm is designed to be used in a system for detecting seizures outside a clinical environment. In this application, a user would wear a monitoring cap that typically consists of 16 to 22 channels. Data from the cap is processed by a low-power portable device.

The algorithm we employ [21] samples data from 22 channels at 256 samples per second. Each sample is 16-bits wide. For each channel, we divide the stream into 2 second windows. When a seizure occurs, oscillatory waves below 20 Hz appear in the EEG signal. To extract these patterns, the algorithm looks for energy in certain frequency bands.

To extract the energy information, we first filter each channel by using a polyphase wavelet decomposition. We use a repeated filtering structure to perform the decomposition. The filtering structure first extracts the odd and even portions of the signal, passes each signal through a 4-tap FIR filter, then adds the two signals together. Depending on the values of the coefficients in the filter, we either perform a low-pass or high-pass filtering operation. This structure is cascaded through 7-levels, with the high frequency signals from the last three levels used to compute the energy in those signals. Note that at each level, the amount of data is halved.

As a final step, all features from all channels, 66 in total, are combined into a single vector which is input into a patient-specific support vector machine (SVM). The SVM detects whether or not each window contains epileptiform activity. After three consecutive positive windows have been detected, a seizure is declared.

There are multiple places where Wishbone can partition this algorithm. If the entire application fits on the embedded node, then the data stream is reduced to only a feature vector—an enormous data reduction. But data is also reduced by each stage of processing on each channel, offering many intermediate points which are profitable to consider.

6.2 Acoustic Speech Detection

We used Wishbone to build a speech detection application that uses sampled audio to detect the presence of a person who is speaking near a sensor. The ultimate goal of such an application would be to perform speaker *identification* using a distributed network of microphones. For example, such a system could potentially be used to locate missing children in a museum by their voice, or to implement various security applications.

However, in our current work we are only concerned with speech *detection*, a precursor to the problem of speaker identification. In particular, our goal is to reduce the volume of data required to achieve speaker identification, by eliminating segments of data that probably do not contain speech and by summarizing the speech data through feature extraction.

Our implementation of speech detection and data reduction is based on Mel Frequency Cepstral Coefficients (MFCC), following the approach of prior work in the area. Recent work by Martin, et al. has shown that clustering analysis of MFCCs can be used to implement robust speech detection [14]. Another article by Saastamoinen, et al. describes an implementation of speaker identification on smartphones, based on applying learning algorithms to MFCC feature sets [19]. Based on this prior work, we chose to exercise our system using an implementation of MFCC feature extraction.

6.2.1 Mel Frequency Cepstral Coefficients

Mel Frequency Cepstral Coefficients (MFCC) are the most commonly used features in speech recognition algorithms. The MFCC feature stream represents a significant data reduction relative to the raw data stream.

To compute MFCCs, we first compute the spectrum of the signal, and then summarize it using a bank of overlapping filters that approximates the resolution of human aural perception. By discarding some of the data that is less relevant to human perception, the output of the filter bank represents a 4X data reduction relative to the original raw data. We then convert this reduced-resolution spectrum from a linear to a log spectrum. Using a log spectrum makes it easier to separate convolutional components such as the excitation applied to the vocal tract and the impulse response of a reverberant environment, because transforms that are multiplicative in a linear spectrum are additive in a log spectrum.

Finally, we compute the MFCCs as the first 13 coefficients of the Discrete Cosine Transform (DCT) of this reduced log-spectrum. By analyzing the spectrum of a spectrum, the distribution of frequencies can be characterized at a variety of scales [6, 5].

6.2.2 Trade-offs in MFCC Extraction

The high level goal of Wishbone is to explore how a complex application written in a single high level language can be efficiently and easily distributed across a network of devices and support many different platforms. As such, the MFCC application presents an interesting challenge because for sensors with very limited resources there appears to be no perfect solution; rather, using Wishbone the application designer can explore different trade-offs in application performance.

These trade-offs arise because this algorithm squeezes a resource-limited device between two insoluble problems: not only is the network capacity insufficient to forward all the raw data back to a central point, but the CPU resources are also insufficient to extract the MFCCs in real time. If the application has any partitioning that fits the resource constraints, then the goal of Wishbone is to select the best partition, for example, lowest cost in terms of energy. If the application does not *fit* at its ideal data rate, ultimately, some data will be dropped on some target platforms. The objective in this case is to find a partitioning that minimizes this loss and therefore maximizes the *throughput*: the amount of input data successfully processed rather than dropped at the input sources or in the network.

6.2.3 Implementing Audio Capture

Some platforms, such as the iPhone and embedded-Linux platforms (such as the Gumstix), provide a complete and reliable hardware and software audio capture mechanism. On other platforms, including both TMotes and J2ME phones, capturing audio is more challenging.

On TMotes, we used a custom-built audio board to acquire audio. The board uses an electret microphone, four opamp stages, a programmable-gain amplifier , and a 2.5 V voltage reference. We have found that when the microphone was powered directly by the analog supply of the TMote, the audio board performed well when the mote was only acquiring audio, but was very noisy when the mote was communicating. The communication causes a slight modulation of the supply voltage, which gets amplified into significant noise. Using a separately regulated supply for the microphone removed this noise. The anti-aliasing filter is a simple RC filter; to better reject aliasing, the TMote samples at a high rate and applies a digital low-pass filter (filtering and decimating a 32 Ks/s stream down to 8 Ks/s works well). The amplified and filtered audio signal is presented to an ADC pin of the TMote's microcontroller, which has 12 bits of resolution. We use TinyOS 2.0 `ReadStream<uint16_t>` interface to the ADC, which uses double buffering to deliver arrays of samples to the application.

Phones naturally have built-in microphones and microphone amplifiers, but we have nonetheless encountered a number of problems using them as audio sensors. Many J2ME phones support the Mobile Media API (JSR-135), which may allow a program to record audio, video, and take photographs. Support for JSR-135 does not automatically imply support for audio or video recording or for taking snapshots. Even when audio recording is supported, the API permits only batch recording to an array or file (rather than a continuous stream) resulting in gaps.

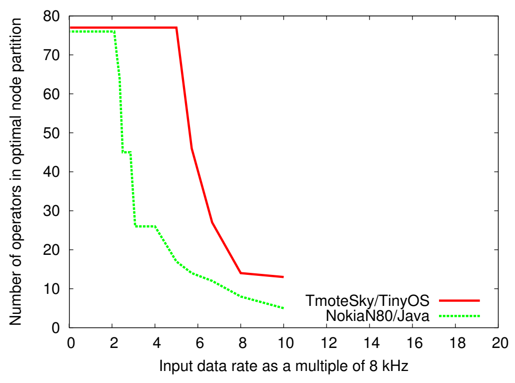
We ran into a bug on the Nokia N80: after recording audio segments for about 20 minutes, the JVM would crash. Other Nokia phones with the same operating system (Symbian S60 3rd Edition) exhibited the same bug. We worked around this bug using a simple Python script that runs on the phone and accepts requests to record audio or take a photograph through a TCP connection, returning the captured data also via TCP. The J2ME program acquires audio by sending a request to this Python script, which can record indefinitely without crashing.

The J2ME partition of the Wishbone program uses TCP to stream partially processed results to the server. When the J2ME connects, the phone asks the user to choose an IP access point; we normally use a WiFi connection, but the user can also choose a cellular IP connection. With any of these communication methods, dependence on user interaction presents a practical barrier to using phones in an autonomous sensor network. Yet these software limitations are incidental rather than fundamental, and should not pose a long-term problem.

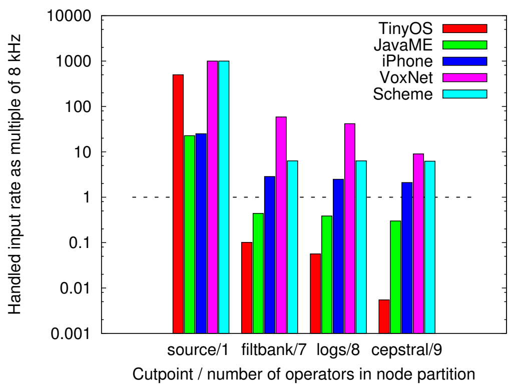
7 Evaluation

In this section we evaluate the Wishbone system on the EEG and speech detection applications we discussed in Section 6. We focus on two key questions:

1. Can Wishbone efficiently select the best partitioning for a real application, across a range of hardware devices and data rates?
2. In an overload situation, can Wishbone effectively predict the effects of load-shedding and recommend a "good" partitioning?



(a)



(b)

Figure 5: Relationship between partitioning and compute-bound sustainable data rates. On the left (a), a subset of the EEG application (one channel). The X axis shows a required data rate, the Y axis the number of operators in computed optimal node partition. On the right (b), the speaker detection application; we flip the axes due to the small number of viable cut-points. For each viable cut-point, we show the maximum data-rate supported on each hardware platform.

7.1 EEG Application

Our EEG application provides an opportunity to explore the scaling capability of our partitioning method. In particular, we look at our worst case scenario—partitioning all 22-channels (1412 operators). As the CPU budget increases, the optimal strategy for bandwidth reduction is to move more channels to the nodes. On our lower-power platforms, not all the channels can be processed on one node. The graph in Figure 5(a) shows partitioning results only for the *first* of 22 channels, where we vary the input data rate on the X axis and measure the number of operators that "fit" on different platforms. We ran `lp_solve` to derive a partitioning 2100 times, linearly varying the data rate to cover everything from "everything fits easily" to "nothing fits". To remove confounding factors, the objective function was configured to minimize network bandwidth subject to not exceeding CPU capacity ($\alpha = 0, \beta = 1$): that is, allow the CPU to be fully utilized (but not over-utilized). As we increased the data rate (moving right), fewer operators can fit within the CPU bounds on the node (moving down). The sloping lines show that every stage of processing yields data reductions.

The distribution of resulting execution times are depicted as two CDFs in Figure 6, where the x axis shows execution time in seconds, on a log scale. The top curve in Figure 6 shows that even for this large graph, `lp_solve` always found the optimal solution in under 90 seconds. The typical case was much better: 95 percent of the executions reached optimality in under 10 seconds. While this shows that an optimal solution is typically discovered in a reasonable length of time, that solution is not necessarily *known* to be optimal. If the solver is used to prove optimality, both worst and typical case runtimes become much longer, as shown by the lower CDF curve

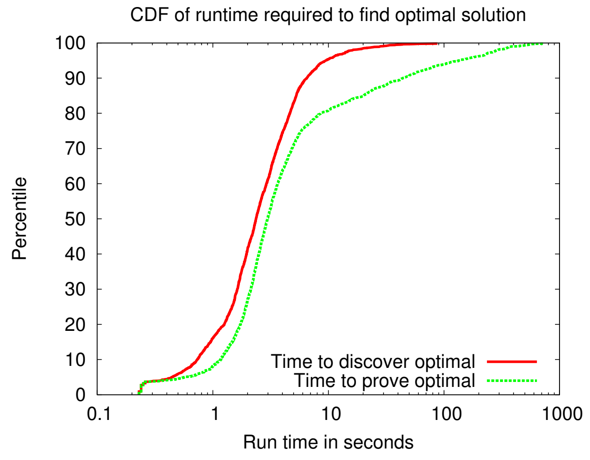


Figure 6: CDF of the time required for `lp_solve` to reach an optimal partitioning for the full EEG application (1412 operators), invoked 2100 times with data rates. The higher curve shows the execution time at which an optimal solution was found, while the lower curve shows the execution time required to prove that the solution is optimal. Execution times are from a 3.2 GHz Intel Xeon.

(yet still under 12 minutes). To address this, we can use an approximate lower bound to establish a termination condition based on estimating how close we are to the optimal solution.

7.2 Speech Detection Application

The speech detection application is a linear pipeline of only a dozen operators. Thus the optimization process for picking a cut point should be trivial—a brute force testing of all cut points will suffice. Nevertheless, this application's simplicity makes it easy to visualize and study, and the fact that the data rate it needs to process all data is unsustainable for TinyOS devices provides an op-

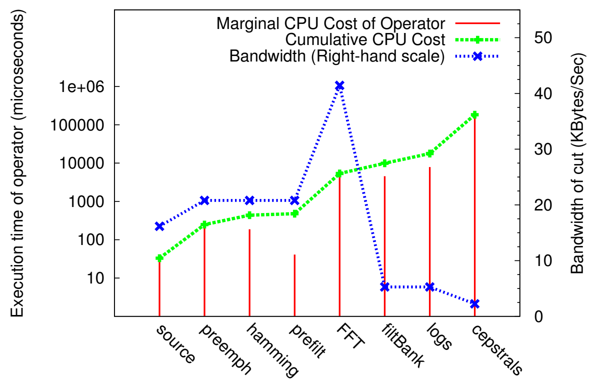


Figure 7: Data is reduced by processing, lowering bandwidth requirements, but increasing CPU requirements.

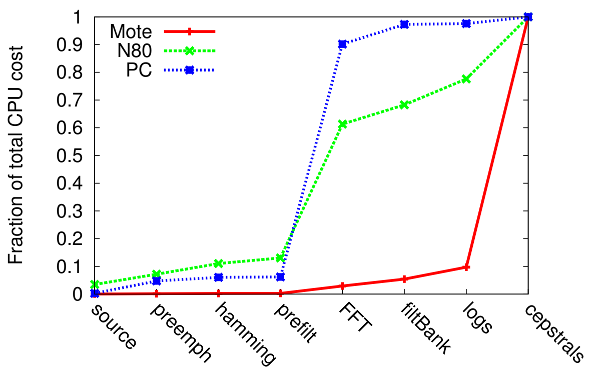


Figure 8: Normalized cumulative CPU usage for different platforms. Relative execution costs of operators vary greatly on the tested systems.

portunity to examine the other side of Wishbone's usage: what to do when the application doesn't fit.

In applying Wishbone to the development process for our speech detection application, we were able to quickly assess the performance on several different platforms. Figure 7 is a detailed visualization of the performance trade-offs, showing only the profiling results for TMote Sky (a TinyOS platform). In this figure, the X axis represents the linear pipeline of operators, and the Y axis represent profiling results. Each vertical impulse represents the number of microseconds of CPU time consumed by that operator per frame (left scale), while the line represents the number of bytes per second output by that operator. It is easy to visualize the trade-off between CPU cost and data rate. Each point on the X-axis represents a potential graph cut, where the sum of the red bars to the left provides the processing time per frame.

Thus, we see that the MFCC dataflow has multiple data-reducing steps. The algorithm must natively process 40 frames per second in real time, or one frame every 25 ms. The initial frame is 400 bytes; after applying the filter bank the frame data is reduced to 128 bytes, using 250 ms of processing time; after applying the DCT, the frame data is further reduced to 52 bytes, but using a total of 2 s of processing time. This structure means that although no split point can fit the application on the TMote at the full rate, we can achieve different CPU/bandwidth trade-offs by selecting different split points. Selecting a bad partitioning can result in retrieving no data, and the best "working" partition provides 20 times more data than the worst. Figure 5(b) shows an axes-flipped version of Figure 5(a): predicted data-rate as a function of the partition point. Only viable (data reducing) cutpoints are shown. Bars falling under the horizontal line indicate that the platform cannot be expected to keep up with the full (8 kHz) data rate.

As expected, the TMote is the worst performing platform, with the Nokia N80 performing only about twice as fast—surprisingly poor performance given that the N80 has a 32-bit processor running at 55X the clock rate of the TMote. This is due to the poor performance of the JVM implementation. The 412 MHz iPhone platform using GCC performed 3X worse than the 400 MHz Gumstix-based Linux platform; we believe that this is due to the frequency scaling of the processing kicking in to conserve power.

We can also visualize the relative performance of different operators across different platforms. For each platform processing the complete operator graph, Figure 8 shows the fraction of time consumed by each operator. If the time required for each operator scaled linearly with the overall speed of the platform, all three lines would be identical. However, the plot clearly shows that the different capabilities of the platforms result in very different relative operator costs. For example, on the TMote, floating point operations, which are used heavily in the `cepstrals` operator, are particularly slow. This shows that a model that assumes the relative costs of operators are the same on all platforms would mis-estimate costs by over an order of magnitude.

7.3 Wishbone Deployment

To validate the quality of the partitions selected by Wishbone, we deployed the speech detection application on a testbed of 20 TMote Sky nodes. We also used this deployment to validate the specific performance predictions that Wishbone makes using profiling data (e.g., if a combination of operators were predicted to use 15% CPU, did they?).

7.3.1 Network Profiling

The first step in deploying Wishbone is to profile the network topology in the deployment environment. It is important to note that simply changing the network size changes the available per-node bandwidth and thus requires re-profiling of the network and re-partitioning of the application. We run a portable WaveScript program that measures the goodput from each node in the network. This tool sends packets from all nodes at an iden-

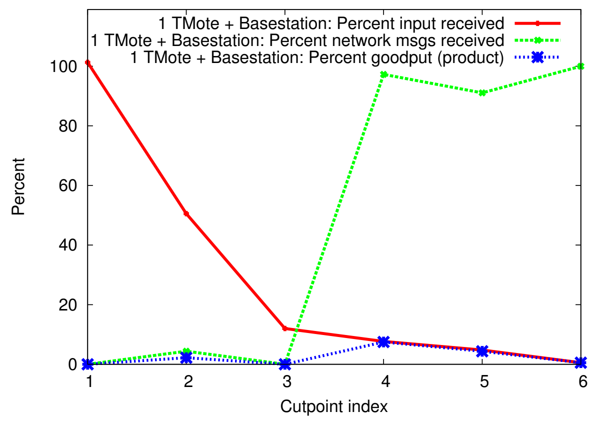


Figure 9: Loss rate measurements for a single TMote plus basestation across different partitionings. Lines show the percentage of input data processed, the percentage of network messages received, and the product of these: the goodput.

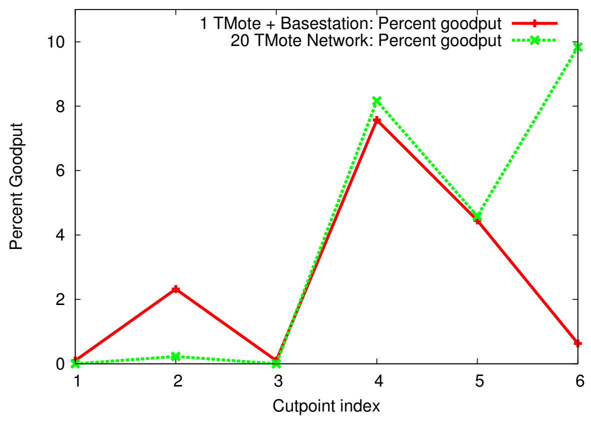


Figure 10: Goodput rates for a single TMote and for a network of 20 TMotes, over different partitionings when running on our TMote testbed.

tical rate, which gradually increases. For our 20 node testbed the resulting network profile is typical for TMote Sky devices: each node has a baseline packet drop rate that stays steady over a range of sending rates, and then at some drops off dramatically as the network becomes excessively congested. Our profiling tool takes as input a target reception rate (e.g. 90%), and returns a maximum send rate (in msgs/sec and bytes/sec) that the network can maintain.For the range of sending rates within this upper bound the assumption mentioned in 4.3 holds—attempting to send more data does not result in actual bytes of data received. Thus we are free to maximize the data rate within the upper bound provided by the network profiling tool, and thereby maximize total application throughput. This enables us to use binary search to find the the maximum sustainable data rate when we are in an overload situation.

To empirically verify that our computed partitions are optimal, we established a ground truth by exhaustively running the speech detection application at every cut point on our testbed. Figures 9 and 10 show the results for six relevant cutpoints, both for a single node network (testing an individual radio channel) and for the full 20 node TMote network. Wishbone counts missed input events and dropped network messages on a per-node basis. The relevant performance metric is the percentage of sample data that was fully processed to produce output. This is roughly the product of the fraction of data processed at sensor inputs, and the fraction of network messages that were successfully received.

Figure 9 shows the input event loss and network loss for the single TMote case, as well as the resulting goodput. On a single mote, the data rate is so high at early cutpoints that it drives the network reception rate to zero. At later cutpoints too much computation is done at the node and the CPU is busy for long periods, missing input events. In the middle, even a underpowered TMote can process 10% of sample windows. This is equivalent to polling for human speech four times a second—a reasonably useful configuration.

Figure 10 compares the goodput achieved with a single TMote and basestation to the case of a network of 20 TMotes. For the case of a single TMote, peak throughput rate occurs at the 4th cut point (filterbank), while for the whole TMote network in aggregate, peak throughput occurs at the 6th and final cut point (cepstral). As expected, the throughput line for the single mote tracks the whole line closely until cut point six. For a high-data rate application with no in-network aggregation, a many node network is limited by the same bottleneck as a network of only one node: the single link at the root of the routing tree. At the final cut point, the problem becomes compute bound and the aggregate power of the 20 TMote network makes it more potent than the single node.

We also ran the same test on an a Meraki Mini based on a low-end MIPS processor. While the Meraki has relatively little CPU power—only around 15 times that of the TMote—it has a WiFi radio interface with at least 10x higher bandwidth. Thus for the Meraki the optimal partitioning falls at cut point 1: send the raw data directly back to the server.

Having determined the optimal partitioning in our real deployment, we can now compare it to the recommendation of our partitioning algorithm. Doing this is slightly complex as the algorithm does not model message loss; instead, it keeps bandwidth usage under the user-supplied upper bound (using binary search to find the highest rate at which partitioning is possible), and minimizes the objective function. In the real network, lost packets may cause the actual delivered bandwidth to be somewhat less than expected by the profiler. Fortunately, the cut-point that maximizes throughput should be the same irrespective of loss as CPU and network load scale linearly with data rate.

In this case, binary search found that the highest data rate for which a partition was possible (respecting network and CPU limits) was at 3 input events per second (with each event corresponding to a window of 200 audio samples). The optimal partitioning at that data rate[4] was in fact cut point 4, right after filterbank, as in the empirical data. Likewise, the computed partitions for the 20 node TMote network and single node Meraki test matched their empirical peaks, which gives us some confidence in the validity of the model.

In the future, we would like to further refine the precision of our CPU and network cost predictions. To use our ILP formulation we necessarily assume that both costs are additive—two operators using 10% CPU will together use 20%, and don't account for operating system overheads or processor involvement in network communication. For example, on the Gumstix ARM-linux platform the entire speaker detection application was predicted to use 11.5% CPU based on profiling data. When measured, the application used 15% CPU. Ideally we would like to take an automated approach to determining these scaling factors.

8 Related Work

First we overview other systems that, like Wishbone, automatically partition programs—either dynamically or statically—to run on multiple devices. Generally speaking, Wishbone differs from these existing systems by using a profile-driven approach to automatically derive a partitioning, as well as its support for diverse platforms.

The Pleiades/Kairos systems [13] statically partition a centralized C-like program into a collection of node-level nesC programs that run on motes. Pleiades is primarily concerned with the correct synchronization of shared state between nodes, including consistency, serializability, and deadlocks. Wishbone, in contrast, is concerned with high-rate shared-nothing data processing applications, where all nodes run the same code. Because Wishbone programs are composed of a series of discrete dataflow operators that repeatedly process streaming data, they are amenable to our profile-based approach for cost estimation. Finally, by constraining ourselves to a single cut point, we can generate optimal partitionings quickly, whereas Pleiades uses a heuristic partitioning approach to generate a number of cut points.

Triage [3] is a related system for "microservers" that act as gateways in sensor network applications. Triage's focus is on power conservation on such servers by using a lower-power device to wake a higher-power device based on a profile of expected power consumption and utility of data coming in over the sensor network. However, it does not attempt to automatically partition programs across the two device classes as Wishbone does.

[4] In this case with $\alpha = 0$, $\beta = 1$, although the linear combination in the objective function is not particularly when we are maximizing data rate we are saturating either CPU or bandwidth

In stream processing there has been substantial work looking at the problem of migrating operators at runtime [2, 18]. Dynamic partitioning is valuable in environments with variable network bandwidth, unpredictable load, but also comes with serious downsides in terms of runtime overheads. Also, by focusing on static partitioning, Wishbone is able to provide feedback to users at compile time about whether their program will "fit" their sensor platform and hardware configuration.

There has been related work in the context of traditional, non-sensor related distributed systems. For example, the Coign [11] system automatically partitions binary applications written using the Microsoft COM framework across several machines, with the goal of minimizing communication bandwidth. Like Wishbone, it uses a profile-driven approach. Unlike Wishbone, Coign does not formulate partitioning as an optimization problem, and only targets Windows PCs. Neubauer and Thiemann [15] present a similar framework for partitioning client-server programs. Automatic partitioning is also widely-used in high-performance computing, where it is usually applied to some underlying mesh, and in automatic layout of circuits. Finally, several systems, including JESSICA2 [25], MagnetOS [4], and cJVM [1], implement distributed Java virtual machines that appear as a single system. These systems must use runtime methods to load-balance threads between machines. The overheads on communication and synchronization are typically high, and only applications with a high ratio of computation to communication will scale effectively.

Tenet [9] proposes a two-tiered architecture with programs decomposed across sensors and a centralized server, much as in Wishbone. The VanGo system [10], which is related to Tenet, proposes a framework for building high data rate signal processing applications in sensor networks, similar to the applications that inspired our work on Wishbone. But VanGo is constrained to a linear chain of filters, does not support automatic partitioning, and runs only TinyOS code.

Marionette [24] and SpatialViews [17] use static partitioning of programs between sensor nodes and a server that is explicitly under the control of the programmer. These systems work by allowing users to invoke predefined handlers (written in, for example, nesC) from a high-level centralized program that runs on a server, but neither offers automatic partitioning.

Abstract Regions [22] and Hood [23] enable operations over clusters of nodes (or "regions") rather than single sensors. They allow data from multiple nodes to be combined and processed, but are targeted at coordinating sensors rather than stream processing.

9 Future Work/Conclusions

The model presented in this paper enables communication between embedded endpoints and a central server. But it would be straightforward to extend our model with a basic form of in-network aggregation: namely, tree-based aggregation that happens at every node in the network, useful, for example, for taking average sensor readings. This communication pattern would be exposed as a "reduce" operator that would reside in the logical node partition, but would implicitly take its input not just from streams within the local node, but from child nodes routing through it in an aggregation tree. The partitioning algorithm remains the same. If the reduce operator is assigned to the embedded node, aggregation happens in-network, otherwise all data is sent to the server.

Also, while our prototype implementation only supports networks of one type of node, the model can also handle certain kinds of mixed networks. A single logical node partition can take on different physical partitions at different nodes. This is accomplished simply by running the partitioning algorithm once for each type of node. The server would need to be engineered to deal with receiving results from the network at various stages of partial processing. In the future, mixed partitions may be desirable even for homogeneous networks. Varying wireless link quality can create a situation where each node should partitioned differently.

A more radical change would extend the model with multiple logical partitions corresponding to categories of devices. This opens up several design choices; for example, what communication relationship should the logical partitions should have? We have verified that we can use an ILP approach for a restricted *three tier* network architecture. (Motes communicate only to microservers, and microservers to the central server.) But going further would require revisiting the partitioning algorithm.

Acknowledgements: This work was supported by the NSF under grants 0520032, 0720079, and 0754662.

References

[1] Y. Aridor, M. Factor, and A. Teperman. cjvm: A single system image of a jvm on a cluster. In *Proc. ICPP*, 1999.

[2] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-Based Load Management in Federated Distributed Systems. In *Proc. USENIX NSDI*, San Francisco, CA, Mar. 2004.

[3] N. Banerjee, J. Sorber, M. D. Corner, S. Rollins, and D. Ganesan. Triage: balancing energy and quality of service in a microserver. In *MobiSys*, 2007.

[4] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer. On the need for system-level support for ad hoc and sensor networks. *SIGOPS Operating Systems Review*, 36(2), 2002.

[5] R. A. Cole, J. Mariani, H. Uszkoreit, A. Zaenen, and V. Zue. Survey of the state of the art in human language technology, 1995.

[6] S. Davis and P. Mermelstein. Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences. *IEEE Trans. on ASSP*, 28:357–366, 1980.

[7] E. B. et al. *Zoltan 3.0: Data Management Services for Parallel Applications; User's Guide*. Sandia National Laboratories, 2007. Tech. Report SAND2007-4749W.

[8] M. Garey, D. Johnson, , and L. Stockmeyer. Some simplified NP complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[9] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler. The tenet architecture for tiered sensor networks. In *SenSys*, 2006.

[10] B. Greenstein, C. Mar, A. Pesterev, S. Farshchi, E. Kohler, J. Judy, and D. Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *SenSys*, pages 279–292, 2006.

[11] G. C. Hunt and M. L. Scott. The Coign automatic distributed partitioning system. In *Proc. OSDI*, 1999.

[12] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20:359–392, 1998.

[13] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *Proc. PLDI*, 2007.

[14] A. Martin, D. Charlet, and L. Mauuary. Robust speech/non-speech detection using LDA applied to MFCC. In *IEEE Intl. Conference on Acoustics, Speech, and Signal Processing*, pages 237–240, 2001.

[15] M. Neubauer and P. Thiemann. From sequential programs to multi-tier applications by program transformation. In *Proc. PLDI*, 2005.

[16] R. R. Newton, L. D. Girod, M. B. Craig, S. R. Madden, and J. G. Morrisett. Design and evaluation of a compiler for embedded stream programs. In *Proc. LCTES*, 2008.

[17] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proc. PLDI*, 2005.

[18] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[19] J. Saastamoinen, E. Karpov, V. Hautamki, and P. Frnti. Accuracy of MFCC-Based speaker recognition in series 60 device. *EURASIP Journal on Applied Signal Processing*, (17):2816–2827, 2005.

[20] A. Shoeb et al. Patient-Specific Seizure Onset. *Epilepsy and Behavior*, 5(4):483–498, 2004.

[21] A. Shoeb et al. Detecting Seizure Onset in the Ambulatory Setting: Demonstrating Feasibility. In *IEEE EMBS 2005*, September 2005.

[22] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[23] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proc. Mobisys*, 2004.

[24] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proc. IPSN*, 2006.

[25] W. Zhu, C.-L. Wang, and F. C. M. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Proc. CLUSTER*, page 381, 2002.

Softspeak: Making VoIP Play Well in Existing 802.11 Deployments

Patrick Verkaik, Yuvraj Agarwal, Rajesh Gupta, and Alex C. Snoeren

University of California, San Diego
{pverkaik,yuvraj,gupta,snoeren}@cs.ucsd.edu

Abstract

Voice over IP (VoIP) in 802.11 wireless networks (WiFi) is an attractive alternative to cellular wireless telephony. Unfortunately, VoIP traffic is well known to make inefficient use of such networks. Indeed, we demonstrate that increasing handset deployment has the potential to cripple existing hotspot and enterprise WiFi networks. Our experiments show that VoIP halves the available TCP capacity of an 802.11b hotspot when six to eight VoIP stations share the medium, and effectively extinguishes TCP connectivity when ten VoIP stations are present. Further, we show that neither the higher data rates of 802.11a/g nor the 802.11 standard for quality of service, 802.11e, fully ameliorate the problem. Instead, the problem is rooted in WiFi's contention-based medium-access control mechanism and considerable framing overhead.

To remedy this problem, we propose Softspeak, a pair of backwards-compatible software extensions that enables VoIP traffic to share the channel in a more efficient, TDMA-like manner. Softspeak does not require any modifications to the WiFi protocols and significantly reduces the impact of VoIP on TCP capacity while simultaneously improving key VoIP call-quality metrics. Results show improvements in TCP download capacity of 380% for 802.11b and 25-200% for 802.11g.

1 Introduction

Voice-over-IP (VoIP) technology is now pervasive in wire-line networks, embodied by wildly successful applications like Skype. Wireless deployment, in contrast, has so far been limited to certain niche products. Recently, however, WiFi-capable consumer phone handsets such as T-Mobile's UMA and the Apple iPhone have been released to the US market in large numbers, portending a huge influx of WiFi VoIP users once third-party applications like iCall [1] become widely available for these platforms. In the near future, it may not be unusual for a dozen active WiFi VoIP handsets to be in range of a single WiFi hot-spot, for example at a local Starbucks.

One might imagine that such a scenario would be easily supported by existing installations, as VoIP is a relatively low-bandwidth protocol. For example, given an 802.11b channel with 11 Mbps of capacity, a G.729[1] VoIP codec rate of 6.4 Kbps, and a combined header size of RTP, UDP and IP of 40 bytes, one might expect a single AP to support over 70 bidirectional VoIP calls and still leave half of the channel capacity for data traffic. It is well known, however, that nothing could be further from the truth; previous researchers have shown that an 802.11b network supports as few as six simultaneous VoIP sessions [4, 9, 20], depending upon the particular characteristics of the network and codecs in use. This counterintuitive result is due to the large per-packet overhead imposed by WiFi for each VoIP packet—both in terms of protocol headers and due to WiFi contention.

Call quality has traditionally been a major concern for WiFi VoIP deployments, since real-time audio traffic has stringent requirements in terms of loss rate, delay and jitter, and needs to be sent at a high rate (e.g., 50–100 packets per second for many VoIP codecs) to maintain acceptable audio quality. In mixed-use cases, best-effort traffic can cause excessive queuing of VoIP traffic at access points and may increase packet loss rate due to contention for the medium. Since a VoIP call occupies only a very small amount of bandwidth (possibly as few as eight bytes of voice data per packet), many researchers [4, 25] and commercial providers [2] have proposed prioritizing VoIP packets, with the unstated assumption that the impact on overall network performance will be minimal. However, as we demonstrate experimentally, as few as six VoIP calls may remove over half of the TCP capacity in 802.11b. Moreover, prioritizing VoIP sessions runs the very real danger of drowning out all competing best-effort traffic, such as Web browsing and email messaging. Somewhat surprisingly, our experiments show that neither the increased speed of 802.11a/g nor the quality-of-service mechanisms of 802.11e change this reality.

In this paper, we address the impending potential disaster: that widespread VoIP usage will cripple hotspot and enterprise WiFi networks. In addition to quantifying and explaining the impact of VoIP on the capacity of WiFi, we propose backward-compatible modifications to 802.11 that aggregate multiple VoIP clients into the equivalent of a single VoIP client, thus reducing VoIP's impact on the network's data-carrying capacity.

Previous work in this domain has proposed the concept of 'downlink aggregation' in simulation [23, 24], which encapsulates multiple VoIP packets into a single packet at the AP, addressed to all VoIP stations associated with the same AP. Our experiments demonstrate, however, that downlink aggregation is insufficient to fully address the problem. We present a complementary technique for the uplink direction that serializes channel access by establishing a TDMA-like schedule. We show that this can be done in a distributed manner by independent VoIP stations. We combine uplink TDMA and downlink aggregation mechanisms to develop a system called Softspeak that simultaneously improves VoIP call quality while preserving network capacity for best-effort data transfer.

We implement and evaluate Softspeak on a testbed of Linux-based 802.11b/g/e devices within an operational enterprise WiFi network. We show that Softspeak improves residual downlink TCP capacity of the network substantially, e.g., by 380% in the presence of ten VoIP calls in 802.11b and by 200% in 802.11g (protected mode). We also achieve significant improvements in UDP and TCP uplink capacity, as well as in 802.11g unprotected mode. Furthermore, we show that Softspeak can improve VoIP call quality, providing an important incentive for client deployment. To the best of our knowledge, our work is the first to present a system based on commodity hardware that performs both uplink TDMA and downlink aggregation to improve the performance of multiple, simultaneous VoIP sessions while increasing the residual data-carrying capacity of the WiFi network.

2 The impact of VoIP on WiFi

In this section we empirically demonstrate the degradation of WiFi network capacity as well as VoIP call quality in the presence of an increasing number of VoIP clients. We then employ a detailed simulation of the 802.11 DCF algorithm to determine the precise source of the problem.

2.1 Sources of overhead

The 802.11 protocol is designed to allow clients to access the channel in a distributed manner. Uncoordinated approaches are known to be inefficient under heavy load as collisions become more frequent and the total airtime utilization of the wireless channel reduces dramatically due to airtime wasted on garbled frames. This problem is particularly relevant in the case of VoIP traffic, since VoIP clients contend often due to the real-time nature of the traffic. The resulting increased collision rate increases loss and jitter, which in turn degrade TCP performance and harm VoIP call quality.

Furthermore, given the small data payload of VoIP packets the overhead of transmitting the various headers in a VoIP packet becomes considerable: each VoIP packet in a WiFi network is typically encumbered with RTP, UDP, IP, MAC and PHY headers as well as a synchronous 802.11 ACK frame. For example, a G.729 packet may take 157 μs to transmit at the maximum rate in 802.11b, or 273 μs if we include the ACK frame (and assume it is sent at maximum rate). Of this time, the eight bytes of voice data carried inside the packet take up only six microseconds; the entire IP packet requires only 35 μs of airtime, resulting in 680% overhead. Although 802.11g can reduce this overhead to 240% in the best case, the overhead remains substantial at over 400% (again optimistically assuming maximum rates are used) in protected mode, which is required when any legacy 802.11b device is present.

Additionally, airtime usage may increase in response to loss rate, as rate control algorithms frequently lower the transmission rate in response to loss, regardless of whether the loss was due to poor signal quality or frame collision. Finally, we note that the resulting increase in airtime scarcity in turn tends to increase collision probability and loss rate as more stations attempt to seize the channel at once, thereby completing a vicious circle.

2.2 Experimental observation

To quantify the impact of VoIP traffic on background data transmissions, we have configured a testbed to reflect a realistic scenario for VoIP usage in the enterprise: stations sending and receiving VoIP traffic are spread out over several offices and are connected to an operational building-wide wireless network. For controlled experimentation we ensure that all stations associate to the same AP and do not roam between different APs. We use wireless cards from two different manufacturers to ensure our results are not artifacts of a particular piece of hardware and consider 802.11b, g and e. (Full details of the testbed are included in Section 4.1.) Unless specified otherwise, all experiments employ a 10-ms G.729 codec.

2.2.1 Residual capacity

We are interested in the residual WiFi capacity as well as VoIP call quality in the presence of a varying number of VoIP stations. Here, we measure the residual capacity by simultaneously running a bulk flow and measuring its throughput. We conduct separate experiments for uplink and downlink bulk flows, using both TCP and UDP. Our experiments with UDP measure the raw channel capacity available, while TCP measures the effective capacity for flows that are sensitive to loss and delay. For simplicity, we restrict our discussion to experiments using a single non-VoIP flow at a separate client; we present results for multiple data clients in Section 4.5.

Figure 1 plots the throughput of TCP in the presence of a varying number of VoIP stations in an 802.11b network. As we increase the number of VoIP streams, the

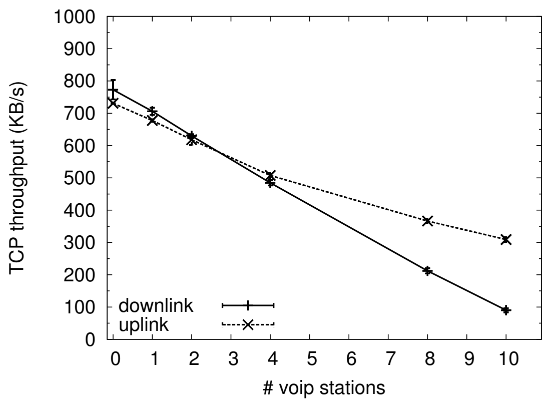


Figure 1: TCP throughput as a function of the number of VoIP streams in 802.11b (Avaya AP-8 access point).

throughput of a TCP uplink flow (where "uplink" refers to the direction of the TCP data packets) degrades, halving at around eight VoIP streams. In typical TCP usage (e.g., Web traffic) more throughput is required from the downlink direction than from the uplink direction. Unfortunately, throughput degradation is far worse for a TCP downlink flow, which can be explained as follows. TCP's congestion control mechanism attempts to use the maximum bandwidth available given the loss rate and the RTT. For both cases, the TCP sender needs to share the AP with other traffic for its downlink traffic (data packets for TCP downlink or ACK packets for TCP uplink), and it is therefore at the AP that most losses are expected to occur. Losing a data packet is far worse than losing an ACK packet, however. Therefore, TCP is able to tolerate a higher loss rate at the AP and achieve a higher throughput when sending data uplink. As a result, TCP downlink throughput halves at six VoIP streams and degrades by over 85% in the presence of ten VoIP streams.

UDP throughput degradation is less severe than that of TCP because UDP is less sensitive to loss and delay. Nevertheless we observe a significant throughput degradation (over 55% with ten VoIP sessions). We further note that the behavior of uplink UDP and TCP traffic and their impact on VoIP traffic appears quite similar, indicating that in our testbed the TCP uplink behavior is characterized mostly by channel capacity, rather than by loss and delay.

2.2.2 Call quality

As we increase the number of simultaneous VoIP sessions, the individual call quality also decreases. Call quality is a function of packet loss rate, delay and delay jitter, and is typically represented as a Mean Opinion Score (MOS) ranging from 1 (bad) to 5 (good). We use an approximation of MOS based on network-level metrics [6] with codec-specific parameters calibrated using simulation [7]. We assume a playout buffer that is able to adapt its de-jitter delay such that on average no more than 1% of packets are late. We find that in the presence of TCP and bulk UDP uplink traffic, MOS decreases from 3.8 to 1 as the number of VoIP stations increases from one to ten. In these cases VoIP traffic undergoes severe loss (reaching 50%) due to drop-tail queuing at the AP queue where it competes with bulk data or TCP acknowledgments. Conversely, TCP downlink traffic is suppressed by VoIP traffic to such an extent that the VoIP MOS remains relatively unaffected. A major challenge is thus to improve TCP downlink performance without sacrificing call VoIP quality.

2.2.3 802.11 protocol extensions

To evaluate whether higher bit rates alleviate problems of contention and overhead we perform the same set of experiments using 802.11g. We find that throughput degradation is less severe in pure 802.11g networks than in 802.11b. For example, TCP downlink performance does not drop as sharply as it does in 802.11b, but degrades in a similar way to TCP uplink and UDP performance. The loss in capacity when ten VoIP clients are present is still substantial, however, ranging from a 32% reduction in the case of UDP downlink to 39% for TCP downlink traffic. Similarly, while VoIP MOS is higher in 802.11g, it is still unacceptably low, dropping from 3.8 to 1.3 as the number of VoIP sessions increases from one to ten due to frequent losses.

In practice, however, our enterprise WiFi deployment almost never supports only 802.11g clients. For backwards compatibility, 802.11g requires a "protected mode" be used when 802.11b stations are detected. In protected mode an 802.11g station precedes each transmission by a clear-to-send (CTS) frame, thus increasing per-frame overhead. We observe that the capacity degradation caused by 802.11g VoIP clients in an 802.11g protected-mode network is comparable to that of native 802.11b. Thus, the presence of a single legacy 802.11b client (VoIP or otherwise) alongside ten VoIP clients removes 87% of TCP downlink capacity. In addition, we find that whereas VoIP uplink loss is negligible in 802.11b in the presence of TCP downlink traffic, it varies from 10–40% in 802.11g protected mode, resulting in an average VoIP MOS value of 2.0.

The 802.11e protocol is specifically designed to allow real-time and data traffic to co-exist efficiently by prioritizing real-time traffic. We compare the performance of 802.11b and 802.11b+e using a popular 802.11e capable access point (a Linksys WAP4400N, different from the Avaya AP-8 used in the previous experiments, which does not support 802.11e), with VoIP traffic configured to be classified and prioritized over other traffic at both the AP and the clients. In the presence of TCP uplink traffic, we observe that compared to 802.11b, 802.11e

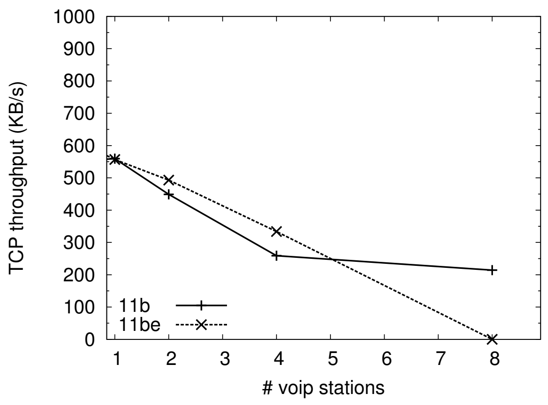


Figure 2: TCP uplink throughput as a function of the number of VoIP stations in both 802.11b and 802.11b+e (Linksys WAP4400N access point).

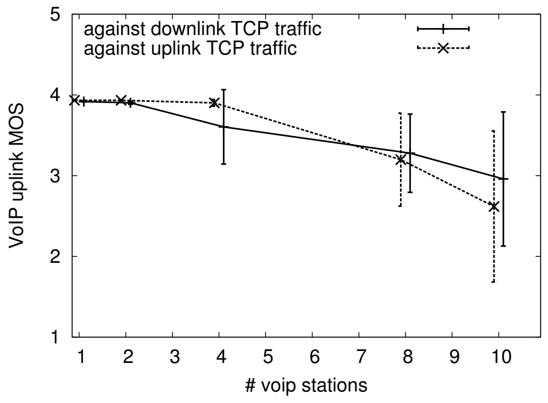


Figure 3: Uplink MOS of a 20-ms codec in 802.11g (protected mode) in the presence of TCP traffic. (Data points are slightly offset to avoid overlapping error bars.)

does indeed improve the MOS of VoIP traffic. However, as shown in Figure 2, this improvement is achieved at the expense of TCP uplink throughput, which degrades far more severely than is the case for 802.11b. TCP downlink performance is essentially similar to that of 802.11b, with a slight improvement in MOS. We conclude that while 802.11e (at least as implemented by a popular AP vendor) is able to improve call quality in some cases, it does not mitigate throughput degradation in the presence of a large number of VoIP clients.

2.2.4 Less aggressive codecs

By combining multiple 10-ms voice frames into a single IP packet, G.729 can be run at longer inter-packet intervals, thereby making more efficient use of network resources. Figure 3 considers a 20-ms G.729 codec in combination with TCP in 802.11g protected mode. As expected, the impact is less than for a 10-ms codec yet remains severe; the MOS for uplink VoIP traffic drops from 4 to 3 on average (compared to 2 in the 10-ms case) and, more importantly, becomes highly erratic. Uplink and downlink TCP throughput reduce by around 40% (not shown, c.f. 87% in the 10-ms case for TCP downlink).

2.3 802.11b simulator

While our experiments clearly demonstrate real-world performance problems, it is often difficult to determine to what extent the degradation measured is due to the 802.11 protocol rather than interference, fading, hidden terminals, or other environmental factors. In order to cleanly separate these factors, we have implemented an 802.11 protocol simulator that allows us to evaluate how aspects of the standard distributed coordination function (DCF) algorithm impact performance, in particular residual capacity. We specifically omit the simulation of RF properties, rate adaptation, background broadcast traffic (e.g., DHCP and ARP), and hardware imperfections, in order to show that the DCF algorithm by itself explains our experimental observations of residual capacity. We focus on the percentage of time a client uses the medium, since it not only directly reflects bulk UDP throughput, but also indirectly reflects loss rate: in a DCF-based model losses are caused by colliding packets, which in turn occupy airtime.

2.3.1 Configuration and validation

The simulator contains objects representing the AP and wired and wireless stations that send UDP traffic (bulk traffic or based on the traffic characteristics of a VoIP codec). Wired stations are modeled as directly connected to the AP. The wireless stations and AP contend for access using the standard 802.11 DCF algorithm. We parameterize the simulator to mimic the behavior of our testbed hardware (particular settings are detailed later in Table 1) and use a bit rate of 11 Mbps. We configure an AP queue length of 500 and station queue lengths of 10, but note that our simulation results are not sensitive to the choice of queue-length parameters.

We simulate the 802.11b experiment described earlier for UDP and find that the results are very similar in airtime. For example, simulated throughput degradation is within 10% of the experimental results. The largest difference between the simulated and experimental results is seen in the uplink VoIP loss rate which is 0.8–2.3% for ten VoIP stations versus less than 0.02% on the testbed.

2.3.2 DCF's share of VoIP impact

Having established that our simulation exhibits a similar behavior as the testbed in 802.11b, and that a DCF-based model is sufficient to explain the degradation of residual capacity in our testbed under VoIP, we now analyze the simulation data to determine which aspect of DCF causes the observed behavior. Figure 4 shows the simulated airtime used by each of the following components: non-colliding bulk traffic (*bulk*), non-colliding VoIP uplink

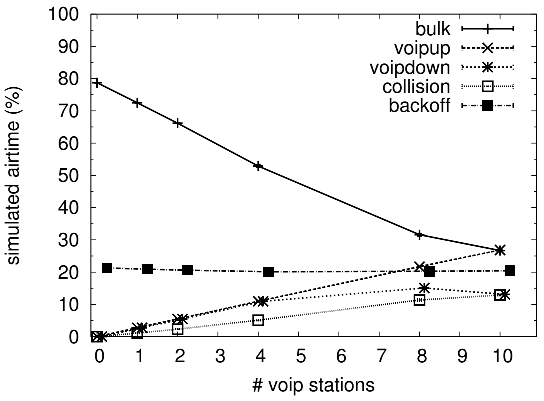


Figure 4: Simulated airtime versus the number of VoIP streams, in the presence of 802.11b UDP uplink traffic.

and downlink traffic (*voipup*, *voipdown*), colliding packets (*collisions*), and times when all stations are backing off or sensing the medium (*backoff*).

VoIP takes up a large fraction of the airtime, e.g., 40% for ten sessions, exceeding the airtime used by bulk traffic. Most of the VoIP airtime (35%) consists of framing overhead. Additionally, 33% of total airtime is overhead due to contention (20% backoff plus 13% wasted on collisions). The techniques presented in the next section are capable of reducing a significant portion of overhead, specifically the framing overhead of downlink VoIP traffic (11%) and the collision time (13%). Based upon these numbers alone there is potential to almost double the residual channel capacity.

3 Softspeak

Softspeak targets the key challenges of excessive contention and framing to build a software-only solution that can be deployed on existing commodity hardware. The main idea is to aggregate voice traffic by combining many small packets into larger ones, thereby reducing per packet overhead. Others have observed that all *downlink* packets must pass through the AP; hence, the opportunity to aggregate exists at either at the AP itself or just before the packets are sent to the AP [23, 24]. However, physically aggregating *uplink* VoIP packets is challenging since there are multiple, independent VoIP senders. Instead, we propose a time-division multiple access (TDMA) scheme that approximates uplink aggregation to the extent that it provides a similar reduction in contention overhead. Our uplink TDMA scheme can function independently of the downlink scheme and requires only client-side modifications. Downlink aggregation, on the other hand, also requires either modifying the AP, or, more realistically, adding a separate "VoIP aggregator" device upstream from the AP. Both mechanisms conform to the existing 802.11 specification and coexist with VoIP stations that do not use Softspeak.

3.1 Uplink TDMA

Our uplink approach reduces the amount of contention created by VoIP clients. Specifically, we alter the contention behavior of the VoIP clients to no longer contend with non-VoIP clients, and then devise a distributed mechanism to schedule the VoIP clients in a TDMA fashion so that they no longer contend with each other either.

We remove the VoIP clients from the standard contention process by modifying their backoff behavior. Instead of sensing the medium for the 802.11-mandated DCF inter-frame spacing (DIFS) followed by a random backoff before sending, a Softspeak VoIP client senses for a shorter period of time and does not perform backoff, thus preventing collisions with non-VoIP traffic. (In the absence of hidden terminals, collisions with ACKs are prevented by 802.11's NAV mechanism.) This behavior effectively prioritizes uplink VoIP traffic and improves call quality. (A similar mechanism is employed by a commercial product, SVP [2].) By itself, however, this alteration inhibits DCF's ability to prevent collisions among the VoIP stations. In fact, when we simulate only two VoIP stations that sense for a short inter-frame spacing (SIFS) without backoff in combination with bulk traffic that uses standard contention, we find that neither VoIP station is able to sustain a viable VoIP session.

To prevent VoIP stations from colliding with each other, we introduce coarse-grained time slots and construct a TDMA schedule for the VoIP clients. When used in combination with downlink aggregation, the downlink aggregator node can assign TDMA slots as well as perform admission control, since it has knowledge of all the clients using our scheme. In the absence of a centralized scheduler, we devise a distributed mechanism (Section 3.1.1) that leverages management frames within the 802.11 protocol to allocate slots.

3.1.1 Slot allocation and admission control

In an ideal deployment, the network operator will have installed a Softspeak VoIP downlink aggregator that can assign slots for uplink TDMA. If all available slots are in use it can deny access to a new Softspeak client, in which case that client resorts to normal 802.11 DCF. In some scenarios, however, it may be easier for individual clients to install Softspeak software than to convince network operators to install new hardware. Moreover, uplink TDMA is useful by itself, i.e., without downlink aggregation, since it reduces contention by uplink VoIP stations. Hence, if clients are unable to locate a VoIP aggregator (Section 3.2 describes the registration process), they proceed with a distributed allocation process.

Independent of how TDMA slots are allocated to clients, VoIP stations need to be synchronized in order to correctly use their assigned slots. Each client uses the periodic beacon frame broadcast by an 802.11 AP to synchronize with other VoIP clients. Beacons are sent at fixed intervals (usually 100 ms), and, since they are sent by the AP at a low bit rate, are typically received by all clients. It is important to note that a VoIP client may also hear beacons from an AP other than the one to which it is associated. To use beacon-based synchronization, VoIP clients need two important pieces of information: a) The AP to whose beacons other nearby VoIP clients are synchronizing, and b) which TDMA slots they are using. The slot allocation process provides both pieces of information. In the case of distributed slot allocation each VoIP client encodes the information by temporarily spoofing its MAC address (6 octets) as follows:

- The first three octets (known as the OUI) are taken from a reserved OUI address space to ensure the resulting address is valid and unique.
- The next two octets are the same as the *last two* octets of the BSSID of the AP to whose beacons the VoIP station is synchronizing.
- The last octet is used to denote the particular real time slot the VoIP station is using or wants to use.

The main concern when coordinating clients is that there is no guarantee they can hear each other's transmissions. Hence, Softspeak clients coerce the AP into generating specially crafted packets that the other clients can hear. VoIP stations using uplink TDMA periodically (e.g. once a second) send directed Probe-Requests on the channel and to the AP to which they are currently associated using the modified MAC address. The destination (unmodified) AP will respond with a Probe-Response packet whose destination is the VoIP station's modified MAC address, which is heard by all associated clients.

A new VoIP station that wants to use uplink TDMA first enters promiscuous mode for a few seconds to sense the channel to check if there are any special Probe-Response packets (easily identifiable by the first three octets of the destination MAC address), thus determining which AP's beacons are being used for synchronization and which slots are in use. If the VoIP client detects any such Probe-Responses, it extracts the encoded AP and uses that for TDMA synchronization. Otherwise it synchronizes using the AP with which it is associated. In either case, the VoIP client picks an unused slot and starts to periodically broadcast a Probe-Request with its source MAC address denoting its slot and the AP it is using for synchronization. As before, the AP sends Probe-Responses which can be heard by new VoIP clients wanting to join. Finally, when a VoIP station finishes its session it stops sending Probe-Requests.

Our slot assignment scheme seamlessly supports dynamic node arrivals and departures. Moreover, this scheme works even when nearby clients are associated to different APs, since a client may synchronize with an AP other than the one it is associated to. Finally, our scheme works if APs use various 802.11 security features since Probe-Request and Probe-Responses are always sent unencrypted. We have deployed our scheme with an AP that employs MAC-address-based access control, WPA2 or WEP encryption, and disabled SSID broadcasting.

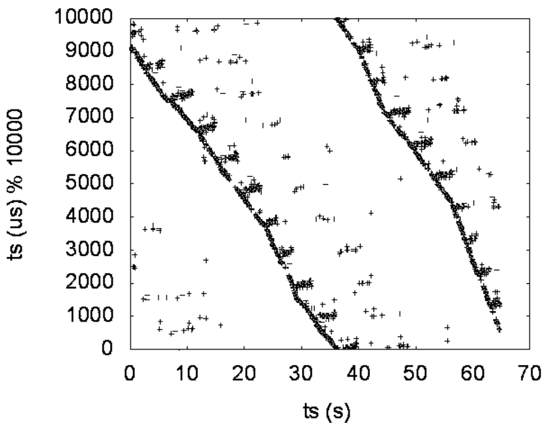


Figure 5: Time series of transmission times by a single station, no synchronization.

A drawback of the distributed allocation scheme as currently described is that it is unable to detect multiple clients attempting to allocate the same slot simultaneously. We observe that this problem can be solved (or made unlikely to occur) by adding some bits of randomness to the spoofed MAC address, allowing the clients to arbitrate among conflicting slot allocations. For example, the scheme may be extended by having VoIP clients announce the BSSID and the slot number in separate Probes, thus allowing room for some bytes to be set randomly by each client.

3.1.2 Synchronizing TDMA slots

To implement uplink TDMA, we modify the Ralink RT2560F wireless card protocol stack in Linux 2.6.21 (without modifying the WiFi hardware or firmware). Ideally, once slots are allocated, each VoIP station contends for the channel in its assigned slot and refrains from contending outside its slot. By default, the Linux 2.6 kernel timer interrupt is programmed to fire every millisecond; we show later that this also happens to be close to the optimal granularity for VoIP slotting in 802.11b. Using one-millisecond slots, a TDMA scheme can support ten simultaneous VoIP stations using a codec with 10-ms inter-packet arrival rate, or 20 stations using a 20-ms codec. Since 802.11a/g frames for these codecs take less airtime, Softspeak could use smaller slots, allowing a larger number of VoIP stations to be admitted; we have not yet implemented sub-millisecond slotting.

A straightforward implementation of one-millisecond slotting is to suspend and resume transmission from within Linux's timer interrupt handler in accordance with a station's assigned slot. However, the naïve approach faces two problems: clock skew and timer inaccuracy. Figure 5 illustrates both. In this experiment, a single station uses `iperf` to emulate a G.729 VoIP codec with a 10-ms inter-packet arrival rate. We manually assign the station a static TDMA slot; there is little to no background traffic on the same AP during the experiment

In the figure, the x axis plots time in seconds, and the y axis shows the start time of each transmission modulo 10,000 μs (10 ms). The figure shows the effect of the timer interrupt firing faster than 1,000 times per second as well as `iperf` sending slightly slower than the configured rate of 100 packets per second. If the timer interrupt and `iperf` operated at their correct rate, we would expect to see a single horizontal band corresponding to the station's assigned slot. Instead, `iperf` schedules packets at a rate slower than the timer interrupt, and as a result iperf and the implemented TDMA slot drift with respect to each other. When `iperf` happens to send inside the slot, a short almost horizontal line appears starting at the bottom of the slot (the slight upward slope of this line is the clock skew). Once transmissions reach the top of the slot, packets are buffered until the start of the next slot, causing the downward sloping lines. The slope is caused by the timer interrupt firing too fast.

Different stations may exhibit different degrees of skew, possibly even varying across time. We address this issue by effectively slaving each station's clock to an AP. Specifically, we reset the timer every time a station hears the periodic beacon frame from the AP that was assigned during the slot allocation process. On the Soekris net4801 in our testbed, Linux uses the programmable interval timer (PIT) as its time interrupt source. Therefore, we modify the driver to reset the PIT every time it hears a beacon, which we have measured to be roughly once every 102–103 ms for the APs in our network.

Manipulating the PIT timer in this way may conceivably cause unintended timing artifacts in the station's operation. Therefore, we have developed an alternative implementation that uses Linux's high-resolution timers to schedule the VoIP slots and have observed a similar degree of synchronization. However, the results in this paper are based on manipulating the PIT timer.

3.1.3 Controlling transmission timing

An obvious complication with our scheme is that when a TDMA slot starts, a station other than the station that has been assigned the slot may already be transmitting a frame. At 11 Mbps a maximum-sized IP packet (1500 bytes) together with ACK will take 1376 μs, potentially delaying the station by that time from the start of its slot into the next slot.[2] In addition, the VoIP station may repeatedly fail to capture the channel even while actively

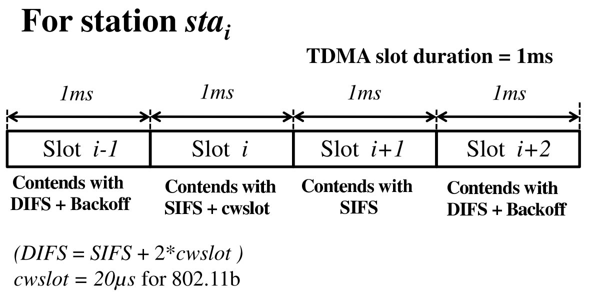

For station sta_i
TDMA slot duration = 1ms
1ms
1ms
1ms
1ms
Slot i-1
Slot i
Slot i+1
Slot i+2
Contends with DIFS + Backoff
Contends with SIFS + cwslot
Contends with SIFS
Contends with DIFS + Backoff
(DIFS = SIFS + 2*cwslot)
cwslot = 20μs for 802.11b

Figure 6: Illustration of dynamic IFS showing the various contention parameters, depending on the TDMA slot sta_i is contending in.

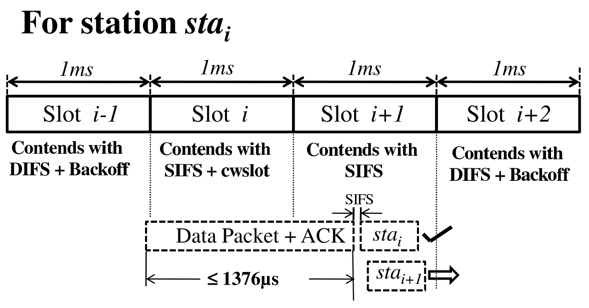


Figure 7: Dynamic IFS in the presence of other data traffic. In TDMA slot $i + 1$ sta_i wins over sta_{i+1} since it contends with SIFS rather than SIFS $+$ cwslot

contending. We address this challenge by letting the WiFi card driver adjust the way VoIP station contends for the channel during its assigned slot, a mechanism we term *dynamic IFS* (dynamic inter-frame spacing).

In standard DCF, stations contend using an inter-frame spacing of SIFS $+$ $(2 \cdot \text{cwslot})$ followed by a random backoff. (By cwslot we denote an 802.11 contention-window slot—20 μs in 802.11b—not Softspeak's 1-ms TDMA slot.) We use the two 20-μs cwslot intervals starting at SIFS and (SIFS $+$ cwslot), respectively, to (a) prioritize the VoIP traffic over non-VoIP traffic and (b) prioritize among different VoIP stations to avoid collisions. Accordingly, we let each station contend as follows: Figure 6 considers a station sta_i which is assigned TDMA slot i. During the station's assigned TDMA slot it contends with (SIFS $+$ cwslot) (and no backoff). In slot $i + 1$, it contends with SIFS (and no backoff). In any other slot it contends as specified by DCF (SIFS $+$ $(2 \cdot \text{cwslot})$ $+$ backoff).

Now let us consider the scenario as illustrated in Figure 7, in which a station sta_i in TDMA slot i is delayed into the next TDMA slot $(i + 1)$ by an ongoing transmission and assume for the moment that sta_i's packet was ready at the start of the slot i. After the transmission has ended, stations sta_i and sta_{i+1} contend for the channel. However, due to the assigned contention parameters, sta_i is guaranteed to win over station sta_{i+1}. Furthermore, after sta_i has finished transmitting and received its ACK (after 430 μs for a large-payload G.711 codec), there is still at least (2 ms - 1376 μs - 430 μs = 194 μs) for sta_{i+1} to commence its transmission and therefore not contend in TDMA slot $(i + 2)$. It can be shown that in the absence of retransmissions, as long as (a) the duration of a VoIP frame is less than one TDMA slot and (b) the duration of a bulk frame is less than two TDMA slots, station i will never contend in slot $(i + 2)$. Even if due to, e.g., 802.11 retransmissions or imperfect control of timing by Softspeak, a station ends up contending in a TDMA slot other than i or $(i + 1)$, it will do so using conventional DCF contention parameters and do no worse than without our improvements.

Figure 8 plots the transmission start times of ten VoIP stations, each assigned a separate TDMA slot, when competing against background traffic. In particular, a bulk UDP sender generates background traffic in the downlink direction to a separate wireless station. Using dynamic IFS, the slotting is clearly defined: while the bands are longer than 1 ms due to delays caused by ongoing background traffic transmissions (as explained above), the majority of transmissions do not commence more than one slot away.

The first slot (assigned to the VoIP station plotted in the first column of Figure 8) commences roughly 500 μs after the beacon time. This offset is caused by inevitable delays between the time that the beacon is generated by the AP and when it is received and processed by a station, and also between the time the station driver generates a packet for a particular slot and the time that it is transmitted. In particular, 400 μs of this time is accounted for by beacon transmission time, the remainder consisting of processing delays in the station. While some of these processing delays may vary across different stations, as subsequent figures show, the delay is consistent enough across multiple stations with the same hardware configuration that a station's synchronization can be tuned for that hardware.

3.2 Downlink aggregation

Downlink aggregation introduces an aggregator component that is placed at or before the WiFi AP (uplink from the AP). The aggregator is on-path and transparently forwards all traffic to and from the AP; non-VoIP traffic is forwarded without modification. The aggregator buffers VoIP frames destined for wireless stations and releases a frame encapsulating the buffered frames at a regular interval (every M ms, where M is the minimum packetization interval of the VoIP codecs in use.) By combining all the VoIP sessions into one packet per codec interval, downlink aggregation can virtually eliminate the marginal header and contention overhead of additional

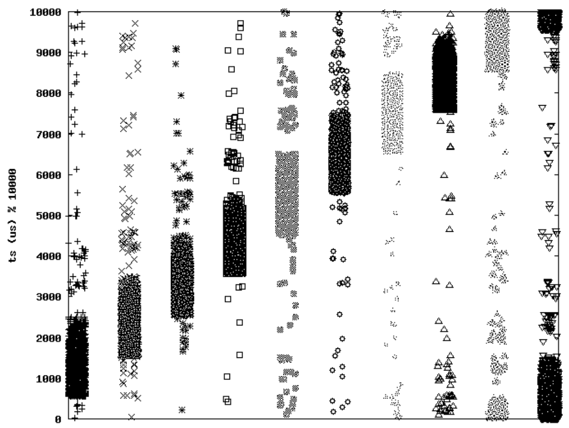


Figure 8: TDMA slotting by ten VoIP stations using dynamic IFS in the presence of UDP downlink background traffic. Each column represents a distinct VoIP station.

VoIP clients. There is a down side however: when the aggregator buffers a packet, it adds a constant delay of $M/2$ ms in expectation, e.g., 5 ms given a 10-ms codec.

When a new Softspeak VoIP session starts up (or when the station roams to a different AP) it registers with the aggregator node, which we implement on a separate Linux machine. When the aggregator receives a downlink packet addressed to a registered VoIP client, it buffers the packet and combines it with all other buffered packets into a single encapsulated packet that it sends out at fixed intervals (e.g., 10 ms for G.729). The aggregator node uses the IP header information from the most recently heard uplink packet (say from station $S1$) to construct a new frame. Addressing the packet to $S1$ increases the likelihood that the packet will be acknowledged by a currently active VoIP client. We define an aggregation header that stores the set of destinations and original IP packet lengths for each station. The aggregation header is prepended to the UDP header and packet payload for $S1$, and then the respective IP and UDP headers and payloads for the remaining buffered VoIP packets are appended.

In contrast to previous proposals [23], we address the aggregated frame to only one of the VoIP stations; we configure the WiFi interface of each of the VoIP stations to be in promiscuous mode to allow them to receive the aggregated packets regardless of the destination. The client passes aggregated packets to the Softspeak module that de-encapsulates the packet, extracts the portion meant for the current station, and passes it up the networking stack. Because the aggregated packet is addressed to only one station, there will be at most one MAC-layer acknowledgment. Wang *et al.*, on the other hand, propose the use of multicast in order to eliminate the MAC ACK frame. We preserve the ACK frame for two pragmatic reasons. First, in our experience, while

Card	CWmin	CWmax	Retry limit
Ralink RT2560F	8	256	8
Atheros AR5212	32	32	11
Avaya AP-8	16	16	11

Table 1: 802.11b contention parameters measured for our wireless hardware.

obviously unable to eliminate all loss, the single ACK frame is a cost-effective mechanism to protect the aggregated packet against many collisions. Secondly, and perhaps more importantly, commodity access points typically transmit multicast frames only at a multiple of the beacon interval to inter-operate with clients in power-save mode, introducing intolerable delay.

4 Evaluation

We now evaluate the effect of downlink aggregation and uplink TDMA, both independently and in concert. In particular, we show that (a) our schemes significantly increase the available channel capacity while usually maintaining—and sometimes improving—VoIP call quality, and (b) our implementation of Softspeak is close to optimal in terms of throughput improvement.

4.1 Experimental testbed

The wireless infrastructure in our building is a managed 802.11b/g deployment of enterprise-class Avaya AP-8 access points. There are multiple APs per floor which are configured to orthogonal channels to increase spatial diversity. We configure eleven Soekris net4801 boxes to act as VoIP stations. Each has two mini-PCI wireless cards: an Atheros AR5212 chipset-based card and an Ralink RT2560F-based interface. The net4801 is a single-board based computer with a 266-MHz CPU running the Linux operating system. To simplify our experiments, we emulate VoIP traffic using `iperf`. We use `iperf` to generate UDP traffic that mimics a commonly used VoIP codec, G.729, at 10-ms inter-packet intervals. RTS/CTS is disabled on all Soekris boxes and APs. All experiments are conducted late at night to minimize background wireless activity.

We employ ten commodity PCs connected over wired gigabit Ethernet as endpoints for the (emulated) VoIP traffic generated by the Soekris boxes. Essentially, each PC-Soekris pair serves as a distinct bi-directional VoIP call. One additional PC-Soekris pair conducts a bulk transfer (TCP or UDP) to measure the residual capacity of the wireless channel in the presence of the VoIP traffic. The TCP receive-window size is configured to be large enough that our TCP transfers are never receive-window limited. Unless otherwise noted, bulk transfer is conducted through the Atheros card, while the Ralink interfaces send and receive VoIP traffic.

Table 1 reports the default contention parameters for the various devices in our testbed as measured by the Jigsaw wireless monitoring infrastructure [5]. We note that neither the Atheros card nor the Avaya AP appears to double its contention window size on retries, in contrast with the default behavior specified by 802.11.

4.2 Results for 802.11b

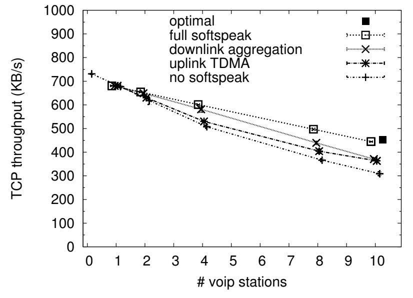
Figures 9 and 10 compare bulk throughput and VoIP call quality across all combinations of applying uplink TDMA and/or downlink aggregation in 802.11b, for TCP uplink and downlink. The results for UDP bulk uplink (not shown) are similar to those of TCP uplink. We discuss the case of UDP bulk downlink in Section 4.3. The most important conclusions are that (a) applying a combination of uplink TDMA and downlink aggregation improves residual bulk throughput, in some cases drastically, (b) with one exception, call quality is preserved or greatly improved, (c) applying only one of uplink TDMA or downlink aggregation does not achieve these results across all three cases of bulk traffic load.

We summarize the benefits of Softspeak (combined uplink TDMA and downlink aggregation) over 802.11, for the case of ten VoIP sessions, as follows:

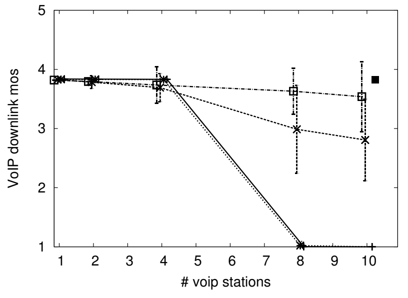
TCP uplink and UDP uplink: Capacity increases by around 50% (Figure 9(a)). Downlink VoIP improves from being completely unusable for VoIP to being usable (Figure 9(b)). The bulk of this improvement comes from a reduction in downlink loss rate (from 55% to 4.8%) by downlink aggregation. However, uplink TDMA contributes significantly by further reducing the downlink loss rate (to 1.8%), resulting in a substantial increase in MOS. For uplink VoIP (Figure 9(c)) most of the MOS improvement comes from downlink aggregation, which reduces the RTT from over 400 ms to below 25 ms by reducing queuing at the AP.[3]

TCP downlink: Capacity multiplies 4.8 times (380% increase) from 92 KB/s to 445 KB/s (Figure 10(a)). Unfortunately, VoIP downlink MOS degrades somewhat (Figure 10(b)). On closer examination, we find that downlink MOS suffers from an increased loss rate from downlink aggregated packets: since Softspeak's downlink aggregation scheme receives link-layer acknowledgments from only one VoIP client, only frame losses experienced by that client result in retransmission. Frame corruption experienced by other clients remains unnoticed. We address this issue when we present our results for 802.11g (Section 4.4) where higher frame rates may further increase the probability of frame corruption.

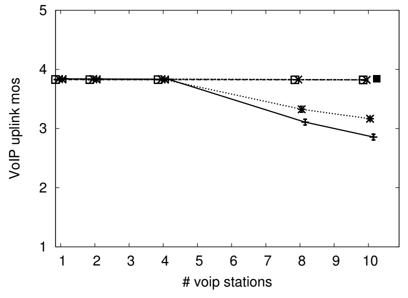
While these results show that Softspeak improves the efficiency of 802.11b networks in the presence of VoIP



(a) Capacity

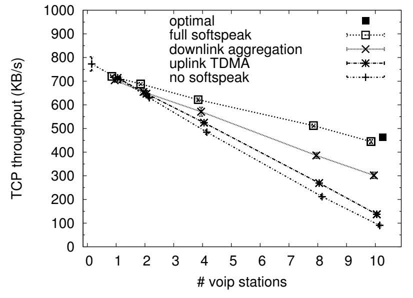


(b) Downlink MOS

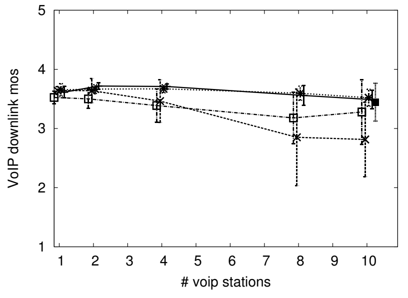


(c) Uplink MOS

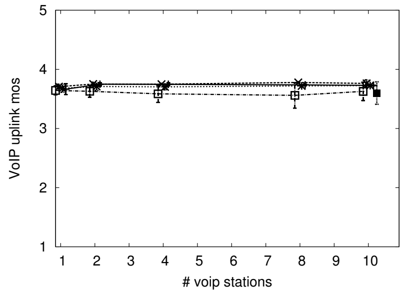
Figure 9: Impact of a varying number of VoIP stations in combination with TCP uplink traffic (802.11b).



(a) Capacity



(b) Downlink MOS.



(c) Uplink MOS

Figure 10: Impact of a varying number of VoIP stations in combination with TCP downlink traffic (802.11b).

in terms of residual TCP capacity (while mostly preserving VoIP call quality), an important question is whether further improvements to our implementation could be made. For example, it might be the case that our implementation of uplink TDMA lacks sufficient control of VoIP packet scheduling, causing collisions. An optimal implementation (e.g., one that is implemented in the 802.11 hardware or firmware) might do a better job at controlling the emission of frames according to the TDMA schedule.

To investigate to what extent further improvements may be made to our implementation (but while remaining faithful to Softspeak), we compare our results with those based on an emulation of an optimal implementation. We emulate downlink aggregation by replacing the individual VoIP senders that generate downlink VoIP traffic by a *single* sender that generates packets of the size produced by the downlink aggregator, eliminating any jitter and loss potentially caused by the downlink aggregator. Furthermore, downlink packets are sent to, and their loss rate measured at, a single VoIP station, eliminating any losses due to imperfect overhearing. We emulate uplink TDMA by replacing the VoIP stations by a single VoIP station that sends packets on behalf of all VoIP stations, in other words, it sends packets at ten times the codec rate. The single VoIP station naturally serializes the transmission of uplink VoIP packets, thereby eliminating any collision among VoIP stations. To minimize the probability of colliding with other traffic, it uses SIFS without backoff. In Figures 9 and 10 the results of the emulation are plotted as an 'optimal' point for ten VoIP clients. In terms of capacity and uplink MOS, Softspeak achieves close to what is optimally achievable. For downlink MOS, consistent with our earlier observation, Softspeak performs worse than optimal due to imperfect overhearing. However, note that in Figure 10(b) even optimal Softspeak's downlink MOS is worse than that of 'no softspeak'. This may be expected, given that (optimal) Softspeak enables TCP traffic to considerably increase network resource usage. For example, we measure a 25% increase in RTT (as well as an increased RTT variance) due to a higher AP queue occupation, which in turn explains the higher loss rate of downlink VoIP traffic.

4.3 UDP and 802.11e

While Softspeak can improve the capacity available for bulk UDP downlink traffic in 802.11b networks (Table 2), it cannot simultaneously reduce the high VoIP downlink loss rate that result from competing with a CBR UDP flow. These losses are caused by the AP queue filling with bulk UDP downlink traffic, combined with the fact that UDP does not respond to increasing loss and delay. Similarly, when replacing a single bulk

Metric	No Spk	Spk	Spk+Prio
Downlink bulk tput (KB/s)	375	605	561
Downlink VoIP loss rate	67%	61%	<0.1%
Uplink VoIP loss rate	0.82%	<0.1%	<0.1%

Table 2: The effectiveness of combining Softspeak (Spk) with prioritization (Prio) in the presence of ten VoIP stations and downlink bulk UDP traffic (802.11b, simulated). (UDP throughput without VoIP is 924 KB/s.)

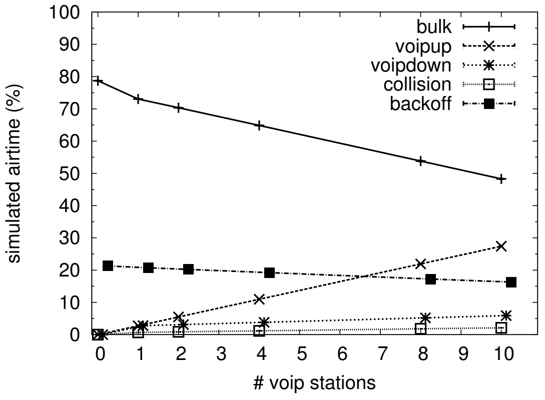


Figure 11: Simulated Softspeak airtime usage versus the number of active VoIP streams, in the presence of 802.11b UDP uplink bulk traffic (c.f. Figure 4).

TCP stream by a sufficiently large number of bulk TCP streams, the AP queue fills up with TCP packets causing large delay. These losses and delays can only be ameliorated by adding prioritization at the AP: (aggregated) VoIP packets would therefore not be dropped regardless of the amount of non-VoIP traffic buffered at the AP. Luckily, prioritization is part of the 802.11e standard.

4.3.1 Prioritization

Unfortunately, our testbed hardware cannot simultaneously support 802.11e (supported only by the Atheros chipset) and Softspeak (which is currently only implemented for the Ralink interfaces). We therefore evaluate Softspeak combined with 802.11e-like prioritization at the AP using our simulator. Consistent with our results in Section 2.3.2, our simulator produces results similar to those measured experimentally for the case of UDP without prioritization for the combination of uplink TDMA and downlink aggregation, and we therefore believe that we can extrapolate to the case of AP prioritization. Table 2 shows that when we combine Softspeak with prioritization, we not only achieve a 47% improvement on downlink bulk UDP capacity, but also improve VoIP loss rate compared to the baseline.

4.3.2 Airtime utilization

Implementing Softspeak in our simulator also allows us to isolate the source of our performance improvement. Figure 11 shows the simulated airtime plot corresponding to Figure 4, but with uplink TDMA and downlink aggregation enabled (and no prioritization). The figure indicates that we have achieved our objective of converting almost all time spent on downlink framing overhead and on collision into bulk data capacity. Consistent with the reduction in collision airtime we have also reduced the collision rate, thereby improving loss rate, jitter, and as a result, VoIP call quality and TCP throughput.

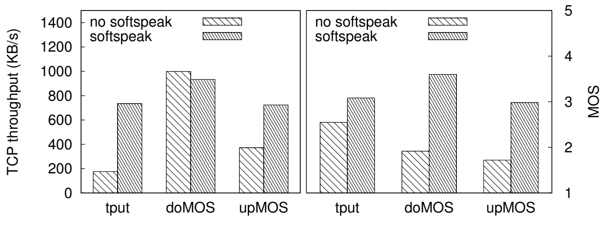
Softspeak enabled	No measures	Fixed=11b	Fixed=11b, optout
No	3.7 ± 0.095		
Yes	2.8 ± 1.0		
Yes, fixed Station 1	3.4 ± 0.63	3.5 ± 0.23	
Yes, fixed Station 2	2.7 ± 1.0	3.2 ± 0.81	3.5 ± 0.31

Table 3: Downlink aggregation losses in the presence of TCP downlink traffic (802.11g protected mode). The values given are the average and standard deviation MOS across all downlink VoIP sessions.

4.4 Results for 802.11g

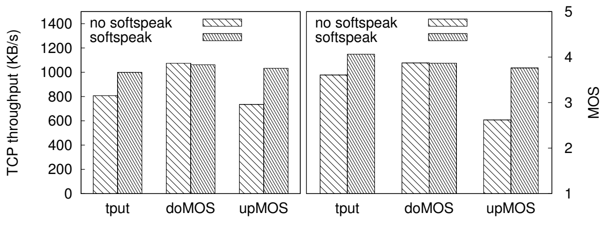
For 802.11g we observe that Softspeak as currently described makes significant improvements in capacity (24–32% for ten VoIP stations), while maintaining or lowering jitter and VoIP uplink loss to negligible levels. Recall that when 802.11g runs in protected mode, TCP downlink capacity suffers tremendously in the presence of VoIP. Using Softspeak we are able to triple (increase by 200%) the TCP downlink capacity for ten VoIP stations. However, Softspeak also introduces significant downlink VoIP loss, rising to 30% for some stations, where in some cases virtually none was experienced without enabling Softspeak. In the case of 802.11g protected mode this reduces MOS from 3.7 to 2.8 on average and substantially increases the variance of MOS (Table 3, *no measures*).

As noted in Section 4.2 for 802.11b, downlink aggregation is susceptible to frame corruption by any receiver that is not the link-layer recipient of the aggregated packet, and the higher rates of 802.11g only increase the likelihood of frame corruption. Our solution to this problem is three-fold. First, we observe that judiciously selecting a fixed station as the destination for aggregated packets may greatly alleviate loss: picking a station that consistently experiences frame corruption causes the AP to often retransmit aggregated frames thereby increasing each station's probability of receiving a correct copy. For a particular choice of station (Station 1 in Table 3), we observe that the average downlink loss rate consistently reduces to below 2%, resulting in an average MOS of 3.4. However, the MOS variance remains high. Second, the selected station can be made to associate with the AP at a lower rate, causing aggregated packets to be transmitted at the lower rate and further reducing frame corruption. To test this, we force Station 1 to associate in 802.11b mode (*fixed=11b* in Table 3) and



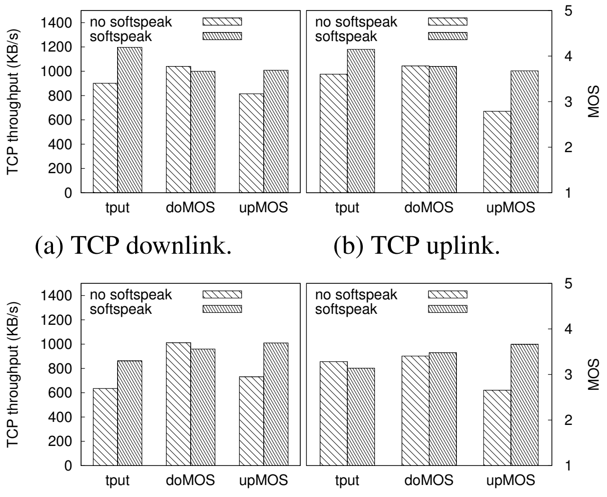
(a) TCP downlink. (b) TCP uplink.

Figure 12: 10-ms code VoIP in combination with TCP traffic (802.11g protected mode, two stations opt out).



(a) TCP downlink. (b) TCP uplink.

Figure 13: 20-ms codec VoIP in combination with TCP traffic (802.11g protected mode, two stations opt out).



(a) TCP downlink. (b) TCP uplink.

(c) TCP downlink+Web. (d) TCP uplink+Web.

Figure 14: VoIP in combination with bulk TCP traffic (802.11g protected mode, no opt-out). Only five VoIP stations are active. In (c) and (d) the remaining five stations engage in Web traffic. The throughput measured is that of bulk TCP.

obtain a MOS of 3.5 as well as reduced variance. Note that to avoid condemning one of the stations to low-rate communication, a dummy 802.11 receiver can be added to the downlink aggregator box (or placed separately) and made to associate at the lower rate.

Our third measure is to have any remaining bad receivers opt out of the downlink portion of Softspeak (not evaluated for Station 1). By de-registering with the aggregator, these clients receive separate VoIP frames as in the non-aggregated case (while continuing to measure loss rate from received aggregated packets to help decide whether and when to re-register). Note that these stations can still participate in uplink aggregation. To demonstrate that such a scheme can gracefully address this situation in practice, we evaluate all three measures when making a poor choice for the fixed station: Station 2 in Table 3, which gives a low MOS value of 2.7. After making the fixed station associate in 802.11b (improving average MOS to 3.2), we find that two stations consistently experience a high loss rate and MOS. Once these two stations opt out of downlink aggregation, we arrive at a MOS of 3.5 with low variance (*fixed=11b,optout*).

Of course, several of these measures have the potential of sacrificing much of the bulk traffic throughput gains that were obtained from downlink aggregation in the first place. We evaluate both TCP throughput and VoIP quality based on the above Station 2 and while applying all three measures. Downlink TCP throughput (Figure 12(a)) does not much suffer much from these countermeasures: Softspeak continues to more than triple TCP downlink throughput. However, the resulting uplink TCP throughput (781KB/s, Figure 12(b)) is 12% less than the throughput achievable by Softspeak without enabling these countermeasures (not shown). Nevertheless, even with the countermeasures enabled Softspeak is able to achieve a significant improvement on residual throughput (34%) on TCP uplink traffic. For both TCP downlink and uplink Softspeak mostly maintains or significantly improves VoIP quality. For completeness, Figure 13 presents the corresponding results when all clients use a 20-ms G.729 codec. As expected, Softspeak delivers less benefit in terms of throughput increase, yet remains critical for uplink VoIP call quality.

4.5 Softspeak and Web traffic

So far we have focused on Softspeak's impact on bulk traffic, without other traffic present. In reality, of course, one may expect a diverse traffic mix. We next evaluate how our results change in the presence of Web traffic, by running an equal number of VoIP clients and Web clients in combination with a bulk TCP stream, where each of the Web clients repeatedly downloads the front page of `cnn.com` (630 KB). Note that the size of our testbed limits us to five VoIP clients and five Web clients, and the magnitude of improvement is expected to be smaller than for a larger number of clients. In Figure 14, we plot Softspeak's improvements before (a and b) and after (c and d) adding Web traffic. Comparing the two scenarios we find that, independent of the presence of Web traffic, Softspeak (a) raises uplink MOS to an identical level, (b) roughly maintains downlink MOS, and (c) improves downlink TCP throughput to the same degree (roughly 35%). However, we also find that the gains made by Softspeak on TCP uplink throughput diminish in the presence of Web traffic. In summary, it appears that, with the exception of TCP uplink throughput, Softspeak's improvements on the efficiency of the network are maintained, even when Web traffic is present.

5 Limitations and discussion

The scalability of Softspeak is limited by the number of slots available for uplink TDMA, i.e., ten clients in 802.11b (given 10-ms inter-packet interval VoIP codecs). In 802.11g (non-protected mode) the number of clients can be raised to twenty by choosing 500-μs TDMA slots (assuming a 48-Mbps sending rate). In addition, the number of available slots can be further doubled in the case that only 20-ms codecs are in use.

Softspeak relies on clients overhearing each other's VoIP communication to perform downlink aggregation. Therefore, if a WLAN uses a WiFi encryption protocol such as WPA2, downlink aggregation is no longer possible. Uplink TDMA, on the other hand, is not affected by encryption. Protocols encrypted above the MAC layer, such as Skype, can continue to take advantage of Softspeak's downlink aggregation, as long as they allow some way of being detected as VoIP.

Another consequence of downlink aggregation is that Softspeak places a station's interface in promiscuous mode, raising concerns of increased power usage. Stations engaging in VoIP traffic cannot currently benefit from 802.11 power saving mode (PSM) with or without Softspeak enabled, since PSM's duty cycling granularity is too coarse (a multiple of the beacon interval time). However, Softspeak introduces a well-defined schedule, both for uplink (TDMA) and downlink traffic (the aggregator's schedule), even in the face of jitter caused by VoIP applications or the wide-area network. Future rapid-duty cycling hardware may be able to exploit Softspeak to provide more fine-grained power savings.

VoIP silence suppression may go some way towards mitigating the impact of VoIP, decreasing the need for Softspeak. However, it appears that silence suppression is not universally implemented or supported by all codecs. For example, while monitoring a G.711 call between a Linksys VoIP phone and a softphone (Twinkle), we observe no change to inter-packet time in traffic sent by either side, even when the sender is muted. The same applies when we monitor a SkypeOut call. On the other hand, we have observed that Skype-to-Skype calls do employ silence suppression by lowering the sending rate, rather than eliminating traffic completely.

6 Related work

Researchers have studied VoIP call quality in wireless networks and attempted to quantify how many VoIP calls traditional WiFi networks can handle while maintaining various quality-of-service (QoS) metrics. These range from analytical and simulation-based studies [3, 14, 22, 25] to those that validate findings by measurements on actual experimental testbeds [4, 9, 20]. While precise findings vary, all studies agree that the effective VoIP capacity of a WiFi network is less than one might expect given the bandwidth usage of typical VoIP streams.

The poor performance of VoIP in WiFi networks is not protocol specific, but is symptomatic of a general issue with any CSMA (carrier-sense, multiple-access) network: channel access and arbitration becomes increasingly inefficient as load (in terms of number of attempted channel accesses) increases. TDMA can be far more efficient under heavy load. Indeed, 802.11 includes both a point coordination function (PCF) mode and a hybrid coordination function (HCF) mode, in which the AP explicitly arbitrates channel access. Unfortunately, very few deployed 802.11 networks employ these modes.

If one considers modifying the hardware, a variety of options exist. For example, researchers have proposed modifying 802.11 PCF [3, 11] as well as alternative ways of implementing 802.11e-like functionality [22]. Of course, non-backwards compatible modifications do not address the issue facing today's networks. Accordingly, researchers have proposed a variety of explicit time-slotting mechanisms, both within the context of infrastructure-based networks [10, 15, 16, 18, 21] and multi-hop mesh networks [13, 17].

MadMAC [18], ARGOS [13], and the Overlay MAC Layer (OML) [17] each propose to enable time-slotting on the order of 20 ms. Snow *et al.* [21] present a similar TDMA-based approach to power savings where each slot is of the order of 100 ms and requires changes at the access points themselves. These scheduling granularities are too coarse to effectively support most VoIP codecs. While software TDMA (STDMA) [10] proposes to do TDMA for all traffic, they focus particularly on the performance of VoIP. Their approach is a substantial and backward-incompatible modification to 802.11 that requires accurate clock synchronization. More significantly, each of the above schemes require the entire network to support the new TDMA architecture with no support for unmodified clients.

Over and above TDMA mechanisms, the SoftMAC [15] and MultiMAC [8] projects also suggest modifications to 802.11 MAC behavior, including changing the ACK timing and modifying back-off parameters. The authors do not provide many details about their implementations, however, nor do they evaluate their scheme with deadline-driven VoIP traffic.

Focusing explicitly on improving the performance of VoIP traffic in mixed-use networks, various proposals have suggesting prioritizing VoIP traffic [4, 25], notably a commercial product, Spectralink Voice Priority (SVP) [2]. SVP prioritizes downlink VoIP packets in the AP transmit queue and does not back-off when attempting VoIP transmissions. While we leverage similar optimizations, SVP does not do scheduling, thereby increasing collision rate due to the lack of back-off.

Finally, several studies [12, 19] have shown using simulations that prioritizing traffic, using modified contention parameters, can lead to fairness and better resource allocation in both uplink and downlink directions. In contrast to our work, these proposals aim only to balance uplink and downlink traffic flows and do not evaluate TCP traffic in combination with VoIP traffic.

7 Conclusion

As WiFi-capable smartphone handsets become more popular, the number of simultaneous VoIP users is likely to increase dramatically in WiFi hotspots and enterprise networks. While previous work has aggregated downlink VoIP traffic, it has focused on improving VoIP call quality in the face of competing best-effort traffic, but has ignored the impact of a large number of simultaneous VoIP sessions on the residual capacity of the network.

We present Softspeak, a set of backward-compatible changes to WiFi that address contention and framing overhead. We show that our dynamic IFS contention scheme, combined with downlink aggregation, dramatically reduces the impact of VoIP on network capacity yet improves call quality. Our project page (including audio samples) is at `http://sysnet.ucsd.edu/wireless/softspeak/`.

Acknowledgments

We would like to thank Mikhail Afanasyev, Alvin AuYoung, Keon Jang, Shaan Mahbubani, Sue Moon, Stefan Savage, Geoff Voelker, the anonymous reviewers, and our shepherd Dina Katabi for their comments. This work is funded in part by NSF CAREER grant CNS-0347949.

References

[1] iCall for the iPhone (beta). `http://www.icall.com/iphone/`.

[2] Spectralink Inc - Spectralink Voice Priority (SVP). `http://www.spectralink.com/files/literature/SVP_white_paper.pdf`.

[3] J. Al-Karaki and J. Chang. A simple distributed access control scheme for supporting QoS in IEEE 802.11 wireless LANs. *Wireless Communications and Networking Conference, 2004. WCNC. 2004 IEEE*, 1, 2004.

[4] F. Anjum, M. Elaoud, D. Famolari, A. Ghosh, R. Vaidyanathan, A. Dutta, P. Agrawal, T. Kodama, and Y. Katsube. Voice performance in WLAN networks-an experimental study. *Global Telecommunications Conference, 2003. GLOBECOM'03. IEEE*, 6, 2003.

[5] Y.-C. Cheng, M. Afanasyev, P. Verkaik, P. Benkö, J. Chiang, A. C. Snoeren, S. Savage, and G. M. Voelker. Automating cross-layer diagnosis of enterprise wireless networks. In *Proceedings of the ACM SIGCOMM Conference*, Kyoto, Japan, Aug. 2007.

[6] R. G. Cole and J. H. Rosenbluth. Voice over ip performance monitoring. *SIGCOMM Comput. Commun. Rev.*, 31(2):9–24, 2001.

[7] L. Ding and R. Goubran. Speech quality prediction in voip using the extended e-model. *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, 7:3974–3978 vol.7, Dec. 2003.

[8] C. Doerr, M. Neufeld, J. Fifield, T. Weingart, D. Sicker, and D. Grunwald. MultiMAC-An Adaptive MAC Framework for Dynamic Radio Networking. *IEEE DySPAN*, 2005.

[9] S. Garg and M. Kappes. An experimental study of throughput for udp and voip traffic in ieee 802.11b networks. *Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE*, 3:1748–1753 vol.3, 16-20 March 2003.

[10] F. Guo and T. Chiueh. Software TDMA for VoIP applications over IEEE802.11 wireless LAN. *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2366–2370, May 2007.

[11] T. Kawata, S. Shin, A. Forte, and H. Schulzrinne. Using Dynamic PCF to Improve the Capacity for VoIP Traffic in IEEE 802.11 Networks. In *Wireless Communications and Networking Conference, 2005 IEEE*, volume 3, 2005.

[12] S. Kim, B. Kim, and Y. Fang. Downlink and Uplink Resource Allocation in IEEE 802.11 Wireless LANs. *Vehicular Technology, IEEE Transactions on*, 54(1):320–327, 2005.

[13] R. R. Kompella, S. Ramabhadran, I. Ramani, and A. C. Snoeren. Cooperative packet scheduling via pipelining in 802.11 wireless networks. In *Proceedings of the Workshop on Experimental Approaches to Wireless Network Design and Analysis (E-WIND)*, pages 35–40, Philadelphia, PA, Aug. 2005.

[14] K. Medepalli, P. Gopalakrishnan, D. Famolari, and T. Kodama. Voice capacity of IEEE 802.11b, 802.11a and 802.11g wireless LANs. In *Proc. IEEE Global Telecommunications Conference*, 2004.

[15] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. SoftMACflexible wireless research platform. *Proc. HotNets-IV*.

[16] G. Paschos, I. Papapanagiotou, S. Kotsopoulos, and G. Karagiannidis. A New MAC Protocol with Pseudo-TDMA Behavior for Supporting Quality of Service in 802.11 Wireless LANs. *EURASIP Journal on Wireless Communications and Networking*, 6:76–84, 2006.

[17] A. Rao and I. Stoica. An overlay MAC layer for 802.11 networks. *Proc. of Mobile Systems, Applications and Services (Mobisys '05)*, 2005.

[18] A. Sharma, M. Tiwari, and H. Zheng. MadMAC: Building a reconfigurable radio testbed using commodity 802.11 hardware. *Proc. First IEEE Workshop on Networking Technologies for Software Defined Radio (IEEE SECON 2006 Workshop)*.

[19] S. Shin and H. Schulzrinne. Balancing uplink and downlink delay of VoIP traffic in WLANs using Adaptive Priority Control (APC). In *International Conference on Quality of Service in Heterogeneous Wired/Wireless Networks*, 2006.

[20] S. Shin and H. Schulzrinne. Experimental Measurement of the Capacity for VoIP Traffic in IEEE 802.11 WLANs. *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, pages 2018–2026, 2007.

[21] J. Snow, W. Feng, and W. Feng. Implementing a low power TDMA protocol over 802.11. *Wireless Communications and Networking Conference, 2005 IEEE*, 1, 2005.

[22] P. Wang, H. Jiang, and W. Zhuang. Capacity Improvement and Analysis for Voice/Data Traffic over WLAN. *IEEE Transactions on Wireless Communications*.

[23] W. Wang, S. C. Liew, and V. Li. Solutions to performance problems in voip over a 802.11 wireless lan. *Vehicular Technology, IEEE Transactions on*, 54(1):366–384, Jan. 2005.

[24] W. Wang, S. C. Liew, Q. Pang, and V. O. K. Li. A multiplex-multicast scheme that improves system capacity of voice-over-ip on wireless lan by 100In *ISCC '04: Proceedings of the Ninth International Symposium on Computers and Communications 2004 Volume 2 (ISCC"04)*, pages 472–477, Washington, DC, USA, 2004. IEEE Computer Society.

[25] J. Yu, S. Choi, and J. Lee. Enhancement of VoIP over IEEE 802.11 WLAN via Dual Queue Strategy. *Proc. IEEE ICC*, 4, 2004.

Notes

[1]In G.729 each direction has a 10-ms inter-packet arrival, an eight-byte voice payload, and twelve additional bytes of RTP header. Variants of G.729 also run at longer inter-packet times and/or increased voice payload sizes.

[2]We assume short preambles throughout the paper.

[3]Note that delay in one direction affects MOS in both directions.

Block-switched Networks: A New Paradigm for Wireless Transport

Ming Li, Devesh Agrawal, Deepak Ganesan, and Arun Venkataramani
{*mingli, dagrawal, dganesan, arun*}*@cs.umass.edu*
University of Massachusetts Amherst

Abstract

TCP has well-known problems over multi-hop wireless networks as it conflates congestion and loss, performs poorly over time-varying and lossy links, and is fragile in the presence of route changes and disconnections.

Our contribution is a clean-slate design and implementation of a wireless transport protocol, Hop, that uses *reliable per-hop block transfer* as a building block. Hop is 1) fast, because it eliminates many sources of overhead as well as noisy end-to-end rate control, 2) robust to partitions and route changes because of hop-by-hop control as well as in-network caching, and 3) simple, because it obviates complex end-to-end rate control as well as complex interactions between the transport and link layers. Our experiments over a 20-node multi-hop mesh network show that Hop is dramatically more efficient, achieving better fairness, throughput, delay, and robustness to partitions over several alternate protocols, including gains of more than an order of magnitude in median throughput.

1 Introduction

Wireless networks are ubiquitous, but traditional transport protocols perform poorly in wireless environments, especially in multi-hop scenarios. Many studies have shown that TCP, the universal transport protocol for reliable transport, is ill-suited for multi-hop 802.11 networks. There are three key reasons for this mismatch. First, multi-hop wireless networks exhibit a range of loss characteristics depending on node separation, channel characteristics, external interference, and traffic load, whereas TCP performs well only under low loss conditions. Second, many emerging multi-hop wireless networks such as long-distance wireless mesh networks, and delay-tolerant networks exhibit intermittent disconnections or persistent partitions. TCP assumes a contemporaneous end-to-end route to be available and breaks down in partitioned environments [13]. Third, TCP has well-known fairness issues due to interactions between its rate control mechanism and CSMA in 802.11, e.g., it is common for some flows to get completely shut out when many TCP/802.11 flows contend simultaneously [37]. Although many solutions (e.g. [16, 32, 38]) have been proposed to address parts of these problems, these have not gained much traction and TCP remains the dominant available alternative today.

Our position is that a clean slate re-design of wireless transport necessitates re-thinking three fundamental design assumptions in legacy transport protocols, namely that 1) a packet is the unit of reliable wireless transport, 2) end-to-end rate control is the mechanism for dealing with congestion, and 3) a contemporaneous end-to-end route is available. The use of a small packet as the granularity of data transfer results in increased overhead for acknowledgements, timeouts and retransmissions, especially in high contention and loss conditions. End-to-end rate control severely hurts utilization as end-to-end loss and delay feedback is highly unpredictable in multi-hop wireless networks. The assumption of end-to-end route availability stalls TCP during periods of high contention and loss, as well as during intermittent disconnections.

Our transport protocol, Hop, uses *reliable per-hop block transfer* as a building block, in direct contrast to the above assumptions. Hop makes three fundamental changes to wireless transport. First, Hop replaces packets with *blocks*, i.e., large segments of contiguous data. Blocks amortize many sources of overhead including retransmissions, timeouts, and control packets over a larger unit of transfer, thereby increasing overall utilization. Second, Hop does not slow down in response to erroneous end-to-end feedback. Instead, it uses hop-by-hop backpressure, which provides more explicit and simple feedback that is robust to fluctuating loss and delay. Third, Hop uses hop-by-hop reliability in addition to end-to-end reliability. Thus, Hop is tolerant to intermittent disconnections and can make progress even when a contemporaneous end-to-end route is never available, i.e., the network is always partitioned [3].

Large blocks introduce two challenges that Hop converts into opportunities. First, end-to-end block retransmissions are considerably more expensive than packet retransmissions. Hop ensures end-to-end reliability through a novel retransmission scheme called *virtual retransmissions*. Hop routers cache large in-transit blocks. Upon an end-to-end timeout triggered by an outstanding block, a Hop sender sends a token corresponding to the block along portions of the route where the block is already cached, and only physically retransmits blocks along non-overlapping portions of the route where it is not cached. Second, large blocks as the unit of transmission exacerbates hidden terminal situations. Hop uses a novel *ack withholding* mechanism that sequences block transfer across multiple senders transmitting to a single receiver. This lightweight scheme reduces collisions in hidden terminal scenarios while incurring no additional control overhead.

In summary, our main contribution is to show that reliable per-hop block transfer is fundamentally better than the traditional end-to-end packet stream abstraction through the design, implementation, and evaluation of Hop. The individual components of Hop's design are simple and perhaps right out of an undergraduate networking textbook, but they provide dramatic improvements in combination. In comparison to the best variant of 1) TCP, 2) Hop-by-hop TCP, and 3) DTN 2.5, a delay tolerant transport protocol [8],

- ► Hop achieves a median goodput benefit of 1.6× and 2.3× over single- and multi-hop paths respectively. The corresponding lower quartile gains are 28× and 2.7× showing that Hop degrades gracefully.
- ► Under high load, Hop achieves over an order of magnitude benefit in median goodput (e.g., 90× over TCP with 30 concurrent large flows), while achieving comparable or better aggregate goodput and transfer delay for large as well as small files.
- ► Hop is robust to partitions, and maintains its performance gains in well-connected WLANs and mesh networks as well as disruption-prone networks. Hop also co-exists well with delay-sensitive VoIP traffic.

2 Why reliable per-hop block transfer?

In this section, we give some elementary arguments for why reliable per-hop block transfer with hop-by-hop flow control is better than TCP's end-to-end packet stream with end-to-end rate control in wireless networks.

Block vs. packet: A major source of inefficiency is transport layer per-packet overhead for timeouts, acknowledgements and retransmissions. These overheads are low in networks with low contention and loss but increase significantly as wireless contention and loss rates increase. Transferring data in blocks as opposed to packets provides two key benefits. First, it amortizes the overhead of each control packet over larger number of data packets. This allows us to use additional control packets, for example, to exploit in-network caching, which would be prohibitively expensive at the granularity of a packet. Second, it enables transport to leverage link-layer techniques such as 802.11 burst transfer capability [1], whose benefits increase with large blocks.

Transport vs. link-layer reliability: Wireless channels can be lossy with extremely high raw channel loss rates in high interference conditions. In such networks, the end-to-end delivery rate decreases exponentially with the number of hops along the path, severely degrading TCP throughput. The state-of-the-art response today is to use a sufficiently large number of 802.11 link-layer acknowledgements (ARQ) to provide a reliable channel abstraction to TCP. However, 802.11 ARQ 1) interacts poorly with TCP end-to-end rate control as it increases RTT variance, 2) increases per-packet overhead due to more carrier sensing, backoffs, and acknowledgments, especially under high contention and loss (in §5.1.1, we show that 802.11b ARQ has 35% overhead). Note that TCP's woes cannot be addressed by just setting the 802.11 ARQ limit to a large value as it would reduce the overall throughput by disproportionately using the channel for transmitting packets over bad links. Unlike TCP, Hop relies solely on transport-layer reliability and avoids link-layer retransmissions for data, thereby avoiding negative interactions between the link and transport layers.

Hop-by-hop vs. end-to-end congestion control: Rate control in TCP occurs in response to end-to-end loss and delay feedback reported by each packet. However, end-to-end feedback is error-prone and has high variance in multi-hop wireless networks as each packet observes significantly different wireless interference across different contention domains along the route. This variance hurts TCP's utilization as: 1) its window size shrinks conservatively in response to loss, and 2) it experiences frequent retransmission timeouts when no data is sent.

Our position is that fixing TCP's rate control algorithm in environments with high variability is fundamentally difficult. Instead, we circumvent end-to-end rate control, and replace it with hop-by-hop backpressure. Our approach has two key benefits: 1) hop-by-hop feedback is more robust than end-to-end feedback as it involves only a single contention domain, and 2) block-level feedback provides an aggregated link quality estimate that has less variability than packet-level feedback.

In-network caching: The use of reliable per-hop block transfer enables us to exploit caching at intermediate hops for two benefits. First, caching obviates redundant retransmissions along previously traversed segments of a route. Second, caching is more robust to intermittent disconnections as it enables progress even when a contemporaneous end-to-end route is unavailable. Hop

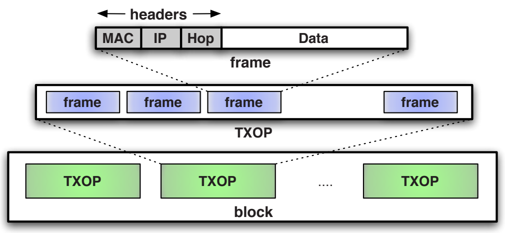


Figure 1: Structure of a block.

can also use secondary storage if needed in partitionable networks with long disconnections and reconnections.

3 Design

This section describes the Hop protocol in detail. Hop's design consists of six main components: 1) reliable per-hop block transfer, 2) virtual retransmissions for end-to-end reliability, 3) backpressure congestion control, 4) handling routing partitions, 5) ack withholding to handle hidden terminals, and 6) a per-node packet scheduler.

3.1 Reliable per-hop block transfer

The unit of reliable transmission in Hop is a *block*, i.e., a large segment of contiguous data. A block comprises a number of txops (the unit of a link layer burst), which in turn consists of a number of frames (Figure 1). The protocol proceeds in rounds until a block B is successfully transmitted. In round i, the transport layer sends a BSYN packet to the next-hop requesting an acknowledgment for B. Upon receipt of the BSYN, the receiver transmits a bitmap acknowledgement, BACK, with bits set for packets in B that have been correctly received. In response to the BACK, the sender transmits packets from B that are missing at the receiver. This procedure repeats until the block is correctly received at the receiver.

Control Overhead: Hop requires minimal control overhead to transmit a block. At the link layer, Hop disables acknowledgements for all data frames, and only enables them to send control packets: BSYN and BACK. At the transport layer, a BACK acknowledges data in large chunks rather than in single packets. The reduced number of acknowledgement packets is shown in Figure 2, which contrasts the timeline for a TCP packet transmission alongside a block transfer in Hop. For large blocks (e.g. 1 MB), Hop requires orders of magnitude fewer acknowledgements than for an equivalent number of packets using TCP with link-layer acknowledgements. In addition, Hop reduces idle time by ensuring that packets do not wait for link-layer ACKs, and at the transport layer by disabling rate control. Thus, Hop nearly always sends data at a rate close to the link capacity.

Spatial Pipelining: The use of large blocks and hop-by-hop reliability can hurt spatial pipelining since each

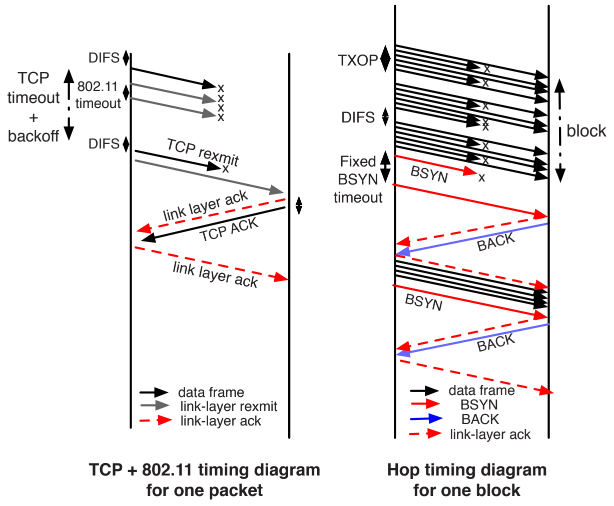


Figure 2: Timeline of TCP/802.11 vs. Hop

node waits for the successful reception of a block before forwarding it. To improve pipelining, an intermediate hop forwards packets as soon as it receives at least a txop worth of new packets instead of waiting for an entire block. Thus, Hop leverages spatial pipelining as well as the benefits of burst transfer at the link layer.

3.2 Ensuring end-to-end reliability

Hop-by-hop reliability is insufficient to ensure reliable end-to-end transmission. A block may be dropped if 1) an intermediate node fails in the middle of transmitting the block to the next-hop, or 2) the block exceeds its TTL limit, or 3) a cached block eventually expires because no next-hop node is available for a long duration.

Hop uses virtual retransmissions together with in-network caching to limit the overhead of retransmitting large blocks. Hop routers store all packets that they overhear. Thus, a re-transmitted block is likely cached at nodes along the original route until the point of failure or drop, and might be partially cached at a node that is along a new path to the destination but overheard packets transmitted on the old path. Hence, instead of retransmitting the entire block, the sender sends a *virtual retransmission*, i.e., a special BSYN packet, using the same hop-by-hop reliable transfer mechanism as for a block. Virtual retransmissions exploit caching at intermediate nodes by only transmitting the block (or parts of the block) when the next hop along the route does not already have the block cached as shown in Figure 3.

A premature timeout in TCP incurs a high cost both due to redundant transmission as well as its detrimental rate control consequence, so a careful estimation of timeout is necessary. In contrast, virtual retransmissions due to premature timeouts do little harm, so Hop simply uses the most recent round-trip time as its timeout estimate.

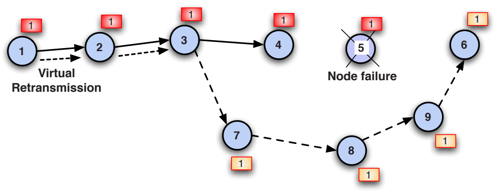


Figure 3: Virtual retransmission due to node failure.

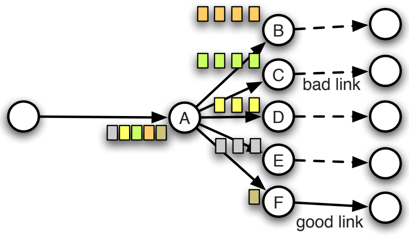


Figure 4: Example showing need for backpressure. Without backpressure, Node A would allocate $1/5$th of out-going capacity to each flow, resulting in queues increasing unbounded at nodes B through E. With backpressure, most data is sent to node F, thereby increasing utilization.

3.3 Backpressure congestion control

Rate control in response to congestion is critical in TCP to prevent congestion collapse and improve utilization. In wireless networks, congestion collapse can occur both due to increased packet loss due to contention [11], and increased loss due to buffer drops [9]. Both cases result in wasted work, where a packet traverses several hops only to be dropped before reaching the destination. Prior work has observed that end-to-end loss and delay feedback has high variance and is difficult to interpret unambiguously in wireless networks, which complicates the design of congestion control [2, 32].

Hop relies only on hop-by-hop backpressure to avoid congestion. For each flow, a Hop node monitors the difference between the number of blocks received and the number reliably transmitted to its next-hop as shown in Figure 4. Hop limits this difference to a small fixed value, H, and implements it with no additional overhead to the BSYN/BACK exchange. After receiving H complete blocks, a Hop node does not respond to further BSYN requests from an upstream node until it has moved at least one more block to its downstream node. The default value of H is set to 1 block.

Backpressure in Hop significantly improves utilization. To appreciate why, consider the following scenario where flows $1, \ldots, k$ all share the first link with a low loss rate. Assume that the rest of flow 1's route has a similar low loss rate, while flows $2, \ldots, (k-1)$ traverse a poor route or are partitioned from their destinations. Let C be the link capacity, p_1 be the end-to-end loss observed by the first flow, and p_2 be the end-to-end loss rate observed by other flows ($p_1 \ll p_2$). Without backpressure, Hop would allocate a $1/k$ fraction of link capacity to each flow, yielding a total goodput of $C\frac{((1-p_1)+(1-p_2)\cdot(k-1))}{k}$. And the number of buffered blocks at the next-hops of the latter $k-1$ flows grows unbounded. On the other hand, limiting the number of buffered blocks for each flow yields a goodput close to $C \cdot (1-p_1)$ in this example.

Why does Hop limit the number of buffered blocks, H, to a small default value? Note that the example above can be addressed simply by choosing the block corresponding to the flow with the largest differential backlog (along A-F). Indeed, classical backpressure algorithms known to achieve optimal throughput [33] work similarly. Hop limits the number of buffered blocks to a small value in order to ensure small transfer delay for finite-sized files, as well as to limit intra-path contention.

3.4 Robustness to partitions

A fundamental benefit of Hop is that it continues to make progress even when the network is intermittently partitioned. Hop transfers a blocks in a hop-by-hop manner without waiting for end-to-end feedback. Thus, even if an end-to-end route is currently unavailable, Hop continues to make progress along other hops.

The ability to make progress during partitions relies on knowing which next-hop to use. Unlike typical mesh routing protocols [23, 4], routing protocols designed for disruption-tolerance expose next-hop information even if an end-to-end route is unavailable (e.g. RAPID [3], DTLSR [7]). In conjunction with such a disruption-tolerant routing protocol, Hop can accomplish data transfer even if a contemporaneous end-to-end route is never available, i.e., the network is always partitioned.

In disruption-prone networks, a Hop node may need to cache blocks for a longer duration in order to make progress upon reconnection. In this case, the backpressure limit needs to be set taking into account the fraction of time a node is partitioned and the expected length of a connection opportunity with a next-hop node along a route to the destination (see §5.7 for an example).

3.5 Handling hidden terminals

The elimination of control overhead for block transfer improves efficiency but has an undesirable side-effect — it exacerbates loss in hidden terminal situations. Hop transmits blocks without rate control or link-layer retransmissions, which can result in a continuous stream of collisions at a receiver if the senders are hidden from each other. While hidden terminals are a problem even for TCP, rate control mitigates its impact on overall throughput. Flows that collide at a receiver observe increased loss and throttle their rate. Since different flows get different perceptions of loss, some reduce their rate more aggressively than others, resulting in most flows being completely shut out and bandwidth being devoted to one or few flows [37]. Thus, TCP is highly unfair but has good aggregate throughput.

Hop uses a novel *ack withholding* technique to mitigate the impact of hidden terminals. Here, a receiver acknowledges only one BSYN packet at any time, and withholds acknowledgement to other concurrent BSYN packets until the outstanding block has completed. In this manner, the receiver ensures that it is only receiving one block from any sender at a given time, and other senders wait their turn. Once the block has completed, the receiver transmits the BACK to one of the other transmitters, which starts transmitting its block.

Although ack withholding does not address hidden terminals caused by flows to different receivers, it offers a lightweight alternative to expensive and conservative techniques like RTS/CTS for the common single-terminal hidden terminal case. The high overhead of RTS/CTS arises from the additional control packets, especially since these are broadcast packets that are transmitted at the lowest bit-rate. The use of broadcast also makes RTS/CTS more conservative since a larger contention region is cleared than typically required [39]. In contrast, ack withholding requires no additional control packets (as BSYNs and BACKs are already in place for block transfer).

3.6 Packet scheduling

Hop's unit of link layer transmission is a txop, which is the maximum duration for which the network interface card (NIC) is permitted to send packets in a burst without contending for access [1]. Hop's scheduler leverages the burst mode and sends a txop's worth of data from each concurrent flow at a time in a round-robin manner.

Hop traffic is isolated from delay-sensitive traffic such as VoIP or video by using link-layer prioritization. 802.11 chipsets support four priority queues—voice, video, best-effort, and background in decreasing order of priority—with the higher priority queues also having smaller contention windows [1]. Hop traffic is sent using the lowest priority background queue to minimize impact on delay-sensitive datagrams.

The design choices that we have presented so far can be detrimental to delay for small files (referred to as micro-blocks) in three ways: 1) the initial BSYN/BACK exchange increases delay for micro-blocks, 2) a sender may be servicing multiple flows, in which case a micro-block may need to wait for multiple txops, and 3) ack-withholding can result in micro-blocks being delayed by one or more large blocks that are acknowledged before its turn. Hop employs three techniques to optimize delay for micro-blocks. First, micro-blocks of size less than a fixed BSYN batch threshold (few tens of KB) are sent piggybacked with the BSYN with link-layer ARQ via the voice queue. This optimization eliminates the initial BSYN/BACK delay, and avoids having to wait for a BACK before proceeding, thereby circumventing ack-withholding delay. Second, the packet scheduler at the sender prioritizes micro-blocks over larger blocks. Finally, Hop use a block-size based ack-withholding policy that prioritizes micro-blocks over larger blocks.

4 Implementation

We have implemented a prototype of Hop with all the features described in §3. Hop is implemented in Linux 2.6 as an event-based user-space daemon in roughly 5100 lines of C code. Hop is currently implemented on top of UDP (i.e., there is a UDP header in between the IP and Hop headers in each frame in Figure 1). Below, we describe important aspects of Hop's implementation.

4.1 MAC parameters

Our implementation uses the Atheros-based wireless chipset and the Madwifi open source 802.11 device driver [18], a popular commodity implementation. By default, the MadWifi driver (as well as other commodity implementations) supports the 802.11e QoS extension. However, MadWiFi supports the extension only in the access point mode, so we modify the driver to enable it in the ad-hoc mode as well. Hop uses default 802.11 settings, except for the following. The transmission opportunity (txop) for the background queue is set to the maximum value permitted by the MadWifi driver (8160 μs or roughly 8KB of data). Link-layer ARQ is disabled for all data frames sent via Hop but enabled for control packets (BSYN, BACK, etc).

4.2 Hop implementation

Parameters A large block size increases batching benefits [15], so we set the default maximum block size to 1MB. Note that this means that a Hop block is allowed to be up to 1MB in size, but may be any smaller size. Hop never waits idly in anticipation of more application data in order to obtain batching benefits. The BSYN batch threshold for micro-blocks is set to a default value of 16KB, and the backpressure limit, H, is set to 1. The virtual retransmission timeout is set to an initial value of 60 seconds and simply reset to the round-trip block delay reported by the most recent block. The TTL limit for a virtual retransmissions is set to 50 hops. In the current implementation, an intermediate Hop node keeps all the blocks that it has received in memory.

Header format: The Hop header consists of the following fields. All frames contain the `msg_type` that identifies if the frame is a data, BSYN, BACK, virtual retransmission BSYN, or an end-to-end BACK frame;

the `flow_id` that uniquely identifies an end-to-end Hop connection; and the `block_num` identifies the current block. Data frames also contain the `packet_num` that is the offset of the packet in the current block. The `packet_num` is also used to index into the bitmap returned in a BACK frame.

End-to-end connection management: Because Hop is designed to work in partitionable networks, it does not use a three-way handshake like TCP to initiate a connection. A destination node sets up connection state upon receiving the first block. The loss of the first block due to a node failure or expiry or the loss of the first end-to-end BACK is handled naturally by virtual retransmissions. In our current implementation, a Hop node tears down a connection simply by sending a FIN message and recovering state; we have not yet implemented optimizations to handle complex failure scenarios.

5 Evaluation

We evaluate the performance of Hop in a 20-node wireless mesh testbed. Each node is an Apple Mac Mini computer running Linux 2.6 with a 1.6 Ghz CPU, 2 GB RAM and a built-in 802.11a/b/g Atheros/MadWiFi wireless card. Each node is also connected via an Ethernet port to a wired backplane for debugging, testing, and data collection. The nodes are spread across a single floor of the UMass CS building as shown in Figure 5.

All experiments, except those in §5.9 and §5.10, were run in 802.11b mode with bit-rate locked at 11 Mbps. There is significant inherent variability in wireless conditions, so in order to perform a meaningful comparison, a single graph is generated by running the corresponding experiments back-to-back interspersed with a short random delay. The compared protocols are run in sequence, and each sequence is repeated many times to obtain confidence bounds.

We compare Hop against two classes of protocols: *end-to-end* and *hop-by-hop*. The former consists of 1) UDP, and 2) the default TCP implementation in Linux 2.6 with CUBIC congestion control [10]; we did not use the Westwood+ congestion control algorithm since it performed roughly 10% worse. The latter consists of 3) Hop-by-Hop TCP, and 4) DTN2.5 [8]. Hop-by-Hop TCP is our implementation of TCP with backpressure. It splits a multi-hop TCP connection into multiple one-hop TCP connections, and leverages TCP flow control to achieve hop-by-hop backpressure. Each node maintains one outgoing TCP socket and one incoming TCP socket for each flow. When the outgoing socket is full, Hop-by-Hop TCP stops reading from the incoming socket, thereby forcing TCP's flow control to pause the previous hop's outgoing socket. This "backpressure" propagates up to the source and forces the source to slow down. DTN2.5 is a reference implementation of the IEEE RFC 4838 and 5050

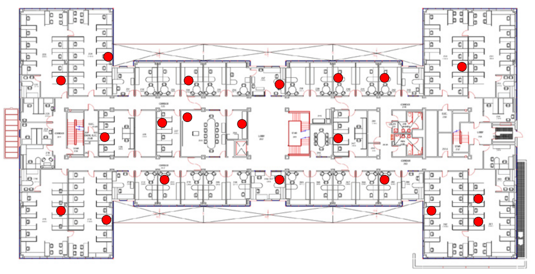


Figure 5: Experimental testbed with dots representing nodes.

from the Delay Tolerant Networking Research Group [8] that reliably transfers a bundle using TCP at each hop. Hop and UDP were set to use the same default packet size as TCP (1.5KB). In all our experiments, the delay and goodput of TCP are measured after subtracting connection setup time.

5.1 Single-hop microbenchmarks

In this section, we answer two questions: 1) What are the best 802.11 settings for link layer acknowledgments (ARQ) and burst mode (txop) for TCP and UDP?, 2) How does Hop's performance compare to that of TCP and UDP given the benefit of these best-case settings?

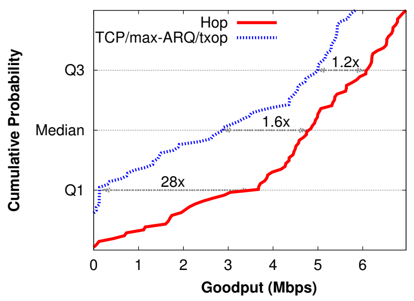


Figure 6: Experiment with one-hop flows. Hop improves lower quartile goodput by 28×, median goodput by 1.6×, and mean goodput by 1.6× over TCP with the best link layer settings.

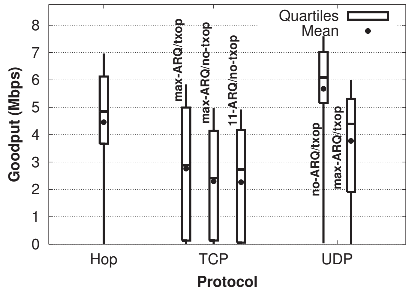


Figure 7: Experiment with one-hop flows. Box shows lower/median/upper quartile, lines show max/min, and dot shows mean. Increasing 802.11 ARQ limit and using txops helps TCP but Hop is still considerably better. UDP results show that ARQs incur significant performance overhead (35%). Hop is within 24% of UDP without ARQ (achievable goodput).

Sec.	Experiment setup	Experiment	Result: Median (Mean)
§5.1	One single-hop flow	Hop vs. TCP	1.6× (1.6×)
§5.2	One multi-hop flow	Hop vs. TCP	2.3× (2×)
		Hop vs. Hop-by-Hop TCP	2.5× (2×)
		Hop vs. DTN2.5	2.9× (3.9×)
§5.3	Many multi-hop flows	Hop vs. TCP	90× (1.25×)
		Hop vs. Hop-by-Hop TCP	20 × (1.4×)
§5.4	Performance breakdown	Base Hop	(1×)
		+ ack withholding	(2.5×)
		+ backpressure	(3.7×)
		+ ack withholding + backpressure	(4.8×)
§5.5	WLAN AP mode	Hop vs. TCP	2.7× (1.12×)
		Hop vs. TCP + RTS/CTS	2× (1.4×)
§5.6	Single small file	Hop vs. TCP	3× to 15× lower delay
	Concurrent small files	Hop vs. TCP	Comparable or lower delay
§5.7	Disruption-tolerance	Hop vs. DTN2.5	2.8× (2.9×)
§5.8	Impact on VoIP traffic	Hop vs. TCP	Slightly lower MOS score but significantly higher throughput
§5.9	Network and link-layer dynamics	Hop vs. TCP + OLSR	4× (1×)
		Hop vs. TCP + auto-rate	95× (2.4×)
		Hop vs. TCP + OLSR + auto-rate	5× (1.8×)
§5.10	Under 802.11g	Hop vs. TCP	22× (1×)
		Hop vs. TCP + auto-rate	6× (3×)

Table 1: Summary of evaluation results. All protocols above are given the benefit of burst-mode (txop) and the maximum number of link-layer retransmissions (max-ARQ) supported by the hardware.

5.1.1 Randomly picked links

In this experiment, we evaluate the single-hop performance of TCP, UDP, and Hop over 802.11 across links in our mesh testbed. The testbed has total of 56 unique links from which a random sequence of 100 links was sampled with repetition for this experiment. The average and median loss rates were 25% and 1% respectively. For each sampled link, a 10MB file is transferred using each protocol; for bad links, flows were cut off at 10 minutes, and goodput measured until the last received packet. The metric for comparison is the goodput that is measured as the total number of unique packets received at the receiver divided by the time until the last byte is received.

We compare Hop against TCP for three 802.11 settings: 1) 11 link layer retries (ARQ) with no txop, the default settings of the MadWifi driver, 2) 11 ARQ + txop, and 3) maximum permitted ARQ setting (18 for the Atheros card) + txop. We do not consider TCP with no ARQ since it (expectedly) performs poorly without 802.11 retransmissions on lossy links. We also compare against UDP under different 802.11 settings. Since UDP has no transport-layer control overhead, and transmits as fast as the card can transmit packets, it provides an upper bound on the achievable capacity on the link. For clarity of presentation, we show cumulative distributions (CDFs) for Hop and the best TCP combination and summary statistics for the other combinations (for which full distributions are available in [15]).

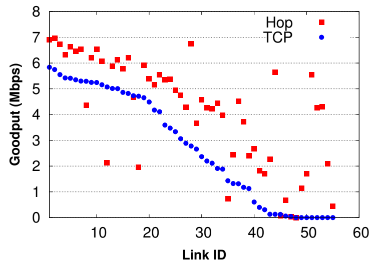
Figure 6 shows that Hop significantly outperforms TCP/max-ARQ/txop, the best TCP combination. The Q1, Q2, and Q3 gains over TCP/max-ARQ/txop TCP combination are 28×, 1.6×, and 1.2× respectively. The Q1 gain is notable and shows Hop's robust performance on poor links compared to TCP.

Figure 7 shows the summary statistics for Hop and two best TCP and UDP schemes using a box plot representation. The "box" shows the upper quartile (Q3), median (Q2) and lower quartile (Q1), and the "whiskers" show the maximum and minimum goodput. UDP/no-ARQ/txop is the best UDP combination and provides an upper bound on the achievable rate. The median Hop is about 24% lower than the achievable rate. Interestingly, turning on ARQ degrades UDP by 35% showing that ARQ in 802.11 comes at a high overhead and ARQ alone is not sufficient to fix TCP's problems.

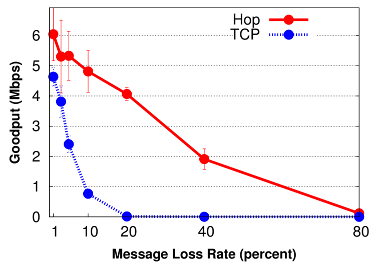
As we find that TCP performance consistently improves by using txops and ARQ with the maximum possible limit, we give TCP and its variants the benefit of txop/max-ARQ in the rest of our evaluation.

5.1.2 Graceful performance degradation

A key benefit of Hop is robustness, i.e., its performance gracefully degrades with increasing link losses and interference. To confirm this, we further analyze the data from the experiment in §5.1.1. Figure 8(a) shows the per-link throughput across the 56 links in the testbed (with multiple runs over the same link averaged) sorted by TCP goodput. Hop degrades gracefully to some of the poorest



(a) Sorted Single-hop Flows



(b) Impact of Loss.

Figure 8: Graceful degradation to adverse channel conditions. First plot shows per-link goodputs from one-hop experiment sorted in TCP order. Second plot shows controlled experiments demonstrating impact of loss. In both cases, Hop is more robust and degrades far more gracefully than TCP.

links in the testbed where TCP's throughput is near-zero. The average goodput for the worst 20 TCP flows is 334 Kbps, whereas Hop's goodput for the same flows is 2.37 Mbps, a difference of 7×.

To understand the cause of TCP's fragile behavior, we evaluate the impact of loss perceived at the transport layer on the performance of Hop and TCP. We start with a perfect link that has a near-zero loss rate and introduce loss by modifying the MadWifi device driver to randomly drop a specified fraction of incoming packets. Figure 8(b) shows that, unsurprisingly, TCP goodput drops to near-zero when loss rate is roughly 20%. Hop shows graceful near-linear degradation and is operational until the loss rate is about 80%.

5.2 Multi-hop microbenchmarks

How does Hop perform on multi-hop paths compared to existing alternatives? To study this question, we pick a sequence of 100 node pairs randomly with repetition from the testbed. Static routes are set up a priori between all node pairs to isolate the impact of route flux (considered in §5.3). The static routes were obtained by running OLSR with the default ETX metric until the routing topology stabilized at the beginning of the experiment. Among the 100 randomly chosen flows, 30% are two-hop, 30% are three-hop, 10% are four-hop, 20% are five-hop, and the remaining 10% are seven-hop flows. We compare the multi-hop goodput of Hop to TCP, Hop-by-Hop TCP, DTN2.5, and UDP.

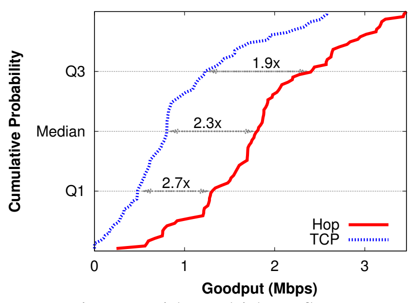


Figure 9: Experiment with multi-hop flows. Hop improves lower quartile goodput by 2.7×, median goodput by 2.3×, and mean goodput by 2×.

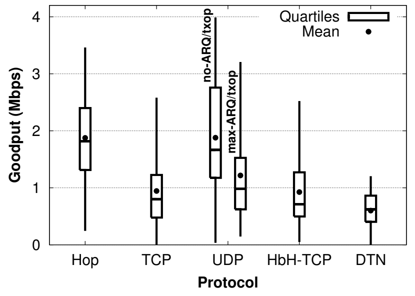


Figure 10: Boxplot of multi-hop single-flow benchmarks. Hop has 2-3× median, and 2-4× mean improvements over other reliable transport protocols. Hop is comparable to UDP/no-ARQ/txop in terms of median/mean — the latter is extremely fast since it has no overhead, but experiences more loss.

Figure 9 shows the CDF of goodput for just Hop and TCP, while Figure 10 shows the summary statistics for all the protocols. Hop consistently outperforms all other protocols. The Q1, Q2, and Q3 gains over TCP are 2.7×, 2.3× and 1.9× respectively. The Q1 gain over TCP is lower than for the single-hop experiment because only good links selected by OLSR are used in this experiment (as evidenced by the better performance of UDP/no-ARQ/txop compared to UDP/max-ARQ/txop). Over lossier paths, Hop's gains are much higher. We also find that the gains also grow with increasing number of hops (refer technical report [15]). For example, the lower quartile gains grow from about 2.7× for two hops to more than 4× for five and six hops.

5.3 Hop under high load

The experiments so far considered one flow in isolation. Next, we evaluate Hop in a heavily loaded network to understand the effect of increased contention and collisions on Hop's performance and fairness. We compare Hop, TCP, and Hop-by-Hop TCP. The experiment consists of thirty concurrent flows that transfer data continually between randomly chosen node pairs in the testbed. All protocols are run over a static mesh topology identical to §5.2. To focus on multihop benefits, we pick src-dst pairs

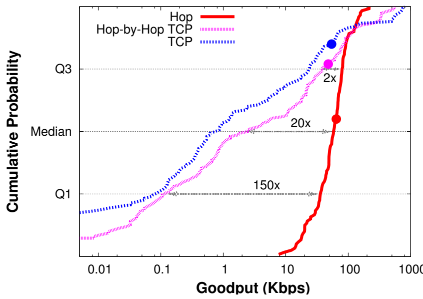


Figure 11: Hop for 30 concurrent flows. Dots on each line shows mean goodput. Median gains of Hop over Hop-by-Hop TCP and regular TCP are huge (20× and 90× respectively) while mean gains are modest (roughly 25% improvement).

that are not immediate neighbors of each other. We run the experiment five times, and for each run, we measure the goodputs of flows half an hour into the experiment, since the network reaches a steady state at this time.

5.3.1 Goodput

Figure 11 shows that Hop achieves a huge improvement in median goodput over TCP and Hop-by-Hop TCP. Hop achieves a median goodput of 54.9 Kbps whereas all the other protocols achieve less than 2.8 Kbps—an improvement of over an order of magnitude! Hop also improves the Q1 goodput by more than two orders of magnitude and upper quartile goodput by 2× over the other protocols. The exact numbers of Hop's median and Q1 gains over other protocols are sensitive to environmental conditions, but we consistently observe them to be large under different conditions. The figure also shows that Hop-by-Hop TCP achieves more than 4× improvement over TCP's median goodput. This shows that end-to-end rate control hurts TCP utilization and using hop-by-hop backpressure with TCP improves its performance. We also run UDP (not shown for clarity), but due to lack of congestion control, around 67% flows get zero goodput (i.e., the median is zero) and the mean goodput is 0.32Kbps.

Hop's mean gain over TCP is just 25%, which is not as impressive as the quartile gains. This is to be expected as TCP is highly unfair and starves a large number of flows to acquire the channel for only a few flows. In many cases, the top three TCP flows get around 90% of the total goodput. In contrast, Hop is significantly fairer and has higher throughput than most of the TCP flows.

	Fairness index
Hop	0.78 (0.09)
TCP	0.12 (0.04)
Hop-by-Hop TCP	0.21 (0.05)

Table 2: Fairness indexes for the 30 flow experiment. Parentheses show 95% confidence intervals.

5.3.2 Fairness

Table 2 shows the fairness index for different protocols. The fairness metric that we use is hop-weighted Jain's fairness index (JFI [28]). When there are n flows, with throughput x_1 through x_n and hop lengths h_1 through h_n, it is computed as follows: JFI $= \frac{(\sum_{i=1}^{n} x_i \cdot h_i)^2}{n \sum_{i=1}^{n} (x_i \cdot h_i)^2}$.

Hop is significantly fairer than both TCP-based protocols. It is noteworthy that while TCP sacrifices fairness for goodput, Hop is superior on both metrics.

5.4 Hop performance breakdown

How much do components of Hop individually contribute to its overall performance? To answer this question, we compare four versions of Hop: 1) the basic Hop protocol that only uses hop-by-hop block transfer, 2) Hop with ack withholding turned on, 3) Hop with backpressure turned on, and 4) Hop with both ack withholding and backpressure turned on. Since the impact of these mechanisms depends on the load in the network, we consider 10, 20 and 30 concurrent flows between randomly picked sender-receiver node pairs. A static mesh topology identical to §5.2 was used. The length of the randomly picked paths are between three and seven hops. The average path length is 3.9 hops in the 10 flow case, 4 hops in the 20 flow case, and 3.9 hops in the 30 flow case. Each flow transmits a large amount of data, and we take a snapshot of the measurements after half an hour.

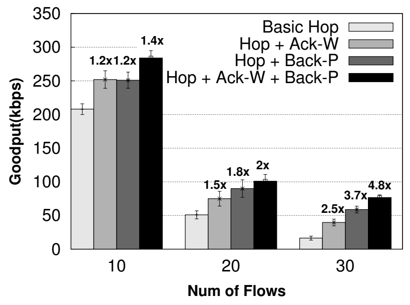


Figure 12: Hop performance breakdown showing contribution of ack withholding and backpressure. Ack withholding and backpressure improve Hop's performance by more than 4.8x under high load.

Figure 12 shows the performance of the different schemes. The benefit of ack withholding and backpressure increases with network load. In the 10 flow case, both ack withholding and backpressure increase goodput by around 20%. With greater network load, congestion increases dramatically, hence the gains due to backpressure is more than due to ack withholding. For example, in the 30 flow case, Hop with backpressure yields 3.7× improvement over basic Hop, whereas Hop with ack withholding yields 2.5× improvement. Furthermore, the benefits of using both backpressure and ack withholding are considerably more than using either one of them.

For instance, the full-fledged Hop yields 4.8× improvement over basic Hop for the 30 flow case.

5.5 Hop with WLAN access points

Next, we evaluate how ack withholding in Hop compares to the 802.11 RTS/CTS mechanism for dealing with hidden terminals. We emulate a typical one-hop WiFi network where a number of terminals connect to a single access point. We setup a 7-to-1 topology for this experiment, by selecting a node in the center of our testbed to act as the "AP node", and transmitting data to this node from all its seven neighbors. Among the seven transmitters, six pairs were hidden terminals (i.e. they could not reach each other but could reach the AP). We verified this by checking to see if they could transmit simultaneously without degradation of throughput. In each run, the nodes transmit data continually, and we measure goodput after 30 minutes when the flow rates have stabilized.

	Mean	Median	Fairness
Hop	663 (24)	652 (33)	0.93 (0.01)
TCP	587 (88)	244 (142)	0.35 (0.06)
TCP + RTS/CTS	463 (20)	333 (87)	0.4 (0.05)

Table 3: Mean/median goodput and Fairness for a many-to-one "AP" setting. 95% confidence intervals shown in parenthesis

We compare Hop against TCP both with and without 802.11 RTS/CTS enabled. The results are presented in Table 3, and show that Hop beats TCP with or without RTS/CTS both in throughput and fairness. While the mean gains over TCP without RTS/CTS are only 12%, the median improvement is about 2.7×. TCP has a crafty way of maintaining high aggregate goodput amidst hidden terminals by squelching all but one of the flows and in effect serializing them. In contrast, Hop achieves almost perfectly fair allocation across the different flows. The addition of RTS/CTS to TCP hurts aggregate throughput but improves median throughput and fairness. However, Hop achieves 1.4× the aggregate throughput, 1.96× the median throughput, in addition to hugely improving fairness over TCP with RTS/CTS.

5.6 Hop delay for small file transfers

How does Hop impact the delay incurred by micro-blocks (small files)? Recall that Hop uses two mechanisms to speed micro-block transfers: 1) It piggybacks micro-blocks less than 16KB in size with the initial BSYN to reduce connection setup overhead, 2) It's ack withholding mechanism prioritizes micro-blocks.

5.6.1 Single-hop transfer delay for small files

First, we evaluate the benefits of Hop's size-aware ack withholding policy. To evaluate this, we pick a one-hop Wifi network where five nodes are connected to an AP (similar setup as our WLAN experiments). In each experiment, one of the five nodes (randomly chosen), transmits a micro-block to the AP at a random time, whereas the other four nodes continually transfer large amounts of data. Each experiment runs until the micro-block completes, at which point we compute the delay for the transfer. We compare against TCP with and without RTS/CTS, and report aggregate numbers over five runs. Figure 13 shows that the transfer delay of the micro-block with Hop is always lower than for TCP (with or without RTS/CTS). In many cases, the delay gains are significant, e.g., for file sizes less than 16KB, the gains range from 3× to 15×. This experiment shows that Hop can be used for delay-sensitive transfers like web transfers, ssh, and SMS in many-to-one AP settings.

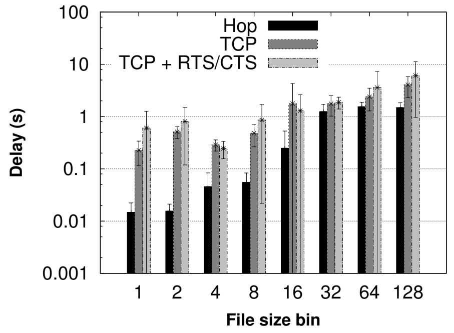


Figure 13: Hop for WLAN: Hop improves delay for all file sizes with improvements between 3-15×

5.6.2 Multi-hop transfer delay for Web file sizes

Next, we evaluate Hop and TCP over a larger workload that comprises predominantly of micro-blocks. (We do not consider TCP with RTS/CTS enabled, since it consistently introduces more delay.) In particular, we consider a Web traffic pattern where most files are small web pages [5]. The flow sizes used in this experiment were obtained from a HTTP proxy server trace obtained from the IRCache project [12]. The CDF obtained was sampled to obtain the representative flow sizes used in this experiment. The distribution of file sizes is as follows: roughly 63% of the files are less than 10KB, 25% are between 10KB-100KB, and remaining are greater than 100KB. To stress multi-hop performance, the sender and receiver for each flow are chosen randomly among the node-pairs that were multiple hops away in our mesh network. Flows followed a Poisson arrival pattern with $\lambda = 2$ flows per second. We present results from 100 flows aggregated in bins of size $[2^{n-1}, 2^n]$ except the bins at the edge, *i.e.* $\leq$2KB, and $\geq$512.

Figure 14 shows that Hop has less or comparable delay to TCP for almost all file sizes except those between 16K-32K. This dip occurs because 16KB is our threshold for piggybacking data with BSYNs. This suggests that a slightly larger threshold might be more effective, but we leave the optimization for future work. For other

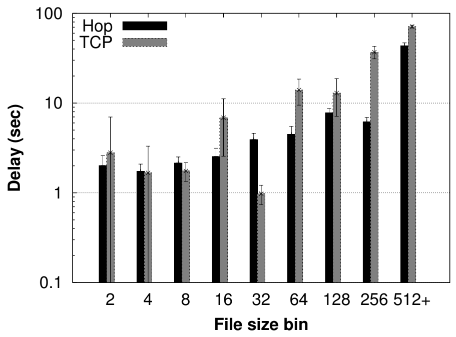


Figure 14: Performance for web traffic: Except the 32KB bin, Hop has comparable or better delay, with gains upto 6×

bins, delay with Hop is mostly lower than TCP (between 19% higher to 6× lower than TCP), demonstrating its benefits for micro-block transfer. Detailed file size microbenchmarks in isolation (i.e., without concurrent transfers) show a similar behavior (detailed in [15]).

5.7 Robustness to partitions

A key strength of Hop is its ability to operate even under disruptions unlike end-to-end protocols such as TCP. We now evaluate how, in a partitioned scenario, Hop compares to hop-by-hop schemes such as DTN2.5 that are designed primarily for disruption-tolerance. In this experiment, we pick a seven hop path and simulate a partition scenario by bringing down the third node and fifth node in succession along the path for one minute each in an alternating manner. Table 4 shows the goodput obtained by Hop averaged over five runs under two different backpressure settings: 1) backpressure limit (H) is set to 1 and 2) backpressure limit is set to 100. Hop outperforms DTN2.5, a protocol specifically designed for partitioned settings, by 2× when $H = 1$, and 3× when $H = 100$. The results show that Hop achieves excellent throughput under partitioned settings, and a large backpressure limit improves throughput by about 15%. This result is intuitive as having a larger threshold enables maximal use of periods of connectivity between nodes. In contrast to Hop, TCP achieves zero throughput since a contemporaneous end-to-end path is never available.

	Goodput (Kbps)
Hop w/ H=1	320 (29)
Hop w/ H=100	457 (18)
DTN2.	159 (15)

Table 4: Goodput achieved by Hop and DTN2.5 in a partitioned network without an end-to-end path.

5.8 Hop with VoIP

In this experiment, we quantify the impact of Hop and TCP on Voice-over-IP (VoIP) traffic. We use two metrics: 1) the mean opinion score (MoS) to evaluate the quality of a voice call, and 2) the conditional loss probability (CLP) to measure loss burstiness. The MoS value can range from 1-5, where above 4 is considered good, and below 3 is considered bad. The MOS score for a VoIP call is estimated as in [6]. The CLP is calculated as the conditional probability that a packet is lost given that the previous packet was also lost.

The experiment consists of a single VoIP flow and multiple Hop/TCP flows that transmit data continually over randomly picked 3-hop paths in the testbed. We emulate the VoIP flow as a stream of 20 byte packets with data rate at 8 Kbps. We evaluate two cases: one VoIP flow with five Hop/TCP flows, and one VoIP flow with ten Hop/TCP flows.

Table 5 shows that Hop achieves significantly better throughput than TCP (in terms of median/mean) but has more impact on the quality of VoIP calls. This is to be expected as TCP starves most of its flows as evidenced by the abysmal median throughput (1-2 Kbps), and therefore has lower impact on the VoIP flow. In contrast, Hop obtains median throughput of a few hundreds of Kbps, while sacrificing a little VoIP quality. We believe that even this discrepancy can be reduced by exploiting 802.11e to set larger contention window parameters to the background queue (e.g. higher backoff), but have not experimented with this so far.

Load		Goodput (Kbps)	CLP	MOS
5 flows	Hop	Median: 468.5 Mean: 1474 (51)	0.37	4.12
	TCP	Median: 2 Mean: 1372 (14)	0.48	4.19
10 flows	Hop	Median: 184 Mean: 336 (24.8)	0.57	3.92
	TCP	Median: 1.7 Mean: 260 (8.5)	0.31	4.16

Table 5: Impact of Hop and TCP on VoIP flows. Result shows the median/mean goodput, conditional loss probability, and MOS for VoIP with 95% confidence intervals in parentheses.

5.9 Network and link layer dynamics

Our experiments so far were run with static routes and with a fixed wireless bit-rate. Now, we evaluate the impact of dynamic routing using OLSR and auto bit-rate control using the default Madwifi *Sample* algorithm. We run TCP under all four combinations of static/dynamic routes and fixed/auto bit-rate selection. We compare these to Hop with a fixed bit-rate and static/dynamic routes. We are unable to evaluate Hop with auto-rate control as the current implementation of Hop disables link-layer ARQs that auto-rate control requires to estimates link quality. As in §5.3, we consider thirty concurrent long-lived flows between randomly chosen node pairs, and run the experiment five times.

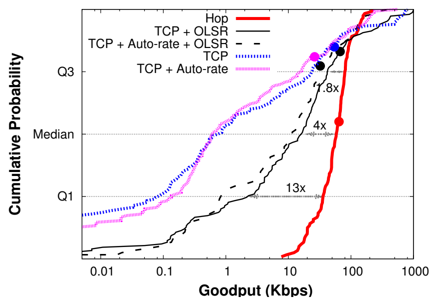


Figure 15: Hop for 30 concurrent flows under dynamic routing and auto bit-rate. Dots on each line shows mean goodput. Median gains by Hop with fixed bit-rate are around 4× over TCP with OLSR and more than 90× over TCP with static routing.

Figure 15 shows that Hop is better than TCP across all combinations, with median gains of 4× over the best of them. (Hop behaves almost identically with dynamic or static routes, therefore we only show the static case in the figure.) Surprisingly, we see that the best combination for TCP is with OLSR and fixed bit-rate. OLSR significantly improves TCP's median goodput or fairness, thereby reducing Hop's gain over TCP in comparison to the static case (§5.3). OLSR benefits TCP as it constantly changes the routing topology with concurrent TCP flows, which makes high goodput flows backoff and yield transmission opportunities to the previously low goodput flows. While the constant shuffling of flows increases TCP's median goodput, OLSR's impact on TCP's mean goodput is small (25%) because the links in the network are already heavily loaded. Auto-rate control makes almost no improvement to TCP since the testbed remains well-connected at 11 Mbps, and hence OLSR choses good links at this bit-rate.

5.10 Hop under 802.11g

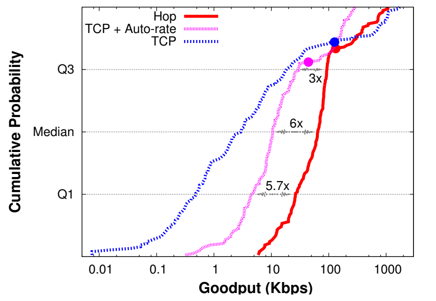


Figure 16: Hop for 30 concurrent flows under 802.11g. Dots on each line shows mean goodput. Hop's median gain is 22× over TCP with bit-rate fixed at 24Mbps, and is 6× over TCP with auto-rate control. Hop's mean gain is 3× over TCP with auto-rate control.

All of our experiments so far were done with 802.11b. How does Hop perform under higher bit-rates obtained using 802.11g? To answer this question, we consider an experiment similar to that in §5.3 with thirty long-lived concurrent flows between randomly chosen node pairs. We use a subset of our testbed (15 nodes) for this experiment as many nodes get disconnected under 802.11g. We ran this experiment with a static routing topology obtained by running OLSR under 802.11g. We consider Hop and TCP with a fixed 802.11g bit-rate of 24 Mbps that yields a reasonably connected topology, as well as TCP with auto-rate control.

Figure 16 shows that Hop improves median goodput by 6× over TCP with auto-rate control and by 22× over TCP with fixed bit-rate. The gains over TCP with auto-rate are lower than in the case of our 802.11b experiments in §5.3 because the maximum bit-rate in 802.11g is higher than the selected fixed bit-rate of 24 Mbps. Thus, TCP with auto-rate control can take advantage of the fact that the maximum bit-rate on 802.11g links is 54 Mbps, whereas Hop's bit-rate is fixed at 24 Mbps. As a result, the highest goodput achieved by a flow that uses TCP with auto-rate control is 23 Mbps, which is higher than Hop's maximum goodput of 16 Mbps. The fact that Hop shows considerable benefits despite using a static best bit-rate suggests that Hop with a good bit-rate selection scheme can benefit even more.

Figure 16 also shows that auto-rate control improves TCP's fairness (median goodput increases by 3.2×) but hurts network utilization (mean goodput decreases by 65%). This is because auto-rate improves the low goodput flows over lossy links by reducing the bit-rate (and thereby the loss rate), but impacts high goodput flows as flows over low bit-rate links are slow and consume a large portion of transmission opportunities.

5.11 Discussion: Hop vs. TCP

Although the above results show Hop's benefits across a wide range of scenarios, our evaluation has some limitations. First, our results are based on a 20-node indoor testbed, so we can not claim that they will hold in other wireless mesh networks. For example, it is conceivable that the benefits due to ack withholding are because of hidden terminals specific to our testbed's topology. Nevertheless, our experience with Hop has been encouraging. Over the last few months, we have experimented with different node placements, static topology configurations, and diurnal as well as seasonal variations in cross traffic and channel conditions, and have seen results consistent with those described in this paper. Second, we have not compared Hop to a large number of proposed TCP modifications for multi-hop wireless networks for which implementations are not available (refer §6.1). We present Hop as a simple and robust alternative to end-to-end rate control schemes, but do not claim that end-to-end rate control can not be fixed to obtain comparable benefits at least in well-connected environments.

TCP's strengths are undeniable. Under high load, it is difficult to outperform TCP significantly in terms of aggregate throughput (refer Figures 11 and 16). TCP backs off aggressively on bad paths reducing contention for flows on good paths resulting in an efficient but unfair allocation. TCP has a similar effect on hidden terminals—by squelching most of the colliding flows, TCP in effect unfairly serializes them but ensures high throughput. Finally, despite its many woes in wireless environments, TCP enjoys the luxury of experience through widespread deployment, setting a high bar for alternate proposals.

Hop is not designed to be TCP-friendly. For example, in the 30 flow scenario, if we convert just 7 of the 30 TCP flows to use Hop instead of TCP, the median goodput of the remaining 23 drops by an order of magnitude [15]. This is unsurprising as Hop's bursty traffic increases the loss and contention perceived by TCP flows causing them to aggressively back off.

6 Related work

Wireless transport, especially the performance and fairness of TCP over 802.11, has seen large body of prior work. Our primary contribution is to draw upon this work and show that reliable per-hop block transfer is a better building block for wireless transport through the design, implementation, and evaluation of Hop.

6.1 Proposed alternatives to TCP

TCP performance: TCP's drawbacks in wireless networks include its inability to disambiguate between congestion and loss [2], and its negative interactions with the CSMA link layer. Proposed solutions include: 1) end-to-end approaches that try to distinguish between the different loss events [25], attempt to estimate the rate to recover quickly after a loss event [19], or reduce TCP congestion window increments to be fractional [21], 2) network-assisted approaches that utilize feedback from intermediate nodes, either for ECN notification [38], failure notification [17] or for rate estimation [32], and 3) link-layer solutions that use a fixed window TCP in conjunction with link-layer techniques such as neighborhood-based Random Early Detection ([9]) or backpressure flow control (RAIN [16]) to prevent losses due to link queues filling up.

TCP fairness: TCP unfairness over 802.11 stems primarily from: 1) excess time spent in TCP slow-start, which is addressed in [32] by use of better rate estimation, and 2) interactions between spatially proximate interfering flows [37, 29] by using neighborhood-based random early detection and rate control techniques.

In comparison to the above schemes, Hop does not rely on end-to-end rate control, and thereby eliminates the complex interaction between TCP and 802.11 that is the root of its performance and fairness problems. Instead, Hop uses simple mechanisms—batching, hop-by-hop backpressure and ack withholding—to improve performance as well as fairness. Hop requires no modifications to the 802.11 MAC protocol.

6.2 Implemented alternatives to TCP

Few implemented alternatives to TCP are available for reliable transport in 802.11 networks today. At the time of writing, we found only two such implementations—TCP Westwood+ and DTN2.5—both of which we compare against Hop. Hop's use of hop-by-hop reliability and backpressure is similar to a recent proposal, CXCC [31], but differs in its use of burst-mode, ack withholding, virtual retransmissions, etc. We could not compare Hop against CXCC as it is not implemented for 802.11.

Two recent systems, WCP [30] and Horizon [27], also address TCP's performance and fairness problems over 802.11. WCP, similar in spirit to NRED [37], augments TCP's end-to-end rate control with network-assisted feedback about contention along the path. WCP shows significant gains in median throughput (or fairness) under load, but often reduces the mean throughput considerably. Horizon uses backpressure scheduling with multi-path routing as a shim between unmodified TCP and 802.11 layers, and shows improved fairness under load in a majority of experimental runs at the cost of mean throughput. In comparison, Hop consistently shows significant improvement in fairness and mild improvement in mean throughput under load. Although we have not performed a head-to-head comparison to Hop, we note that both WCP and Horizon rely on link-layer ARQ per frame that our experiments (Figures 7 and 10) suggest are inefficient for lossy wireless links.

6.3 Other related work

Backpressure: Backpressure was first investigated in ATM [24] and high-speed networks [20] to handle data bursts. A seminal paper by Tassiulas and Ephremides [33] showed that backpressure scheduling can achieve the stable capacity region of a wireless network. This paper sparked off a large body of theoretical work [34] on optimal scheduling, routing, and flow control in wireless networks. However, backpressure scheduling is NP-hard, incurs a high signaling overhead per transmission, and is difficult to implement with the 802.11 MAC layer, so few practical implementations exist.

In recent times, backpressure-like ideas have been adapted for congestion control as an alternative to TCP [31] or underneath TCP [16, 27]; for unreliable hierarchical data aggregation in sensor networks [11]; for reliable bulk transport in linear sensor networks and a single flow [14], etc. In comparison, Hop performs backpressure over blocks to amortize the signaling overhead, uses ack withholding to to alleviate hidden terminal losses, and uses per-hop reliability with virtual retransmissions to efficiently deal with in-network losses.

Batching: Ng et al. [22] show that adapting the burst size of txop's in 802.11e to the load can improve TCP fairness in WLAN settings. WildNet [26] leverages batching with FEC and bulk acknowledgments at the link layer over long-distance unidirectional 802.11 links. Kim et al. [35] aggregate TCP frames using the 802.11n burst mode to amortize the MAC protocol overhead. In comparison, Hop jointly leverages batching both at the link and transport layers.

7 Conclusions

The last decade has seen a huge body of research on TCP's problems over wireless networks, but TCP for good reasons continues to to be the dominant real-world alternative today. One reason may be that TCP is good enough in the common case of wireless LANs, and solutions proposed for more challenged environments do not perform well in the common case. A natural question is if we can have one simple transport protocol that yields robust performance across diverse networks such as WLANs, meshes, MANETs, sensornets, and DTNs. Our work on Hop suggests that this goal is achievable. Hop achieves significant throughput, fairness, and delay gains both in well-connected WLANs and mesh networks as well as disruption-prone networks.

References

[1] `http://standards.ieee.org/getieee802/download/802.11e-2005.pdf`. 802.11e: Quality of Service enhancements to 802.11.

[2] Hari Balakrishnan, Venkata N. Padmanabhan, Srinivasan Seshan, and Randy H. Katz. A comparison of mechanisms for improving tcp performance over wireless links. In *SIGCOMM*, 1996.

[3] A. Balasubramanian, B. Levine, and A. Venkataramani. Dtn routing as a resource allocation problem. *SIGCOMM*, 2007.

[4] John Bicket, Daniel Aguayo, Sanjit Biswas, and Robert Morris. Architecture and evaluation of an unplanned 802.11b mesh network. In *MobiCom*, 2005.

[5] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: evidence and implications. *INFOCOM*, 1999.

[6] R. G. Cole and J. H. Rosenbluth. Voice over ip performance monitoring. *SIGCOMM Comput. Commun. Rev.*, 2001.

[7] M. Demmer and K. Fall. Dtlsr: Delay tolerant routing for developing regions. *NSDR*, 2007.

[8] `http://www.dtnrg.org/`. Delay Tolerant Networking (DTN) Reference Group.

[9] Z. Fu, P. Zerfos, H. Luo, S. Lu, L. Zhang, and M. Gerla. The impact of multihop wireless channel on tcp throughput and loss. In *INFOCOM'03*, 2003.

[10] S. Ha, I. Rhee, and L. Xu. Cubic: a new tcp-friendly high-speed tcp variant. In *SIGOPS*, 2008.

[11] B. Hull, K. Jamieson, and H. Balakrishnan. Mitigating congestion in wireless sensor networks. In *SenSys*, pages 134–147, New York, NY, USA, 2004. ACM Press.

[12] `http://www.ircache.net/`. IRCache: The NLANR Web Caching Project.

[13] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *SIGCOMM*, 2004.

[14] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. Flush: a reliable bulk transport protocol for multihop wireless networks. In *SenSys*, 2007.

[15] M. Li, D. Agrawal, D. Ganesan, and A. Venkataramani. Block-switched networks: A new paradigm for wireless transport. Technical Report TR UM-CS-2008-27, UMass Amherst, 2008.

[16] Chaegwon Lim, Haiyun Luo, and Chong-Ho Choi. RAIN: A reliable wireless network architecture. In *Proceedings of IEEE ICNP*, 2006.

[17] S. Liu, J.; Singh. ATCP: Tcp for mobile ad hoc networks. *IEEE JSAC*, 2001.

[18] `http://www.madwifi.org/`. Madwifi Device Driver.

[19] S. Mascolo, C. Casetti, M. Gerla, M. Y. Sanadidi, and R. Wang. TCP westwood: Bandwidth estimation for enhanced transport over wireless links. In *Mobicom*, 2001.

[20] Partho P. Mishra and Hemant Kanakia. A hop by hop rate-based congestion control scheme. *SIGCOMM*, 1992.

[21] Kitae Nahm, Ahmed Helmy, and C.-C. Jay Kuo. Tcp over multihop 802.11 networks: issues and performance enhancement. In *MobiHoc*, 2005.

[22] Anthony C. H. Ng, David Malone, and Douglas J. Leith. Experimental evaluation of tcp performance and fairness in an 802.11e test-bed. In *E-WIND*, 2005.

[23] `http://www.olsr.org/`. Optimized Link State Routing Protocol.

[24] Cüneyt Özveren, Robert Simcoe, and George Varghese. Reliable and efficient hop-by-hop flow control. *SIGCOMM*, 1994.

[25] Sinha P., T. Nandagopal, N. Venkitaraman, R. Sivakumar, and V. Bharghavan. Wtcp: a reliable transport protocol for wireless wide-area networks. *Wireless Networks*, 2002.

[26] R. Patra, S. Nedevschi, S. Surana, A. Sheth, L. Subramanian, and E. Brewer. WiLDNet: Design and Implementation of High Performance WiFi-based Long Distance Networks. In *NSDI*, 2007.

[27] B. Radunovic, C. Gkantsidis, D. Gunawardena, and P. Key. Horizon: Balancing tcp over multiple paths in wireless mesh network. In *MobiCom*, 2008.

[28] Gojko Babic Raj Jain, Arjan Durresi. Throughput fairness index: An explanation, atm forum/99-0045, february 1999.

[29] S. Rangwala, R. Gummadi, R. Govindan, and K. Psounis. Interference-aware fair rate control in wireless sensor networks. *SIGCOMM*, 2006.

[30] S. Rangwala, A. Jindal, K. Jang, K. Psounis, and R. Govindan. Understanding congestion control in multi-hop wireless mesh networks. *Mobicom*, 2008.

[31] B. Scheuermann, C. Lochert, and M. Mauve. Implicit hop-by-hop congestion control in wireless multihop networks. *Ad Hoc Netw.*, 2008.

[32] K. Sundaresan, V. Anantharaman, H. Hsieh, and R. Sivakumar. Atp: A reliable transport protocol for ad-hoc networks. In *In Proceedings of MOBIHOC 2003*, 2003.

[33] L. Tassiulas, and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *IEEE Transactions on Automatic Control*, 37(12), 1992.

[34] L. Georgiadis, M. Neely, and L. Tassiulas. Resource allocation and cross-layer control in wireless networks. In *Foundations and Trends in Networking,* 1(1):1-144, 2006.

[35] W. Kim, H. Wright and S. Nettles Improving the Performance of Multi-hop Wireless Networks using Frame Aggregation and Broadcast for TCP ACKs In *CoNext*, 2008

[36] M. Vutukuru, K. Jamieson, and H. Balakrishnan. Harnessing exposed terminals in wireless networks. In *NSDI*, San Francisco, USA, April 2008.

[37] K. Xu, M. Gerla, L. Qi, and Y. Shu. Enhancing tcp fairness in ad hoc wireless networks using neighborhood red. In *MobiCom*, 2003.

[38] X. Yu. Improving tcp performance over mobile ad hoc networks by exploiting cross-layer information awareness. In *MobiCom*, 2004.

[39] J. Kurose, K. Ross Computer networking: a top down approach. Addison Wesley, 2007

NetReview: Detecting when interdomain routing goes wrong

Andreas Haeberlen[†⋆] Ioannis Avramopoulos[◇] Jennifer Rexford[‡] Peter Druschel[†]

[†] *Max Planck Institute for Software Systems (MPI-SWS)* [‡] *Princeton University*
[⋆] *Rice University* [◇] *Deutsche Telekom Laboratories*

Abstract

Despite many attempts to fix it, the Internet's interdomain routing system remains vulnerable to configuration errors, buggy software, flaky equipment, protocol oscillation, and intentional attacks. Unlike most existing solutions that prevent specific routing problems, our approach is to detect problems automatically and to identify the offending party. Fault detection is effective for a larger class of faults than fault prevention and is easier to deploy incrementally.

To show that fault detection is useful and practical, we present NetReview, a fault detection system for the Border Gateway Protocol (BGP). NetReview records BGP routing messages in a tamper-evident log, and it enables ISPs to check each other's logs against a high-level description of the expected behavior, such as a peering agreement or a set of best practices. At the same time, NetReview respects the ISPs' privacy and allows them to protect sensitive information. We have implemented and evaluated a prototype of NetReview; our results show that NetReview catches common Internet routing problems, and that its resource requirements are modest.

1 Introduction

Global Internet connectivity is the result of a competitive cooperation of tens of thousands of Autonomous Systems (ASes) using the Border Gateway Protocol (BGP). Unfortunately, interdomain routing is plagued with many serious problems: BGP is hard to manage, and BGP misconfigurations and software bugs can create severe network disruptions [8, 24, 37]. Equipment failures in one AS can cause route flapping and trigger excessive routing announcements in ASes many hops away [35]. The inadvertent configuration of conflicting routing policies in a collection of ASes can lead to persistent oscillation [14]. An adversary that controls a BGP-speaking router can intentionally 'hijack' another AS's address block in order to discard the data packets, snoop on the traffic, impersonate the legitimate destination, or send spam [25, 27].

Many (but not all) of these problems are rooted in the absence of a mechanism to verify routing information. BGP essentially allows anyone to announce any route, whether that route actually exists or not. Hence, there has been a lot of work on securing BGP. However, most of this work focuses on *fault prevention*, that is, masking routing problems by suppressing invalid route announcements. This approach is effective against many common problems, but it cannot prevent other, equally common faults; for example, an ISP might fail to announce a route because of an incorrect export filter. Existing security extensions to BGP, such as S-BGP [22] and soBGP [34], are not effective against such faults. Moreover, existing fault prevention systems require significant buy-in before they can yield much benefit, and they require an Internet-wide public-key infrastructure (PKI); for these and other reasons, prevention systems have not yet achieved widespread deployment.

In this paper, we take a different and complementary approach, namely *fault detection*. If we cannot prevent every routing problem, why not at least ensure that each problem is detected and linked to the ISP that caused it? Fault detection is easy to deploy incrementally: it does not require a central PKI or cryptography on the critical path, and it yields benefits even when the deployment consists of just a few ISPs (or even a single ISP). Moreover, if we accept the possibility of some delay between the occurrence of a fault and its detection, we can catch a very general class of faults, including router and link failures, software bugs, misconfigurations, policy violations, and even attacks by hackers or spammers. In particular, we can detect faults that would be difficult or impossible to prevent, e.g., when a faulty or misconfigured router fails to propagate certain routes.

Fault detection has two main benefits. The first (and most obvious) benefit is that ISPs are automatically informed about routing problems and their causes, which enables them to respond quickly. Thus, ISPs no longer have to rely on monitoring heuristics or customer complaints to find out about problems, which increases customer satisfaction and enables ISPs to swiftly respond even to minor problems. Also, since detection links faults to their causes, ISPs no longer need to diagnose faults manually. Finally, ISPs obtain a 'safety net' that enables them to respond to unexpected problems.

The second, more indirect benefit of fault detection is that it makes an ISP's reliability transparent. Today, ISPs may have little to gain from pushing reliability beyond a certain point, since customers cannot easily attribute a given routing problem to a particular ISP. Fault detection is an opportunity for reliable ISPs to showcase their good performance and to distinguish themselves from the competition, which could help them attract new customers. In the long term, this could even result in a market for reliability, in which customers could directly compare the routing performance of potential providers.

At first, fault detection may appear to be a simple matter of keeping logs of all routing messages and inspecting them (perhaps even manually) for routing problems. However, the problem is complicated by several unique aspects of the interdomain routing system. First, detecting certain types of faults requires that ISPs share information, because the fault cannot be detected based on one ISP's view of the network alone. However, ISPs wish to minimize the amount of information they release to their competitors. Thus, a detection system must balance its detection power against the scope of the information ISPs need to release. Second, the amount of log data collected is so vast that manual inspection is out of the question, except in the most egregious cases. Third, the logs may be incomplete or even incorrect, not least because the routing system is often attacked by hackers who may try to manipulate records in order to cover their tracks. Finally, if the information about faults is to be used as a measure of reliability, we must avoid both false positives and false negatives, which rules out heuristic solutions.

To demonstrate that fault detection is viable, we present NetReview, a system that implements fault detection for BGP. NetReview reliably and automatically detects routing problems by checking secure traces of BGP messages against high-level specifications of the expected routing behavior. NetReview respects the ISPs' privacy and provides strong guarantees: it does not produce false positives or false negatives even when under attack by a Byzantine adversary. Using a prototype implementation of NetReview, we show that its resource requirements are modest, and that it is effective against common Internet routing problems.

Existing work on securing interdomain routing has proven difficult to deploy. A natural question is whether a fault detection system would be hampered by similar problems. To address this concern, we show that NetReview can overcome common deployment hurdles: it can work with existing router hardware, it does not require a global PKI, it can be deployed incrementally, and it offers immediate benefits to early adopters.

The rest of this paper is structured as follows. In Section 2, we begin by giving some background on BGP, and we discuss the specific challenges of BGP fault detection. In Sections 3 and 4, we present the design of NetReview and its specification language. In Section 5, we report results from a feasibility study to show that fault detection is practical. In Section 6, we present solutions to various deployment-related problems, such as operation in a partial deployment or without a CA, and we point out incentives for adoption by ISPs. In Section 7, we describe some advanced features that could be added to NetReview. Section 8 discusses related work, and Section 9 concludes this paper.

2 Background

2.1 Interdomain routing with BGP

The Internet consists of independent administrative entities called *autonomous systems (ASes)*. An AS usually corresponds to a network run by an Internet Service Provider (ISP), although some large ISPs have multiple ASes. Each AS is assigned a unique *AS number (ASN)*; in 2008, about 40,000 ASNs were in active use. In addition, each AS owns a set of IP addresses, which it can assign to its hosts and routers. Usually, ASes use large contiguous sets of addresses that share a common *prefix*; for example, the prefix `128.42.0.0/16` covers all IP addresses whose first two octets are 128 and 42.

To exchange routing information with each other, all ASes use the *Border Gateway Protocol* [28]. Each AS designates some of its routers as *BGP speakers*, which are then connected to BGP speakers in adjacent ASes. When a BGP speaker learns of a route to a new prefix, it can *announce* that route to its peers in adjacent ASes; if the route becomes unavailable later, it must *withdraw* the announcement. BGP is a path-vector protocol, that is, each announcement contains the sequence of ASes that the route traverses in an attribute called `AS_PATH`.

BGP specifies a mechanism for exchanging routing information. Which routes to use and whether or not to announce them to peers is decided independently by each AS according to its own *policy*; for example, an AS might prefer short routes to reduce latency. Some aspects of the policy are determined by an AS's business relationships; for example, an AS might agree to act as the *provider* for another AS, and it would then be expected to offer its customer a route to every prefix it can reach. Adjacent ASes usually sign a *peering agreement*, which specifies the obligations of each peer.

2.2 What is a BGP fault?

The specification of BGP in RFC 4271 [28] describes a message format and a few basic rules; everything else is left to the implementation and the policy of an AS. Therefore, we use a very generic definition of a BGP fault. Suppose we have a complete message trace M_a of all BGP messages a given AS a has sent or received over time (both internally and to/from its peers). Then we simply assume that there is a deterministic function $F_a(M_a, t)$, and we say that AS a is *faulty* at time t if and only if $F_a(M_a, t)$ = true, otherwise we say that AS a is *correct* at time t.[1] Note that F_a is specific to AS a; a different AS b could have a different function F_b.

How can such a function F_a be defined? There are several sources of information that can be used for this purpose (of course, multiple sources can be combined):

- **RFC 4271:** The AS is faulty if it violates the BGP specification, e.g., by sending a malformed message, or by announcing a path that contains a loop.
- **ASN and prefix assignment:** The AS is faulty if it uses a foreign AS number, or if it announces a prefix it does not own.
- **BGP best practices:** The AS is faulty if it does not follow current best practices, e.g., by failing to aggregate prefixes correctly.
- **Peering agreements:** The AS is faulty if it does not honor the peering agreements it has negotiated with its peers, e.g., by failing to export its customers' routes, or by choosing a route through an AS it has promised to avoid.
- **Connectivity:** The AS is faulty if it fails to offer routes to certain prefixes, e.g., because an internal link or equipment failure has caused a partition.
- **Internal goals:** The AS is faulty if its routers fail to achieve some goal the AS has set for itself, e.g., by choosing an expensive route over a cheaper one due to a configuration error.

Note that our definition does not say who defines F_a and who evaluates it; we will address these challenges in Section 3, and we will show which information needs to be shared to ensure that faults are detected. Also, our definition does *not* imply that there is a unique correct message trace for each AS. For example, if an AS is offered multiple routes to a given prefix and its policy does not prefer any route in particular, it can choose any route.

According to our definition, each fault is local to a single AS. Thus, if a faulty AS a exports a bad route to a neighbor b, b does *not* become faulty for propagating the route – except if propagating the route constitutes a fault according to its own function F_b. A special case occurs when a link between two neighboring ASes fails. Since the link is shared by two ASes, we cannot attribute this event to an individual AS, so we attribute it to the pair of ASes instead.

[1]A similar definition can be used for router-level faults. We focus on AS-level faults because they are more general.

2.3 Challenges in BGP fault detection

To illustrate the challenges in building a practical BGP fault detection system, we first consider a simple strawman implementation of fault detection that works as follows. Every ISP enables full logging on all their routers and periodically uploads the logs to a central server, together with a description of their peering agreements and internal goals. Because the central server has full information, it can reconstruct the message trace M_a for each AS a, and it can evaluate F_a for any (past) point in time. This solves the fault detection problem because the central server can eventually detect *any* BGP fault, no matter how complex it is.

However, there are several reasons why this strawman solution would not work in practice:

- **Privacy:** The strawman's logs contain sensitive information that ISPs would not agree to reveal to a third party, such as their routing policy and internal topology. A practical system must protect the ISPs' business secrets while retaining its detection power.
- **Reliability:** The information in the strawman's logs is not necessarily accurate: routers can malfunction, and hackers can tamper with the logs to conceal an attack. A practical system must ensure that no faults go undetected, even when it is under attack.
- **Automation:** Collecting and processing the vast amounts of trace data could prove expensive. A practical system must be able to efficiently check this data without manual intervention.
- **Decentralization:** It is unlikely that ISPs around the world would accept and trust a single fault detector entity. A practical system must not introduce any new trusted entities or require ISPs to coordinate with ISPs they do not already cooperate with.
- **Deployability:** The strawman assumes global deployment. A practical system must have a clear deployment path, with immediate benefits for early adopters and a migration path for legacy equipment.

A fault detection system for BGP should address these five challenges.

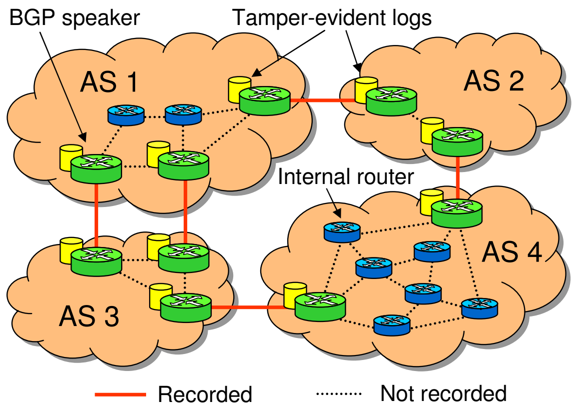


Figure 1: System model. Each BGP speaker maintains a tamper-evident log of the BGP messages it exchanges with other ASes. Internal routing messages are not recorded.

3 NetReview

To demonstrate that the above challenges can be addressed in a practical system, we now present a detection system called *NetReview*. For clarity of presentation, we initially assume that NetReview is deployed universally, and that the allocation of ASNs and IP prefixes to ASes is certified by a trusted certification authority (CA). In Section 6, we describe solutions for partial and incremental deployment, and we show how NetReview can be used without a CA.

3.1 Overview

At a high level, each BGP speaker maintains a log of all the BGP messages it sends and receives (Figure 1). In addition, each AS states a set of *rules* that describe best practices, routing policies, etc. that the AS adopts (the union of these rules specify F_a and thus define what constitutes a fault; they do not necessarily describe the entire routing policy of the AS). Both the logs and the rules are then made available to certain other ASes, who can *audit* them to check whether the rules have been followed. If a rule was broken, NetReview guarantees that at least one auditor can detect this and obtain *verifiable evidence* of the fault, which it can then use to convince third parties.

NetReview only records BGP messages that are exchanged with other ASes, but no internal routing messages. Thus, the log only contains information that an AS would reveal to other ASes anyway; the ISP's proprietary information, such as its internal topology, is not revealed. In addition, each ISP is free to decide which rules it wants to reveal to each auditor. For example, an ISP might choose to reveal its best-practice rules to everyone, and, in addition, it might reveal to each of its business partners a set of rules that describes its policy towards that partner. This is safe because the partner already knows that aspect of the policy from the peering agreement.

NetReview uses cryptographic authenticators [17] to detect if routing messages are not logged correctly. The log itself is tamper-evident, that is, it can detect if log entries are modified after the fact. Thus, NetReview can guarantee that log corruption – due to software bugs or hardware malfunctions – cannot cause faults to go undetected. This guarantee holds even in the presence of Byzantine faults, e.g., when hackers or spammers attempt to cover up the traces of an attack.

NetReview includes a simple specification language for writing rules. The resulting rules can be checked efficiently; we show that a commodity workstation is sufficient to audit several ASes in real time.

NetReview is designed to leverage existing trust and business relationships between *neighboring* ASes. We consider two ASes to be neighbors if they are connected by a direct link.

3.2 Assumptions and guarantees

NetReview's design relies on the following assumptions:

1. **Each AS has at least one diligent neighbor.** By diligent, we mean that this neighbor regularly audits the AS and collects evidence. This is a reasonable assumption because ASes have a natural interest in learning about routing problems of their neighbors.

2. **Each AS is willing to publish a list of its neighbors.** Knowing the nature of the business relationships is not necessary, just the fact that two ASes are connected. This is a reasonable assumption, because the information can already be determined using tools like traceroute or RouteViews [30].

3. **Each AS can eventually send control messages to any other AS.** This property holds for the Internet because the AS graph is connected, and because link failures are repaired in a timely fashion (that is, within at most a few days).

4. **No attacker can invert the hash function or break cryptographic keys.** This is a common assumption for protocols that rely on cryptography.

Note that NetReview is not subject to the limitations for Byzantine fault tolerance techniques, such as the need for $3f + 1$ replicas to tolerate f faults. Fault detection is an easier problem, so this bound does not apply.

NetReview focuses on detecting *observable* faults, that is, faults that a) causally affect at least one non-faulty AS [16], and b) violate a rule that is revealed to at least one diligent AS. This restriction is inevitable because we cannot expect faulty routers to help with fault detection. An example of an unobservable fault would be two faulty routers sending bad routing updates to each other, but neither of them logging the messages or forwarding the authenticators to the other's neighbors. Such a fault cannot be detected as long as it does not affect a correct AS.

Under the above assumptions, NetReview guarantees that a) any observable fault is eventually detected and irrefutably linked to a faulty AS, and that b) no verifiable evidence is ever generated against a non-faulty AS.

3.3 Maintaining tamper-evident logs

In NetReview, each border router maintains a log of all routing messages it has sent to, or received from, a router in another AS. In addition, the logs contain periodic checkpoints of the BGP routing tables, as well as a hash of each rule the AS has adopted. This additional information is needed for auditing and will be discussed in Sections 3.8 and 3.9, respectively.

NetReview's logs are based on the logs in PeerReview [17]. The logs are *tamper-evident*, that is, a router either records precisely the messages it has exchanged with other routers, or it is possible to detect that the router is faulty. Note that, since our goal is fault detection, we do not need to prevent faulty routers from tampering with their logs – being able to detect tampering is sufficient because it is clear evidence of a fault. Specifically, NetReview detects if a router (i) records a message it did not send or receive, (ii) omits a message it did send or receive, (iii) changes an existing log entry, or (iv) keeps multiple logs or a branched log. For lack of space, we only sketch the most important aspects of the log here. Please refer to [17] for a complete description.

Operation: Each log is structured as a hash chain, i.e., every entry e_i is associated with a sequence number s_i and a hash value h_i that covers the entry itself and, transitively, all the previous entries. To explain the protocol for logging message exchanges, we use the example of two routers, Alice and Bob. Whenever Alice sends a message m to Bob, Alice first appends a `SEND(m)` entry to her log and then attaches an *authenticator* to m, which is a signed statement that Alice has logged the transmission of m. The authenticator $\alpha_i = \sigma_{Alice}(s_i, h_i)$ for an entry e_i includes the entry's value in the hash chain h_i and is signed with Alice's cryptographic key σ_{Alice}. The authenticator has two purposes: first, it convinces Bob, and any auditors of Bob's log, that the message is authentic, which rules out (i). Second, it serves as evidence that a `SEND(m)` entry must appear in Alice's log, which addresses case (ii) and, because of the hash chain, case (iii). When the message m arrives, Bob appends a `RECV(m)` entry to his log and then returns an acknowledgment to Alice, which includes an authenticator for the `RECV(m)` entry. At this point, both Alice and Bob have obtained evidence that the other side has properly recorded the message in their log.

NetReview imposes a limit on the number of unacknowledged messages that can be in flight between Alice and Bob at any given time. If this limit is reached, e.g., during an unplanned physical link failure or because Bob refuses to send acknowledgments, the operators are notified and must resolve the problem by leveraging their existing business relationship.

What if Alice or Bob log the message at first but modify or remove it later? When Bob receives the authenticator from Alice, he detaches it from the message (to save bandwidth) and forwards it to Alice's neighbors. Thus, Alice's neighbors eventually learn of all log entries for which Alice issued authenticators. Each neighbor periodically inspects Alice's log to check whether these entries actually appear. If an authenticator is properly signed but the corresponding entry is missing, then Alice must have tampered with the log, maintained multiple logs or a log with multiple branches, and the authenticator is a signed confession. This addresses (iv).

Protocol support: NetReview extends BGP with support for authenticators and acknowledgments. To limit the crypto overhead during bursts of updates, it also introduces a new composite message that allows multiple updates to be covered by a single authenticator (and thus by a single signature). We call this protocol variant *BGP with acknowledgments*, or BGP-A.

Log truncation: Routers require some storage for keeping the log. This storage does not have to be in the router itself – it could be on a separate blade, or on another computer – but capacity is limited, and log entries cannot be stored indefinitely. Therefore we allow routers to discard entries that are older than some time T_{max}, e.g., one year. Since the log contains periodic snapshots of the routing tables, discarding old entries does not destroy information about long-lived routes.

For routers to agree when T_{max} elapses, clocks must be loosely synchronized, e.g., within a few hours. NetReview enforces this by checking the timestamps on the authenticators. If a router's clock is not set properly, its messages will not be accepted by the adjacent routers.

If a log entry were not audited at least once during its lifetime, some faults could remain undetected. However, the typical audit period can be expected to be much shorter than the lifetime of log entries because ASes are likely to be interested in timely fault detection.

3.4 Auditing

To ensure that no fault goes undetected, the logs of each AS must be inspected regularly. Technically, it is possible to allow each AS to audit any other AS; however, NetReview requires only that each AS audit the logs of its neighbors. Neighbors have a natural incentive to learn about each other's routing problems, and because of their existing business relationships, they are in a good position to take action if a problem is discovered. Also, recall our assumption that each AS has at least one diligent neighbor; this ensures that each log entry is properly inspected at least once.

To inspect an interval $I := [t_1, t_2]$ of a target's log, the auditor proceeds as follows:

1. If the auditor is not a neighbor of the target, it asks the target's neighbors for authenticators from interval I.
2. The auditor asks each of the target's border routers for a set of rules[2] and a signed segment of its log that covers interval I.
3. The auditor checks whether the following properties hold for the set of logs it has obtained:
    - **Consistency:** All authenticators match an entry in one of the logs.
    - **Conformance:** The sequence of messages in each log conforms to BGP-A.
    - **Compliance:** The target has followed each of the rules it has revealed.

3.5 Extracting evidence

When an auditor discovers an interval $I' := [t'_1, t'_2] \subseteq I$ for which one of the above properties does not hold, it extracts the corresponding log segment, starting at the most recent snapshot. Then it removes all entries that are not essential for checking the property (such as additional snapshots), as well as any parts of the first snapshot that are not needed to replay this particular segment. The result is a compact data structure that irrefutably ties the fault to the cryptographic key of the responsible AS, and thus (via the certificate) to its principal. This data can be used as evidence of the fault, and a third party can verify it independently without having to repeat the audit.

Once an auditor has obtained evidence, it notifies the local administrator, who can use the evidence in several ways. For example, if a best-practice rule has been violated, the auditor can choose to make the evidence publicly available; thus, it is possible to evaluate an ISP's performance by asking its neighbors for evidence of faults. If a private rule was broken, the evidence can be used to convince an arbitrator or a judge.

[2]In Section 3.9, we describe how the auditor can verify that the rules are genuine.

3.6 Consistency and conformance checks

The consistency check detects if the target AS has tampered with its log. Recall that each BGP-A message or acknowledgment contains a signed authenticator that is linked to a specific log entry, and thus to a specific point in the hash chain. If the target has returned a valid log segment, it will be consistent with all the authenticators; otherwise the log segment and the mismatched authenticator constitute a proof of misbehavior. Since neighbors collect each other's authenticators, and since we assume that each AS has at least one diligent neighbor, we know that any forged, omitted, or modified log entry is eventually detected by at least one neighbor.

The conformance check detects if the target has deviated from the BGP-A protocol. This is a purely syntactic check that does not consider *which* routes were announced, but rather *how* they were announced. For example, NetReview checks whether each message was well-formed and whether sessions were opened with the proper handshake before announcements were sent.

If the target AS passes the consistency and conformance checks, the auditor is convinced that the log accurately reflects the target's BGP traffic. The remaining check is designed to detect routing problems.

3.7 Extracting the routing state

The previous two checks are performed on logs from individual border routers of an AS. However, many routing problems arise because of inconsistencies between multiple routers. Therefore, the auditor must perform the compliance check based on the 'global' routing state of the AS, which it obtains by merging the logs from the individual routers.

NetReview models the 'global' routing state of an AS as follows. At any given point in time, the AS has a set of *peering points* with neighboring ASes, and for each peering point there are two routing information bases (RIBs): the *outRIB* contains routes that the AS has announced to its neighbor, and the *inRIB* contains routes that the neighbor has announced to the AS. Since BGP does not permit the announcement of multiple alternative routes, each RIB can contain at most one route for each prefix.

To determine how the target's routing state evolved over time, the auditor starts by loading the oldest checkpoint from each log, which contains a snapshot of the RIBs. Then it repeatedly picks the unprocessed message entry with the earliest timestamp across all logs, and it applies the updates in the message to the corresponding pair of RIBs. Thus, it obtains a sequence of routing states $S(t_i)$, where t_i indicates the time of the message that triggered the change. Note that each $S(t_i)$ contains a pair of RIBs for each router or peering point; it is *not* a 'global' RIB for the entire AS.

3.8 Compliance check

The compliance check detects if the target has broken any of its rules. Conceptually, this is done by checking each rule against each of the routing states $S(t_i)$. Recall that even the complete set of rules does not necessarily amount to a full specification of the AS's routing policy; thus, checking rules is not equivalent to re-evaluating each routing decision the AS has made.

We have developed a simple specification language that ASes can use to formulate rules. In this language, a rule is written as a predicate on an individual routing state $S(t_i)$. For example, the rule

$$\forall c \forall r \in \text{outRIB}(18, c): \quad (\text{prefix}(r) \in P) \Rightarrow (123 \in \text{communities}(r))$$

stipulates that, when a route r belongs to a prefix from the set P and is announced to AS 18 over any peering point c, r must be tagged with the community 123. We give more details on the specification language and the rule checker in Section 4.

3.9 Rule commitment and access control

For the compliance check, the auditor must know which rules should hold for the target during the audited interval. Also, if a rule is violated, the auditor should obtain evidence that the rule existed at the time of the fault. The easiest way to accomplish both would be to simply record the rules in the tamper-evident log. However, since the logs are visible to each of the target's neighbors, this might reveal proprietary information about the target's routing policies.

Instead, we only require that ASes *commit* to their rules by logging a hash value $H(s_i, r_i)$ for each rule r_i. s_i is a 128-bit salt, which makes it difficult for an inquisitive auditor to learn sensitive information by checking for well-known rules, or to run a dictionary attack. On the other hand, if an auditor knows r_i and s_i a priori (perhaps from a peering agreement it shares with that AS, or because the AS has revealed them earlier), it can easily check whether the corresponding hash value is present. If not, it can use the log as evidence and file a complaint against the AS for breaking the contract.

Why would an AS commit to any rules at all, and why would it reveal a rule to an auditor? For example, ASes can use NetReview to enforce provisions from their peering contracts. The parties could agree to a set of rules and add them to their respective logs; they would then reveal these rules to each other, but not to anyone else. Or an AS could adopt a set of best-practice rules to highlight its good performance, and reveal these rules to everyone.

4 Writing and checking rules

NetReview includes a simple specification language that ASes can use to formulate rules. In this section, we describe this language in more detail, and we explain how rules in this language are evaluated.

4.1 Language design

The language includes three features we believe to be key for BGP fault detection. First, the language is *declarative* and refers to a high-level property, rather than to a specific algorithm for choosing routes. This makes rules easier to write and debug than, say, router configuration files. Moreover, many properties can be specified as rule templates that only require a few AS-specific parameters. A number of common templates are already included with NetReview.

Second, rules are *partial* specifications of the expected behavior. The above example only describes what should happen to routes that are announced to AS 18 and whose prefix is in P, but it does not say anything about the other routes. Thus, an AS can reveal a rule without revealing its entire routing policy. Also, we can vary the strength and number of rules and thus control how restrictive the checking should be.

Finally, rules are *time-local*, that is, they depend only on a small number of past and future states. This is possible because interdomain routing is essentially memoryless: whether or not a route is exported depends solely on which routes are *currently* available; it is irrelevant whether a route was available earlier, or will become available later.[3] This improves efficiency considerably, since NetReview only needs to remember a small number of routing states at any given time.

4.2 Specifying rules

Each NetReview rule consists of a set of constants and a set of predicates in first-order logic. The predicates are written using boolean operators, existential and universal quantifiers, and equality. They can use two functions called `inRIB(i,j)` and `outRIB(i,j)` to access the RIBs for a peering point j with a neighbor with AS number i. An optional third argument contains an interval operator.

[3] A notable exception is age-based tie breaking. We handle this by including the age of each route in the RIBs.

```
const setof(integer) asns = { 8, 9 };
forall cpref in affectedPrefixes, peer in asns {
  forall p1 in peeringPoints(peer), p2 in peeringPoints(peer) {
    (p1 != p2) => forall route in outRIB(peer, p1, intersect[now-5.0s,now]) {
      (prefix(route) == cpref) => exists route2 in outRIB(peer, p2, union[now-5.0s,now]) {
        (prefix(route2) == prefix(route)) and (sizeof(as_path(route2)) == sizeof(as_path(route)))
} } } };
```

Figure 2: Example rule in the syntax used by the NetReview rule checker. `intersect[a,b]` selects routes that were announced continuously between time `a` and time `b`, while `union[a,b]` selects routes that were announced at any point between time `a` and time `b`. `now` is the point in time for which the rule is evaluated.

Additionally, NetReview's rule checker has several built-in functions and operators for manipulating numbers, routes, sets, and sequences. These include basic arithmetic operators, functions for accessing the individual elements of a route, and set operators such as union, intersection, containment, and indexing. Figure 2 shows an example rule. This rule says that, when exporting a route to AS 8 or 9, the adopting AS must advertise AS_PATHs of the same length over all peering points with that AS.

4.3 Interval operators

Why do rules ever have to depend on future or past states? The reason is that, due to propagation delays and clock skew, RIBs from different routers may be slightly out of sync. Hence, there can be short intervals during which a route appears in an inRIB but in none of the outRIBs, or vice versa. To an auditor, this might look like a transient rule violation.[4]

To avoid false positives in this case, we must introduce a bit of leeway. NetReview's specification language contains two timing-related operators. Both operators take an interval $I = [t - \alpha, t + \beta]$ as an argument, where t is an instant in time and α and β specify how far the interval extends into the past and into the future, respectively. The *union operator* returns all routes that have been advertised at some point in I, and the *intersection operator* returns all routes that have been advertised continuously during I. This allows us to mask transient inconsistencies. For example, we might stipulate that a route may only be exported if a prefix of that route was available within two seconds of the current time, or that a route must be exported to some neighbor if it has been available for at least five seconds. We limit α and β to 60 seconds each; thus, the auditor must remember at most two minutes' worth of past or future states.

If a rule contains interval operators, it can miss actual transient faults that exist for less than $\alpha+\beta$ seconds. The interval needs to be no larger than the maximum propagation delay plus the maximum clock skew among the routers of an AS, so this is not a serious limitation.

[4]The use of a distributed snapshot algorithm such as [6] could avoid this problem, but it would require changes to the ISPs' internal route distribution mechanism.

4.4 Optimizations for checking rules

Conceptually, an auditor must evaluate each predicate whenever a) the routing state of the target AS changes due to an incoming or outgoing BGP-A message, or b) the value of an interval operator changes. For example, if a rule contains two intervals $I_1 = [t - 5, t + 3]$ and $I_2 = [t - 2, t + 3]$, the auditor must also evaluate each predicate five and two seconds *before* and three seconds *after* each routing change.

In practice, we can dramatically reduce the number of predicate evaluations using two simple optimizations. First, since rules typically consider each prefix individually, we can often restrict universal quantifiers to the set of prefixes that are actually affected by a routing change during the current evaluation. This set of prefixes is made available in a special variable called `affectedPrefixes`. Second, we can apply some simple query optimizations. For example, in the rule in Figure 2, NetReview combines the check for `prefix(route)==cpref` with the innermost `forall` quantifier, which reduces the quantifier to a simple projection.

4.5 Discussion

Even though our specification language is very simple, we have found that it is sufficient to describe many of the routing problems that have been reported in the literature, including origin misconfigurations [24], incorrect use of communities [24], incorrect extensions of imported routes [29], route deaggregation, redistribution attacks, and inconsistent path lengths [9]. We note that the particular details of the language are not critical to NetReview; NetReview just needs a way to specify and check constraints on the behavior of an AS. Our language could easily be extended or replaced without affecting the rest of NetReview.

No origin misconfiguration	$\forall a \forall p \forall r \in \text{outRIB}(a, p, \cap[t-40, t]): (\lvert \text{as\_path}(r) \rvert = 1 \wedge \text{prefix}(r) \in \text{ownPrefixes}) \vee (\exists a' \exists p' \exists r' \in \text{inRIB}(a', p', \cup[t-40, t+5]): \text{prefix}(r) = \text{prefix}(r') \wedge \text{startsWith}(r, r') \wedge (\forall n \in r - r': n \in \text{ownPrefixes}))$
Export customer routes	$\forall a \in \text{customers}\, \forall p \forall r \in \text{inRIB}(a, p): ((\forall n \in \text{as\_path}(r): n = a) \Rightarrow \forall a' \in (\text{peers} \cup \text{providers})\, \forall p' \exists r' \in \text{outRIB}(a', p', \cup[t-15, t+15]): \text{prefix}(r) = \text{prefix}(r') \wedge \text{endsWith}(r, r'))$
Honor no-advertise community	$\forall a \forall p \forall r \in \text{inRIB}\,(a, p, \cap[t-5, t]) : \texttt{NO\_ADVERTISE} \in \text{communities}(r) \Rightarrow (\neg \exists a' \exists p' \exists r' \in \text{outRIB}(a', p'): \text{prefix}(r) = \text{prefix}(r') \wedge \text{getElement}(\text{as\_path}(r), 1) = a)$
Consistent path length	$\forall a \in (\text{customers} \cup \text{peers})\, \forall p \forall p': (p = p') \vee (\forall r \in \text{outRIB}\,(a, p, \cap[t-5, t])\, \exists r' \in \text{outRIB}(a, p'): \text{prefix}(r) = \text{prefix}(r') \wedge \lvert \text{as\_path}(r) \rvert = \lvert \text{as\_path}(r') \rvert)$
Backup link	$\forall a \in \text{backups}\, \forall a' \in (\text{customers} \cup \text{peers})\, \forall p\, \forall r \in \text{outRIB}(a', p) : (\lvert \text{as\_path}(r) \rvert > 1 \wedge \text{getElement}(\text{as\_path}(r), 1) = a) \Rightarrow (\neg \exists a'' \in \text{providers}\, \exists p' \exists r \in \text{inRIB}(a'', p', \cap[t-5, t]))$

Table 1: Rules we checked in our experiments. Each rule is explained in Section 5.3. The variables a, a' are for AS numbers, p, p' are for peering points, and r, r' are for routes. inRIB(a, p) and outRIB(a, p) stand for the sets of routes imported and exported, respectively, to AS a over peering point p; they can be combined with an interval operator.[5]

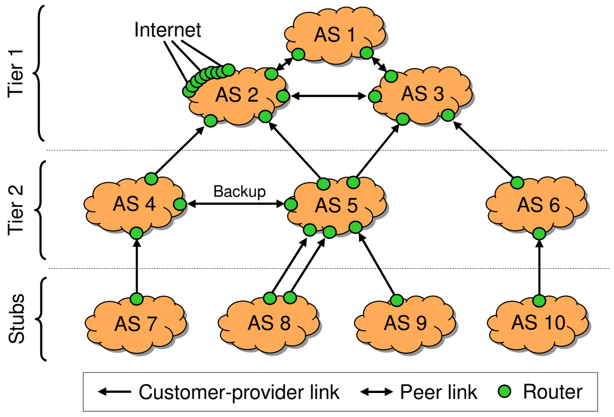


Figure 3: AS topology in our experiments. AS 2 receives updates from an Internet BGP trace.

5 Feasibility study

In this section, our goal is to demonstrate that NetReview (and, more generally, the fault detection approach) is practical. Using a prototype implementation of NetReview, we answer the following high-level questions:

- Are NetReview's rules expressive enough to describe common routing problems?
- How much storage and bandwidth is needed to maintain the tamper-evident logs?
- Is fault detection feasible at Internet scale?

5.1 Methodology

In NetReview, all communication related to a given AS occur among the direct neighbors of that AS. Hence, a small-scale deployment is sufficient to estimate the overhead. However, getting even a small number of contiguous Internet ASes to deploy experimental software would be extremely difficult. Instead, we used software routers to emulate a small AS topology in the lab, but we ensured that the routing table sizes and the amount of BGP traffic closely approximated those of real Internet ASes.

To achieve this, we injected an Internet BGP trace into one of the ASes, including a checkpoint of the initial routing table. From there, the routes were propagated to the other ASes via BGP, creating BGP traffic on each link and populating the other routing tables. This mimicked the conditions that would have occurred if our model topology had been part of the global Internet, so we could get realistic estimates for many performance metrics, e.g., how quickly the logs grow and how much time is required for checking. We found that, since the first trace already contained a route to each available prefix, injecting additional traces would not have increased the routing table sizes.

5.2 Experimental setup

Our NetReview prototype implements the basic system we have described so far, plus the additional techniques described later in Section 6, which enable NetReview to operate without a CA, in a partial deployment and with existing routers. These techniques add some overhead to our results, so the overhead of the basic algorithm would be lower than what we report here.

For our experiments, we set up a synthetic network of 35 Zebra BGP daemons [12], which form a topology of 10 ASes (Figure 3). Our network contains a mix of AS types, ranging from large tier-1 ASes to small stub ASes, as well as both customer/provider and peering relationships. This diversity allowed us to implement and check a variety of different routing policies. Note that AS 8 and AS 5 have two separate peering points, which will become important later.

[5]The interval sizes we use are worst-case values for a mirroring monitor (mainly due to MRAI timers). Much smaller intervals would suffice if the monitor is attached via port replicators or BMP [31].

For each AS, we configured a default routing policy that satisfies the Gao-Rexford conditions [11]. If a route is imported from a customer, it is exported to all neighbors; otherwise (if the route is from a peer or provider), it is exported only to customers. In some of our experiments, we vary this policy by injecting configuration errors or imposing additional constraints. Internally, each AS uses a full-mesh iBGP topology. We did not set up route reflectors because NetReview is oblivious to iBGP.

We injected routing updates from a RouteViews BGP trace [30] into AS 2. We used a 15-minute trace that was collected by a Zebra router at Equinix in Ashburn, VA, on January 27, 2008. The collecting router peers with nine other ASes. The trace contains 15,141 updates from these neighbors, and the corresponding RIB snapshot contains 243,198 unique prefixes. Thus, AS 2 behaved as if it were connected to the Internet in Ashburn, VA, and it exported a realistic set of prefixes to the other ASes.

NetReview's overhead depends in part on the number of neighbors an AS has. Unless otherwise noted, the numbers we report are for AS 5. Since 92% of Internet ASes have degree five or less [3], our results are representative of all but the largest Internet ISPs.

5.3 Rules we checked

In our experiments, we used NetReview to enforce five rules, which are shown in Table 1. In plain English, these rules state the following:

- **No origin misconfiguration:** An AS may only export a route if it owns the corresponding IP prefix, or if the exported route is an extension of another route that the AS is currently importing (motivated by [24]).
- **Export customer routes:** If an AS imports a direct route from one of its customers, it must export that route to its peers and providers.
- **Honor no-advertise community:** An AS must honor the NO_ADVERTISE community; it may not re-export a route that is tagged with this community.
- **Consistent path length:** When exporting a route to a customer or a peer, an AS must advertise AS_PATHs of the same length at all peering points (motivated by [9]).
- **Backup link:** An AS may only export a route via a backup path if its direct links become unavailable.

We chose these five rules because they can be used to detect real problems that have been reported in the Internet [9, 24, 29], and because they demonstrate the different types of conditions NetReview can verify (of course, each rule could be varied and customized in a number of ways). Note that the first two rules are very powerful; together, they can find almost all of the routing problems that were studied in [24]. In particular, the first rule covers AS_PATH manipulations, which are the main focus of secure routing systems like S-BGP (it actually goes beyond S-BGP in that it can also check for timely route withdrawal). The last three rules catch routing problems that would be difficult to find without a detection system, since they can only be detected by combining information from several routers and/or ASes.

5.4 Functionality check

We begin with a simple functionality check to show that the prototype is fully functional and works as expected. Recall that NetReview's design precludes false positives and false negatives if each AS is audited regularly.

We ran a series of six trials. In the first trial, we used the correct configuration for each AS. In the following five trials, we made a configuration change to a NetReview-enabled AS at some point during the experiment that caused one of the five rules to be violated. After each trial, we audited all the logs.

As expected, NetReview did not report any problems during the first trial. In each of the other trials, it reported the fault we had injected. The output also included the time interval in which the fault appeared, as well as the variable assignments (prefixes, AS numbers etc.) for which the corresponding rule did not hold. This is valuable for administrators because it shows not only where the fault occurred (in the audited AS) but also for *which* prefix the exported paths did not have the same length, *which* peering points were affected, etc.

5.5 Processing power

BGP-A speakers and monitors must generate and verify cryptographic signatures. The necessary processing time is a function of the number of messages they send and receive. In our experiment, the monitor in AS 5 sent 1,973 BGP-A messages and received 1,579 during the 15-minute period. Since all messages are acknowledged, this required 3,552 signatures to be generated and an equal number to be validated, on average four signatures and validations per second. On a 3 GHz Pentium 4, a 1024-bit RSA signature can be generated and verified in less than 3.5ms.

Unlike BGP messages, BGP-A messages can contain updates for multiple different routes, which explains why the number of messages is much lower than the number of routing changes in our BGP trace. This also limits the number of validations that are required when updates arrive in bursts. For example, if a router is restarted and re-

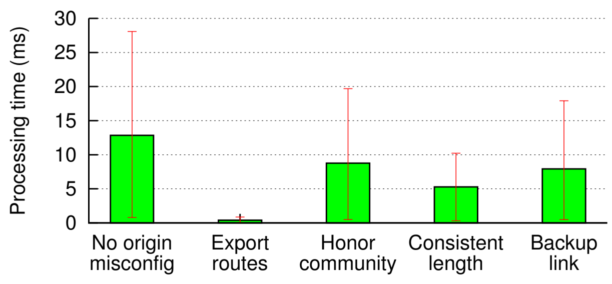


Figure 4: Average processing time required to check a rule over one second of log data (the error bars show the 5th and the 95th percentile). The speed is sufficient for checking multiple ASes in real time.

ceives full routing tables from its neighbors, it only needs to check one signature per routing table. This is in contrast to S-BGP [22], which needs to check a signature for every single route.

Auditors need processing power to extract the routing state from the logs, and to check it against the specified rules. In our experiments, we found that the processing time was dominated by rule checking, which in turn depends on the number of routing changes as well as the complexity of the rules. Our five rules can be evaluated independently for each prefix, so the first optimization from Section 4.4 can be used. It would take more time to check rules that depend on a large number of different prefixes, but we are not aware of any useful rules that have this property.

Figure 4 shows the average time required to check a one-second log segment against each of our five rules.[6] Our 15-minute log required 11,629 such checks, which took 41.5 seconds on a Pentium-4 workstation.

In practice, the checking time would also depend on the number and complexity of the rules the target AS is revealing to the auditor. There is little published information about the policies used by commercial ASes, so we cannot say how large a 'typical' set of rules would be. We already included a generic policy rule (rule #2) in our set, which may be sufficient for small ASes. Even if we assume that a typical set contains 20 rules (four times the size of our set), an AS with five neighbors would still only need a single workstation to perform real-time auditing. If an AS has more neighbors, it can spread the load across multiple machines, since rule checking can be trivially parallelized.

[6]The processing time varies considerably because some one-second intervals contain many updates, while others contain none at all.

5.6 Storage space

BGP-A speakers require storage for checkpoints, the tamper-evident log, and for the certificates that bind each key to the identity of an AS. An X.509 certificate with 1024-bit RSA keys is about 1kB. With web-of-trust signature chains (described in Section 6.1) and a typical AS-path length of four, each certificate is 5kB; thus, a database with certificates for 40,000 ASes would require approximately 195 MB.

The size of a checkpoint is dominated by the RIBs; it depends on the number of prefixes and peering points. One RIB with 244,000 prefixes and a 90-second history takes about 9.0 MB, so, if we conservatively assume that each prefix appears in every inRIB and every outRIB, a complete checkpoint for an AS with six peering points could take up to 108 MB. If the AS records one checkpoint every minute and keeps all checkpoints for one day, plus one checkpoint for each day of the last year, it would require up to 190 GB.

In our experiment, the log grew at a rate of about 332 kB per minute (without checkpoints). Hence, we estimate that one year's worth of log data would take about 166 GB. The log size is also a function of the number of peering points and the frequency of routing changes. Since the log mostly contains routing updates, its growth rate is roughly proportional to the amount of BGP traffic an AS generates. Recall that the numbers we report are for an AS with five neighbors; if an AS has more neighbors (and thus more peering points), its storage requirements are higher. For the largest ASes (UUNet has 2,652 neighbors), on the order of a hundred Terabytes of storage may be necessary to store the log for a year. However, the log would be distributed over thousands of routers.

Auditors require no permanent storage; however, it makes sense for them to cache a recent checkpoint for each AS they are auditing, so they do not have to download one repeatedly.

5.7 Message overhead

BGP-A speakers generate traffic for maintaining BGP-A sessions, for exchanging authenticators and for responding to audits. We look at each type of traffic in turn.

In terms of traffic, BGP sessions and BGP-A sessions are quite similar. If 1024-bit keys are used, a BGP-A message and its acknowledgment have 367 header bytes, while a BGP message only has 16. On the other hand, a BGP-A message can advertise many different routes, while a BGP message can only advertise one. In our experiment, AS 5 generated an average of 132 kB of BGP-A messages and acknowledgments per minute; these were equivalent to 135 kB of BGP messages.

Upon receiving a message or an acknowledgment, a BGP-A speaker detaches the authenticator and forwards it to the sender's neighbors. With 1024-bit keys, the size of an authenticator is 156 bytes; in our experiment, AS 5's neighbors sent 2.1 MB of AS 5's authenticators over the 15-minute period. However, authenticators are also collected from messages read during an audit, so the required traffic is *quadratic* in the number of neighbors: each neighbor audits each message and sends the corresponding authenticator to each of the other neighbors. This can be a problem for large ASes (e.g. UUNet). Therefore, authenticators from large ASes should be sent to only a subset of its neighbors. This does not affect NetReview's guarantees as long as the subsets used by all neighbors intersect in at least one diligent neighbor.

In our experiment, all audits were incremental; the auditor transferred a full checkpoint once and then retrieved only the log entries that were added since the last audit. In the limit, the required traffic is the size of the log times the number of auditors, plus some overhead for headers.

In total, AS 5 caused about 420 kbps of BGP-A traffic, including routing updates, auditing, and authenticators sent by the neighbors. This corresponds to the bandwidth of a typical DSL upstream, which is insignificant compared to the amount of traffic ISPs routinely handle.

5.8 Summary

Our experiments show that NetReview's simple rules are sufficient to describe common, nontrivial routing problems. Also, NetReview's resource requirements are moderate: in a typically-sized AS with five neighbors, routers must sign less than four messages per second on average, a single hard disk is sufficient to keep one year's worth of log data, and the total traffic is less than the capacity of a single broadband upstream link. Finally, we have demonstrated that fault detection is feasible at Internet update rates. By running the NetReview software on just a single workstation, an ISP can audit dozens of neighboring ASes in real time.

6 Practical challenges

In the previous two sections, we have shown that it is feasible to build a fault detection system with strong guarantees, and that its resource requirements are moderate. The goal of this section is to show how NetReview deals with the various practical problems that have hampered the deployment of previous solutions. In particular, we will show that NetReview can operate without a CA, that it can be effective in a partial deployment, that it can initially be deployed without upgrading any routers, and that it offers incentives for incremental deployment.

6.1 NetReview without a CA

Despite many proposals, deploying a global CA for prefixes and ASes has so far not found acceptance [19]. NetReview can use such a CA if it exists, but it does not require it. In the absence of a CA, we need to find replacements for two services that a CA provides: associating each key pair with a real-world identity, and certifying ownership of AS numbers and IP prefixes.

We solve this problem using a web-of-trust approach that is inspired by [33, 34]. Each AS initially generates a key pair and creates a self-signed certificate. Then it sends the certificate to its immediate neighbors, who append their own endorsement and forward it on to their neighbors, etc. The overhead for flooding certificates is not a concern, because the AS topology changes slowly.

Each AS obtains a database of all certificates, each with a chain of endorsements that corresponds to the shortest path between the local AS and the AS represented by a given certificate. Can these certificates be trusted? We can safely assume that each AS knows the true identity of the neighbor attached to each of its physical links. Moreover, we have assumed earlier that each AS has a diligent neighbor. This neighbor can detect if the AS signs a certificate that do not correspond to its true identity, or endorses a certificate that does not come from one of its neighbors. Thus, a node can (transitively) trust every certificate that is endorsed by one of its neighbors.

In addition, we require each AS to log a public pledge that specifies its current ASN and prefix ownerships.[7] ASes extract this pledge during audits and compare it to their database; if there is any change, they flood it to all other ASes. Thus, NetReview can detect if two ASes claim ownership of the same ASN or of overlapping prefixes, and it provides each with evidence of the other's claim. The conflict can then be resolved through existing mechanisms, e.g., by a mediator or a judge.

6.2 Partial deployment

It would be unrealistic to expect that all ASes adopt NetReview, much less that all ASes install the system at the same time. Therefore, NetReview must be able to work in a partial deployment, that is, it must be able to interact with non-participants via BGP.

By default, BGP-A speakers and monitors record only BGP-A messages in their logs, and auditors use only BGP-A messages to reconstruct the routing information. However, legacy neighbors have no components that speak BGP-A. If we simply omitted all routes imported from or exported to these neighbors, the information in the log might not be sufficient to evaluate many interest-

[7]Prefixes used for IP anycast [26] require special handling because they may be owned by multiple ASes simultaneously.

ing conditions. For example, if an AS acts as a provider for another AS, it may be required to export routes for all prefixes it knows about, even if the corresponding route is through a non-participant. Therefore, if an AS has legacy neighbors, its BGP-A speakers and monitors additionally record all the (unsigned) BGP messages they exchange with these neighbors.

Why keep this information in the secure record if a faulty participant AS can simply record whatever it wants? There are three reasons. First, we can isolate non-malicious faults such as misconfigurations or hardware failures, where the faulty AS still records correct information. Second, even if an AS lies about the routes it is importing or exporting via BGP, it must lie *consistently* to avoid detection by the auditors. For example, if the AS claims to have imported a certain route via BGP, it must re-export that route to each participating neighbor if required by its peering agreement, and it cannot export different versions to different neighbors.

Third, logging BGP messages enables an intermediate level of participation in NetReview. If a non-participant AS a is a neighbor of a participant AS b, a can act as an auditor and compare b's log to the BGP messages b actually sent, without fully deploying fault detection itself. All a needs is the NetReview auditor software and a current snapshot of its own BGP tables. If a finds a discrepancy, it can investigate it by contacting the participant AS b. This option could encourage neighbors of participant ASes to 'try out' fault detection.

Partial deployment requires an addition to the web-of-trust technique in Section 6.1. As long as the deployment is contiguous in the AS graph (which is likely if tier-1 ASes join first), the technique works as described. When a second 'island' of participants arises, at least one member of each island must exchange cryptographic credentials out-of-band. These members are then considered NetReview neighbors (even though they do not share a physical link), and they forward certificates from their respective islands to ensure that each AS has a full set. To increase the chance that ASes in small islands have a diligent neighbor, they also collect authenticators for each other and periodically audit each other's log.

6.3 Using existing routers

Requiring ISPs to upgrade or replace their routers to deploy NetReview would present a significant hurdle. Therefore, it is useful to have an intermediate solution that works with existing, unmodified routers. Our solution is to run the NetReview software on ordinary workstations, which we call *monitors*. The monitors speak both BGP and BGP-A; they observe all BGP traffic incident to the AS's existing routers and maintain BGP-A sessions with any monitors (or native BGP-A speakers where available) in adjacent ASes. The monitors also maintain tamper-evident logs and perform all cryptographic operations. Thus, the existing routers need not be modified.

There are two ways to configure a monitor [29]. A *proxying* monitor interposes on all BGP connections of its local AS. When it receives a BGP message from a local border router, it sends an equivalent BGP-A message to the remote BGP-A speaker (or monitor) and vice versa. A *mirroring* monitor snoops on the existing control connections, e.g., using a port replicator, the BGP monitoring protocol [31], or additional BGP sessions. Whenever it sees an outgoing message on the legacy BGP connection, it sends a BGP-A message with the same information over a separate connection to the neighbor's BGP-A speaker or monitor.

Mirroring monitors are safer because the routers do not depend on them. If a monitor fails, the routers can still send or receive routing updates via BGP and normal operation is not affected. On the other hand, mirroring monitors allow inconsistencies between the updates sent via BGP and BGP-A. Consider a case where a misconfigured or faulty router advertises some route A to its monitor and a different route B to the adjacent AS. The monitor would record route A in the tamper-evident log, and the AS could not be held accountable for route B.

To address this case, mirroring monitors maintain a third RIB for each peering point, which we will call *inRIB-BGP*. The inRIB contains the routes advertised via BGP-A as before, while the inRIB-BGP contains the routes received over the monitor's BGP sessions. Normally the two are identical; the scenario described earlier would manifest itself as an inconsistency between inRIB and inRIB-BGP in two adjacent ASes. Thus, an inconsistency cannot go undetected; however, an auditor cannot decide whether an inconsistency between inRIB and inRIB-BGP is caused by the audited AS or by its neighbor, and therefore must suspect both. Because BGP neighbors have a business relationship, they can be expected to swiftly sort out a demonstrated inconsistency between their advertised routes.

6.4 Incentives for deployment

If fault detection is to be deployed incrementally in the current Internet, we need good arguments to persuade ISPs to adopt it. Here, we present two arguments we believe to be compelling: ISPs can use fault detection as a distinguishing feature to attract more customers, and they can use it for root-cause analysis in the *entire* Internet, even in non-participating ASes.

Market forces: The first adopters of NetReview are likely to be large ISPs, such as tier-1 and tier-2 ASes, who tend to adopt new routing technology and best practices early. As a result, their routing performance is often excellent. These ASes can demonstrate their excellent performance by offering fault detection as a value-added service to their customers and thus distinguish themselves from the competition.

Once fault detection is on the market, competitors are encouraged to measure up by offering the service themselves. Thus, small islands of participants emerge. At this point, when a fault is caused by a non-participant, the participants can handle any complaints by proving that they are not the cause, and by tracing the problem to a non-participant just outside the island's perimeter, who must then handle the complaint. This creates an incentive for ASes to be inside the perimeter, and thus causes the islands to expand and the gaps between them to shrink.

Note that this approach works for NetReview because, unlike secure routing protocols like S-BGP, it is effective even in a small deployment of just a few ASes.

Root-cause analysis: As an additional benefit, participant ASes can use the fault detection system to diagnose faults *even if the cause is in a non-participating AS*. Since non-participants do not sign messages, do not maintain tamper-evident logs, and do not reveal any rules, we cannot guarantee that the diagnosis will always be accurate, and we cannot detect certain types of faults, such as policy violations. However, even an approximate diagnosis enables the AS to respond more effectively to faults.

Since non-participants do not have tamper-evident logs, we cannot directly apply auditing to find faults. Instead, we can use the participants' logs as a giant BGP looking glass that provides information about BGP updates from many vantage points. There are several proposed systems that can use this data to diagnose faults [10, 13, 21, 32]. In fact, because NetReview records a history of past states, it provides even more information than existing systems need; this could be used to develop even more powerful systems.

6.5 Accuracy in a partial deployment

When NetReview is used for root-cause analysis in a partial deployment, it returns a *candidate set* – a set of ASes that could have caused the fault. The size of this set depends on the size of the NetReview deployment. To estimate this dependency, we ran a simulation based on CAIDA's Internet AS topology [3], assuming a scenario in which an AS suspects that a given route has been spoofed. With NetReview in place, we can audit the participant ASes on the path and thus localize the fault to either a) a participant AS, b) a segment of non-participants between two participants, or c) the path suffix after the last participant. Here, we will ignore the possibility that the participant ASes record incorrect information.

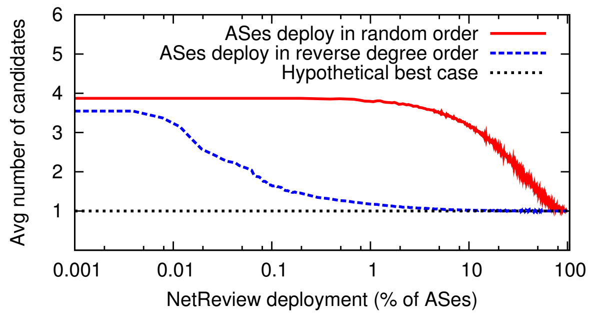


Figure 5: Fault localization under partial deployment. Shown is the average number of ASes that could have caused an observed failure.

For each deployment size, we simulated 10,000 trials as follows: we randomly picked an AS and calculated the shortest path to a random other AS using the Gao-Rexford conditions, then we picked a random AS on that path as the faulty AS and measured the number of candidates. We report averages over all 10,000 trials.

Figure 5 shows our results for two different deployment assumptions: either ASes deploy NetReview in random order, or in order of decreasing node degree. The first assumption is rather conservative; in practice, large ISPs typically run the latest routers, are among the first to apply best common practices and pride themselves on their good performance, so they are more likely to be early adopters.

In both cases, the average number of candidates starts at four (the average AS path length on the Internet). However, if the ASes with the most neighbors deploy NetReview first, the average decreases much more rapidly and reaches perfect localization with only a 15% deployment. The reason is that there are only about 12-15 tier-1 ASes; once these have deployed NetReview, faults can already be localized to one half of the path. 85% of the ASes in the Internet do not have customers of their own; once the other 15% participate, faults can be localized to one of them or to one of their customers.

This result shows that early adopters of a fault detection system like NetReview can derive considerable benefits from it; a deployment that includes the 0.1% highest-degree ASes would already be able to double the accuracy of its diagnoses. In contrast, fault prevention systems like S-BGP are only effective when they are already widely deployed.

7 Future work

In this section, we describe some advanced features that could be added to NetReview.

7.1 Simultaneously inspecting several ASes

NetReview inspects one log at a time, which is sufficient to detect protocol violations and policy violations. However, NetReview cannot detect problematic *interactions* between the policies of multiple ASes that way. An example is bad gadgets [14], which only arise when the routing policies of several ASes conflict in a circular fashion. To detect bad gadgets, NetReview would have to inspect the logs of multiple ASes simultaneously.

Technically, it is not difficult to fetch the logs from multiple ASes and to evaluate rules over multiple RIBs. However, routing policies are typically pair-wise confidential; thus, the check would have to be performed by a mutually trusted auditor. An alternative method to detect such policy conflicts, proposed in [15], is to have ASes annotate BGP advertisements with a *history* in a manner that preserves the privacy of the routing policies. Because NetReview records and publishes histories of BGP advertisements as part of its regular operation, this technique can be readily applied.

7.2 Detecting data-plane inconsistencies

In this paper, we have focused on providing fault detection for the *control plane* – the BGP announcements ASes send to each other. However, an AS could conceivably advertise one path in BGP and forward data packets on another, whether inadvertently or as part of an attack. NetReview already provides two mechanisms that can detect inconsistencies between the control and data planes: (i) it offers authoritative information about the route advertisement in the control plane, and (ii) it establishes the secure log that could also record observations about the data plane.

For example, suppose AS B advertises route "B C" to AS A but instead forwards A's traffic to AS D. If D passively monitors the traffic received from B, D can observe that A's packets are misrouted. D can add this observation to its log, and any auditors can thus obtain evidence of a data-plane inconsistency between B and D.

7.3 Internal audits

NetReview provides fault detection for BGP interdomain routing. It does not record any intra-domain routing messages in the tamper-evident log because these could reveal confidential information, such as the AS's internal topology.

However, NetReview could easily be adapted to cover intra-domain routing using a separate, private record. ASes could then perform internal audits to discover misconfigurations or compromised routers in their internal network, even when these routers have not (yet) caused a routing problem that would be visible to a neighbor.

8 Related Work

Detection: Anomaly detection techniques [7, 18, 21, 23, 36] use the BGP routing updates from one or more vantage points to build a *de facto* registry of the AS topology and prefix ownership. They raise an alarm upon receiving updates that disagree with the registry. Root-cause analysis (RCA) algorithms analyze BGP update messages from multiple vantage points to identify the AS(es) responsible for a routing change [4, 5, 10]. In RCA, each vantage point identifies a set of suspect ASes, then the sets are correlated to determine the potential culprit(s). The accuracy of RCA depends on the number and location of the vantage points. Unlike both RCA and anomaly detection, NetReview produces no false positives or false negatives, and it is not vulnerable to compromised ASes. In addition, NetReview can detect a larger class of faults, and it produces evidence that can be used to convince a third party.

AudIt [2] can determine which ASes are losing or delaying packets on the data plane. However, AudIt can only reveal the symptoms of a malfunctioning control plane, whereas control-plane fault detection can perform diagnosis.

Prevention: Secure routing protocols [20, 22, 33, 34] can ensure that (i) a route advertisement originates from the legitimate origin AS and that (ii) the AS-path of a route advertisement has not been modified or forged. On the one hand, secure routing protocols can prevent certain types of faults, whereas NetReview can only detect them; on the other hand, NetReview covers a larger class of faults, including policy violations (such as a faulty AS redistributing routes from one upstream provider to another), it can localize faults, and it provides incentives to avoid them. Perhaps more importantly, secure routing protocols do not provide appreciable benefits until many (if not all) ASes have adopted them, which explains in part why they have not yet been deployed, whereas NetReview is effective even in small deployments

N-BGP [29] uses trusted hardware to enforce a BGP safety specification for individual routers. Unlike N-BGP, NetReview does not require trusted hardware and it produces evidence of faults that can be verified by third parties. Moreover, NetReview is designed to check an entire AS's operation, not only against a safety specification but also against the AS's routing policy as specified in its peering agreements.

AIP [1] is a clean-slate redesign of IP that, among other things, would greatly simplify the deployment of a secure routing protocol. However, even if AIP were to replace IP entirely, it would be subject to the limitations of secure routing protocols described above.

Accountability: NetReview's tamper-evident log is based on the log in PeerReview [17], a general accountability framework for distributed systems. However, NetReview goes beyond PeerReview, which is based on assumptions that do not hold in interdomain routing. For example, PeerReview requires a certificate authority, it cannot operate in a partial deployment, it cannot protect the business secrets of ISPs, and it detects neither policy violations nor any other condition that involves more than one router.

9 Conclusion

In this paper, we have presented the design, implementation, and evaluation of NetReview, a fault detection system for interdomain routing. NetReview reliably detects incorrect behavior and links it to the responsible AS, while also enabling well-behaved ASes to prove they have adhered to the protocol and their routing policies. NetReview's correctness checks can detect and diagnose a wide variety of problems in BGP, including faulty equipment, buggy software, policy violations, and malicious attacks, which makes it an appealing alternative to specific solutions to any one of these problems. NetReview does not require changes to the underlying routers and is effective even in partial deployments. We believe that a fault detection system like NetReview can play an important role in improving the reliability, stability, and security of interdomain routing.

Acknowledgments

We are grateful to the anonymous reviewers and to our shepherd, Bruce Maggs, for their detailed and helpful comments. Paul Francis, Krishna Gummadi, Alex Fabrikant, Bryan Ford, and Sharon Goldberg provided valuable feedback on earlier versions of this paper. This work was done while Ioannis Avramopoulos was at Princeton University and supported by DHS grant W911NF-05-1-0417. Andreas Haeberlen was supported in part by US National Science Foundation grant ANI-0338856.

References

[1] D. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet protocol (AIP). In *Proceedings of SIGCOMM*, Aug 2008.

[2] K. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability for the Internet. In *Proc. IEEE International Conference on Network Protocols*, Oct. 2007.

[3] AS relationships dataset from CAIDA. `http://www.caida.org/data/active/as-relationships/index.xml`.

[4] M. Caesar, L. Subramanian, and R. H. Katz. A case for an Internet health monitoring system. In *Proc. USENIX Workshop on Hot Topics in System Dependability*, Jun. 2005.

[5] J. Chandrashekar, Z.-L. Zhang, and H. Peterson. Fixing BGP, one AS at a time. In *Proc. ACM SIGCOMM Network Troubleshooting Workshop*, 2004.

[6] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.

[7] Y.-J. Chi, R. Oliveira, and L. Zhang. Cyclops: the AS-level connectivity observatory. *SIGCOMM Comput. Commun. Rev.*, 38(5):5–16, 2008.

[8] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *Proceedings of NSDI'05*, May 2005.

[9] N. Feamster, Z. M. Mao, and J. Rexford. BorderGuard: Detecting cold potatoes from peers. In *Proc. Internet Measurement Conference*, Oct 2004.

[10] A. Feldmann, O. Maennel, Z. Mao, A. Berger, and B. Maggs. Locating Internet routing instabilities. In *Proc. ACM SIGCOMM*, Aug.-Sept. 2004.

[11] L. Gao and J. Rexford. Stable Internet routing without global coordination. *IEEE/ACM Transactions on Networking*, 9(6):681–692, Dec 2001.

[12] GNU Zebra. `http://www.zebra.org/`.

[13] G. Goodell, W. Aiello, T. Griffin, J. Ioannidis, P. McDaniel, and A. Rubin. Working around BGP: An incremental approach to improving security and accuracy of interdomain routing. In *Proc. Network and Distributed Systems Security*, Feb 2003.

[14] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Transactions on Networking*, 10(2):232–243, April 2002.

[15] T. G. Griffin and G. Wilfong. A safe path vector protocol. In *Proc. INFOCOM*, Mar. 2000.

[16] A. Haeberlen, P. Kuznetsov, and P. Druschel. The case for Byzantine fault detection. In *Proc. HotDep'06*. USENIX, Nov 2006.

[17] A. Haeberlen, P. Kuznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *Proc. Symposium on Operating Systems Principles*, Oct 2007.

[18] X. Hu and Z. M. Mao. Accurate real-time identification of IP prefix hijacking. In *Proc. IEEE Symposium on Security and Privacy*, May 2007.

[19] Y.-C. Hu, D. McGrew, A. Perrig, B. Weis, and D. Wendlandt. (R)evolutionary bootstrapping of a global PKI for securing BGP. In *Proc. HotNets Workshop*, Nov 2006.

[20] Y.-C. Hu, A. Perrig, and M. Sirbu. SPV: Secure path vector routing for securing BGP. In *Proc. ACM SIGCOMM*, 2004.

[21] J. Karlin, S. Forrest, and J. Rexford. Autonomous security for autonomous systems. *Computer Networks, special issue on Complex Computer and Communication Networks*, Oct 2008.

[22] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (S-BGP). *IEEE JSAC*, 18(4):582–592, Apr 2000.

[23] M. Lad, D. Massey, D. Pei, Y. Wu, B. Zhang, and L. Zhang. PHAS: A prefix hijack alert system. In *Proc. USENIX Security Symposium*, Aug. 2006.

[24] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *Proc. ACM SIGCOMM*, Sep 2002.

[25] O. Nordstroem and C. Dovrolis. Beware of BGP attacks. *ACM Computer Communications Review (CCR)*, Apr 2004.

[26] C. Partridge, T. Mendez, and W. Milliken. Host anycasting service. RFC 1546, Nov 1993.

[27] A. Ramachandran and N. Feamster. Understanding the network-level behavior of spammers. In *Proc. ACM SIGCOMM*, Sep 2006.

[28] Y. Rekhter, T. Li, and S. Hares. A border gateway protocol 4 (BGP-4). RFC 4271, Jan 2006.

[29] P. Reynolds, O. Kennedy, E. G. Sirer, and F. B. Schneider. Securing BGP using external security monitors. Technical Report TR-2006-2065, Cornell University, Computing and Information Science, Dec 2006.

[30] RouteViews project. http://www.routeviews.org/.

[31] J. Scudder, R. Fernando, and S. Stuart. BGP monitoring protocol, Nov 2008. Internet Draft, draft-ietf-grow-bmp-00.

[32] R. Teixeira and J. Rexford. A measurement framework for pin-pointing routing changes. In *Proc. ACM SIGCOMM Network Troubleshooting Workshop*, Sep 2004.

[33] T. Wan, E. Kranakis, and P. C. van Oorschot. Pretty secure BGP (psBGP). In *Proc. Network and Distributed System Security Symposium*, Feb 2005.

[34] R. White. Securing BGP through secure origin BGP. *Internet Protocol Journal*, 6(3), 2003.

[35] J. Wu, Z. M. Mao, J. Rexford, and J. Wang. Finding a needle in a haystack: Pinpointing significant BGP routing changes in an IP network. In *Processings of NSDI'05*, May 2005.

[36] C. Zheng, L. Ji, D. Pei, J. Wang, and P. Francis. A light-weight distributed scheme for detecting IP prefix hijacks in real-time. In *Proc. ACM SIGCOMM*, Aug. 2007.

[37] E. Zmijewski. Longer is not always better. `http://www.renesys.com/blog/2009/02/longer-is-not-better.shtml`.

Making Routers Last Longer with ViAggre

Hitesh Ballani, *Cornell University*
Paul Francis, *Cornell University*
Tuan Cao, *Cornell University*
Jia Wang, *AT&T Labs – Research*

Abstract

This paper presents ViAggre (Virtual Aggregation), a "configuration-only" approach to shrinking the routing table on routers. ViAggre does not require any changes to router software and routing protocols and can be deployed independently and autonomously by any ISP. ViAggre is effectively a scalability technique that allows an ISP to modify its internal routing such that individual routers in the ISP's network only maintain a part of the global routing table.

We evaluate the application of ViAggre to a few tier-1 and tier-2 ISPs and show that it can reduce the routing table on routers by an order of magnitude while imposing almost no traffic stretch and negligible load increase across the routers. We also deploy Virtual Aggregation on a testbed comprising of Cisco routers and benchmark this deployment. Finally, to understand and address concerns regarding the configuration overhead that our proposal entails, we implement a configuration tool that automates ViAggre configuration. While it remains to be seen whether most, if not all, of the management concerns can be eliminated through such automated tools, we believe that the simplicity of the proposal and its possible short-term impact on routing scalability suggest that it is an alternative worth considering.

I. Introduction

The Internet default-free zone (DFZ) routing table has been growing rapidly for the past few years [20]. Looking ahead, there are concerns that as the IPv4 address space runs out, hierarchical aggregation of network prefixes will further deteriorate resulting in a substantial acceleration in the growth of the routing table [31]. A growing IPv6 deployment would worsen the situation even more [29].

The increase in the size of the DFZ routing table has several harmful implications for inter-domain routing.[1] [31] discusses these in detail. At a technical level, increasing routing table size may drive high-end router design into various engineering limits. For instance, while memory and processing speeds might just scale with a growing routing system, power and heat dissipation capabilities may not [30]. On the business side, the performance requirements for forwarding while being able to access a large routing table imply that the cost of forwarding packets increases and hence, networks become less cost-effective [27]. Further, it makes provisioning of networks harder since it is difficult to estimate the usable lifetime of routers, not to mention the cost of the actual upgrades. As a matter of fact, instead of upgrading their routers, a few ISPs have resorted to filtering out some small prefixes (mostly /24s) which implies that parts of the Internet may not have reachability to each other [19]. This suggests that ISPs are willing to undergo some pain to avoid the cost of router upgrades.

Such concerns regarding FIB size growth, along with problems arising from a large RIB and the concomitant convergence issues, were part of the reasons that led a recent Internet Architecture Board workshop to conclude that scaling the routing system is one of the most critical challenges of near-term Internet design [30]. The severity of these problems has also prompted a slew of routing proposals [7,8,11,14,18,29,32,40]. All these proposals require changes in the routing and addressing architecture of the Internet. This, we believe, is the nature of the beast since some of the fundamental Internet design choices limit routing scalability; the overloading of IP addresses with "who" and "where" semantics represents a good example [30]. However, the very fact that they require architectural change has contributed to the non-deployment of these proposals.

This paper takes the position that a major architectural change is unlikely and it may be more pragmatic to approach the problem through a series of incremental, individually cost-effective upgrades. Guided by this and the aforementioned implications of a rapidly growing DFZ FIB, this paper proposes *Virtual Aggregation or ViAggre*, a scalability technique that focuses primarily on shrinking the FIB size on routers. ViAggre is a "configuration-only" solution that *applies to legacy routers*. Further, ViAggre can be *adopted independently and autonomously by any ISP* and hence the bar for its deployment is much lower. The key idea behind ViAggre is very simple: an ISP adopting ViAggre divides the responsibility for maintaining the global routing table amongst its routers such that individual routers only maintain a part of the routing table. Thus, this paper makes the following contributions:

- We discuss two deployment options through which an ISP can adopt ViAggre. The first one uses *FIB* *suppression* to shrink the FIB of all the ISP's routers while the second uses *route filtering* to shrink both the FIB and RIB on all *data-path* routers.

- We analyze the application of ViAggre to an actual tier-1 ISP and several inferred (Rocketfuel [37]) ISP topologies. We find that ViAggre can reduce FIB size by more than an order of magnitude with negligible stretch on the ISP's traffic and very little increase in load across the ISP's routers. Based on predictions of future routing table growth, we estimate that ViAggre can be used to extend the life of already outdated routers by more than 10 years.
- We propose utilizing the notion of prefix popularity to reduce the impact of ViAggre on the ISP's traffic and use a two-month study of a tier-1 ISP's traffic to show the feasibility of such an approach.
- As a proof-of-concept, we configure test topologies comprising of Cisco routers (on WAIL [3]) according to the ViAggre proposal. We use the deployment to benchmark the control-plane processing overhead that ViAggre entails. One of the presented designs actually reduces the amount of processing done by routers and preliminary results show that it can reduce convergence time too. The other design has high overhead due to implementation issues and needs more experimentation.
- ViAggre involves the ISP reconfiguring its routers which can be a deterrent to adoption. We quantify this configuration overhead. We also implement a configuration tool that, given the ISPs existing configuration files, can automatically generate the configuration files needed for ViAggre deployment. We discuss the use of this tool on our testbed.

Overall, the incremental version of ViAggre that this paper presents can be seen as little more than a simple and structured hack that assimilates ideas from existing work including, but not limited to, VPN tunnels and CRIO [40]. We believe that its very simplicity makes ViAggre an attractive short-term solution that provides ISPs with an alternative to upgrading routers in order to cope with routing table growth till more fundamental, long-term architectural changes can be agreed upon and deployed in the Internet. However, the basic ViAggre idea can also be applied in a clean-slate fashion to address routing concerns beyond FIB growth. While we defer the design and the implications of such a non-incremental ViAggre architecture for future work, the notion that the same concept has potential both as an immediate alleviative and as the basis for a next-generation routing architecture seems interesting and worth exploring.

II. ViAggre design

ViAggre allows individual ISPs in the Internet's DFZ to do away with the need for their routers to maintain routes for all prefixes in the global routing table. An ISP adopting ViAggre divides the global address space into a set of *virtual prefixes* such that the virtual prefixes are larger than any aggregatable (real) prefix in use today. So, for instance, an ISP could divide the IPv4 address space into 128 parts with a /7 virtual prefix representing each part (0.0.0.0/7 to 254.0.0.0/7). Note that such a naïve allocation would yield an uneven distribution of real prefixes across the virtual prefixes. However, the virtual prefixes need not be of the same length and hence, the ISP can choose them such that they contain a comparable number of real prefixes.

The virtual prefixes are not topologically valid aggregates, i.e. there is not a single point in the Internet topology that can hierarchically aggregate the encompassed prefixes. ViAggre makes the virtual prefixes aggregatable by organizing *virtual networks*, one for each virtual prefix. In other words, a virtual topology is configured that causes the virtual prefixes to be aggregatable, thus allowing for routing hierarchy that shrinks the routing table. To create such a virtual network, some of the ISP's routers are assigned to be within the virtual network. These routers maintain routes for all prefixes in the virtual prefix corresponding to the virtual network and hence, are said to be *aggregation points* for the virtual prefix. A router can be an aggregation point for multiple virtual prefixes and is required to only maintain routes for prefixes in the virtual prefixes it is aggregating.

Given this, a packet entering the ISP's network is routed to a close-by aggregation point for the virtual prefix encompassing the actual destination prefix. This aggregation point has a route for the destination prefix and forwards the packet out of the ISP's network in a tunnel. In figure 1 (figure details explained later), router C is an aggregation point for the virtual prefix encompassing the destination prefix and B $\rightarrow$ C $\rightarrow$ D is one such path through the ISP's network.

A. Design Goals

The discussion above describes ViAggre at a conceptual level. While the design space for organizing an ISP's network into virtual networks has several dimensions, this paper aims for deployability and hence is guided by two major design goals:

1) *No changes to router software and routing protocols*: The ISP should not need to deploy new data-plane or control-plane mechanisms.
2) *Transparent to external networks*: An ISP's decision to adopt the ViAggre proposal should not impact its interaction with its neighbors (customers, peers and providers).

These goals, in turn, limit what can be achieved through the ViAggre designs presented here. Routers today have a Routing Information Base (RIB) generated by the routing protocols and a Forwarding Information Base (FIB) that is used for forwarding the packets. Consequently, the FIB is optimized for looking up destination addresses and is maintained on fast(er) memory, generally on the line cards themselves [31]. All things being equal, it would be nice to shrink both the RIB and the FIB for all ISP devices, as well as make other improvements such as shorter convergence time.

While the basic ViAggre idea can be used to achieve these benefits (section VI), we have not been able to reconcile them with the aforementioned design goals. Instead, this paper is based on the hypothesis that given the performance and monetary implications of the FIB size for routers, an immediately deployable solution that reduces FIB size is useful. Actually, one of the presented designs also shrinks the RIB on routers; only components that are off the data path (i.e. route reflectors) need to maintain the full RIB. Further, this design is shown to help with route convergence time too.

B. Design-I: FIB Supression

This section details one way an ISP can deploy virtual prefix based routing while satisfying the goals specified in the previous section. The discussion below applies to IPv4 (and BGPv4) although the techniques detailed here work equally well for IPv6. The key concept behind this design is to operate the ISP's internal distribution of BGP routes untouched and in particular, to populate the RIB on routers with the full routing table but to suppress most prefixes from being loaded in the FIB of routers. A standard feature on routers today is *FIB Suppression* which can be used to prevent routes for individual prefixes in the RIB from being loaded into the FIB. We have verified support for FIB suppression as part of our ViAggre deployment on Cisco 7300 and 12000 routers. Documentation for Juniper [43] and Foundry [42] routers specify this feature too. We use this as described below.

The ISP does not modify its routing setup – the ISP's routers participate in an intra-domain routing protocol that establishes internal routes through which the routers can reach each other while BGP is used for inter-domain routing just as today. For each virtual prefix, the ISP designates some number of routers to serve as aggregation points for the prefix and hence, form a virtual network. Each router is configured to only load prefixes belonging to the virtual prefixes it is aggregating into its FIB while suppressing all other prefixes.

Given this, the ISP needs to ensure that packets to any prefix can flow through the network in spite of the fact that only a few routers have a route to the prefix. This is achieved as follows:

– *Connecting Virtual Networks.* Aggregation points for a virtual prefix originate a route to the virtual prefix that is distributed throughout the ISP's network but not outside. Specifically, an aggregation point advertises the virtual prefix to its iBGP peers. A router that is not an aggregation point for the virtual prefix would choose the route advertised by the aggregation point closest to it and hence, forward packets destined to any prefix in the virtual prefix to this aggregation point.[2]

– *Sending packets to external routers.* When a router receives a packet destined to a prefix in a virtual prefix it is aggregating, it can look up its FIB to determine the route for the packet. However, such a packet cannot be forwarded in the normal hop-by-hop fashion since a router that is not an aggregation point for the virtual prefix in question might forward the packet back to the aggregation point, resulting in a loop. Hence, the packet must be tunneled from the aggregation point to the external router that was selected as the BGP NEXT_HOP. While the ISP can probably choose from many tunneling technologies, we use MPLS Label Switched Paths (LSPs) for such tunnels. This choice was influenced by the fact that MPLS is widely supported in routers, is used by ISPs, and operates at wire speed. Further, protocols like LDP [1] automate the establishment of MPLS tunnels and hence, reduce the configuration overhead.

However, a LSP from the aggregation point to an external router would require cooperation from the neighboring ISP. To avoid this, every edge router of the ISP initiates a LSP for every external router it is connected to. Thus, all the ISP routers need to maintain LSP mappings equal to the number of external routers connected to the ISP, a number much smaller than the routes in the DFZ routing table (we relax this constraint in section IV-B). Note that even though the tunnel endpoint is the external router, the edge router can be configured to strip the MPLS label from the data packets before forwarding them onto the external router. This, in turn, has two implications. First, external routers don't need to be aware of the adoption of ViAggre by the ISP. Second, even the edge router does not need a FIB entry for the destination prefix, instead it chooses the external router to forward the packets to based on the MPLS label of the packet. The behavior of the edge router here is similar to the penultimate hop in a VPN scenario and is achieved through standard configuration.

We now use a concrete example to illustrate the flow of

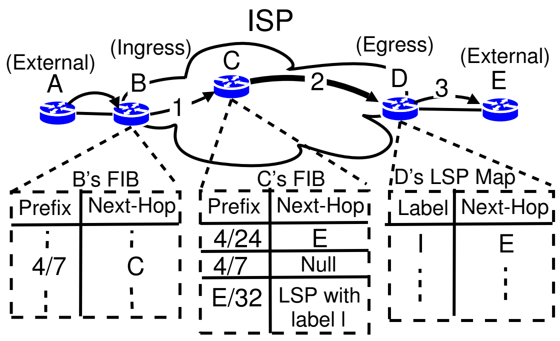


Fig. 1. Path of packets destined to prefix 4.0.0.0/24 (or, 4/24) between external routers A and E through an ISP with ViAggre. Router C is an aggregation point for virtual prefix 4.0.0.0/7 (or, 4/7).

packets through an ISP network that is using ViAggre. Figure 1 shows the relevant routers. The ISP is using /7s as virtual prefixes and router C is an aggregation point for one such virtual prefix 4.0.0.0/7. Edge router D initiates a LSP to external router E with label *l* and hence, the ISP's routers can get to E through MPLS tunneling. The figure shows the path of a packet destined to prefix 4.0.0.0/24, which is encompassed by 4.0.0.0/7, through the ISP's network. The path from the ingress router B to the external router E comprises three segments:

1) VP-routed: Ingress router B is not an aggregation point for 4.0.0.0/7 and hence, forwards the packet to aggregation point C.
2) MPLS-LSP: Router C, being an aggregation point for 4.0.0.0/7, has a route for 4.0.0.0/24 with BGP NEXT_HOP set to E. Further, the path to router E involves tunneling the packet with MPLS label *l*.
3) Map-routed: On receiving the tunneled packet from router C, egress router D looks up its MPLS label map, strips the MPLS header and forwards the packet to external router E.

C. Design-II: Route Reflectors

The second design offloads the task of maintaining the full RIB to devices that are off the data path. Many ISPs use route-reflectors for scalable internal distribution of BGP prefixes and we require only these route-reflectors to maintain the full RIB. For ease of exposition, we assume that the ISP is already using per-PoP route reflectors that are off the data path, a common deployment model for ISPs using route reflectors.

In the proposed design, the external routers connected to a PoP are made to peer with the PoP's route-reflector. This is necessary since the external peer may be advertising the entire DFZ routing table and we don't want all these routes to reside on any given data-plane router. The route-reflector also has iBGP peerings with other route-reflectors and with the routers in its PoP. Egress filters are used on the route-reflector's peerings with the PoP's routers to ensure that a router only gets routes for the prefixes it is aggregating. This shrinks both the RIB and the FIB on the routers. The data-plane operation and hence, the path of packets through the ISP's network remains the same as with the previous design.

With this design, a PoP's route-reflector peers with all the external routers connected to the PoP. The RIB size on a BGP router depends on the number of peers it has and hence, the RIB for the route-reflectors can potentially be very large. If needed, the RIB requirements can be scaled by using multiple route-reflectors which may also be needed to provide customised routes to the PoP's neighbors. Note that the RIB scaling properties here are better than in the status quo. Today, edge routers have no choice but to peer with the directly connected external routers and maintain the resulting RIB. Replicating these routers is prohibitive because of their cost but the same does not apply to off-path route-reflectors, which could even be BGP software routers.

D. Design Comparison

As far as the configuration is concerned, configuring suppression of routes on individual routers in design-I is comparable, at least in terms of complexity, to configuring egress filters on the route-reflectors. In both cases, the configuration can be achieved through BGP route-filtering mechanisms (access-lists, prefix-lists, etc.).

Design-II, apart from shrinking the RIB on the routers, does not require the route suppression feature on routers. Further, as we detail in section V-B, design-II reduces the ISP's route propagation time while the specific filtering mechanism used in design-I increases it. However, design-II does require the ISP's eBGP peerings to be reconfigured which, while straightforward, violates our goal of not impacting neighboring ISPs.

E. Network Robustness

ViAggre causes packets to be routed through an aggregation point which leads to robustness concerns. When an aggregation point for a virtual prefix fails, routers using that aggregation point are re-routed to another aggregation point through existing mechanisms without any explicit configuration by the ISP. In case of design-I, a router has routes to all aggregation points for a given virtual prefix in its RIB and hence, when the aggregation point being used fails, the router installs the second closest aggregation point into its FIB and packets are re-routed almost instantly. With design-II, it is the route-reflector that chooses the alternate aggregation point and advertises this to the routers in its PoP. Hence, as long as another aggregation point exists, failover happens automatically and at a fast rate.

F. Routing popular prefixes natively

The use of aggregation points implies that packets in ViAggre may take paths that are longer than native paths. Apart from the increased path length, the packets may incur queuing delay at the extra hops. The extra hops also result in an increase in load on the ISP's routers and links and a modification in the distribution of traffic across them.

Past studies have shown that a large majority of Internet traffic is destined to a very small fraction of prefixes [10,13,34,38]. The fact that routers today have no choice but to maintain the complete DFZ routing table implies that this observation wasn't very useful for routing configuration. However, with ViAggre, individual routers only need to maintain routes for a fraction of prefixes. The ISP can thus configure its ViAggre setup such that the small fraction of popular prefixes are in the FIB of every router and hence, are routed natively. For design-I, this involves configuring each router with a set of popular prefixes that should not be suppressed from the FIB. For design-II, a PoP's route-reflector can be configured to not filter advertisements for popular prefixes from the PoP's routers. Beyond this, the ISP may also choose to install customer prefixes into its routers such that they don't incur any stretch. The rest of the proposal involving virtual prefixes remains the same and ensures that individual routers only maintain routes for a fraction of the unpopular prefixes. In section IV-B.4, we analyze Netflow data from a tier-1 ISP network to show that not only such an approach is feasible, it also addresses all the concerns raised above.

III. Allocating aggregation points

An ISP adopting ViAggre would obviously like to minimise the stretch imposed on its traffic. Ideally, an ISP would deploy an aggregation point for all virtual prefixes in each of its PoPs. This would ensure that for every virtual prefix, a router chooses the aggregation point in the same PoP and hence, the traffic stretch is minimal. However, this may not be possible in practice. This is because ISPs, including tier-1 ISPs, often have some small PoPs with just a few routers and therefore there may not be enough cumulative FIB space in the PoP to hold all the actual prefixes. More generally, ISPs may be willing to bear some stretch for substantial reductions in FIB size. To achieve this, the ISP needs to be smart about the way it designates routers to aggregate virtual prefixes and in this section we explore this choice.

A. Problem Formulation

We first introduce the notation used in the rest of this section. Let *T* represent the set of prefixes in the Internet routing table, *R* be the set of ISP's routers and *X* is the set of external routers directly connected to the ISP. For each $r \in R$, P_r represents the set of popular prefixes for router r. *V* is the set of virtual prefixes chosen by the ISP and for each $v \in V$, n_v is the number of prefixes in v. We use two matrices, $D = (d_{i,j})$ that gives the distance between routers i and j and $W = (w_{i,j})$ that gives the IGP metric for the IGP-established path between routers i and j. We also define two relations:

– "BelongsTo" relation B: $T \rightarrow V$ such that $B(p)=v$ if prefix p belongs to or is encompassed by virtual prefix v.

– "Egress" relation E: R x $T \rightarrow R$ such that $E(i, p)=j$ if traffic to prefix p from router i egresses at router j.

The mapping relation A: $R \rightarrow 2^V$ captures how the ISP assigns aggregation points; i.e. $A(r) = \{v_1 \dots v_n\}$ implies that router r aggregates virtual prefixes $\{v_1 \dots v_n\}$. Given this assignment, we can determine the aggregation point any router uses for its traffic to each virtual prefix. This is captured by the "Use" relation U: R x $V \rightarrow R$ where $U(i, v) = j$ or router i uses aggregation point j for virtual prefix v if the following conditions are satisfied:

$$\begin{array}{llcll} 1) & v & \in & A(j) & \\ 2) & w_{i,j} & \leq & w_{i,k} & \forall k \in R, v \in A(k) \end{array}$$

Here, condition 1) ensures that router j is an aggregation point for virtual prefix v. Condition 2) captures the operation of BGP with design-I and ensures that a router chooses the aggregation point that is closest in terms of IGP metrics.[3]

Using this notation, we can express the FIB size on routers and the stretch imposed on traffic.

1) Routing State: In ViAggre, a router needs to maintain routes to the (real) prefixes in the virtual prefixes it is aggregating, routes to all the virtual prefixes themselves and routes to the popular prefixes. Further, the router needs to maintain LSP mappings for LSPs originated by the ISP's edge routers with one entry for each external router connected to the ISP. Hence, the "routing state" for the router r, hereon simply referred to as the FIB size (F_r), is given by:

$$F_r = \sum_{v \in A(r)} n_v + |V| + |P_r| + |X|$$

The **Worst FIB size** and the **Average FIB size** are defined as follows:

$$\text{Worst FIB size} = \max_{r \in R}(F_r)$$

$$\text{Average FIB size} = \sum_{r \in R}(F_r)/|R|$$

2) Traffic Stretch: If router i uses router k as an aggregation point for virtual prefix v, packets from router i to a prefix p belonging to v are routed through router k. Hence, the stretch (S) imposed on traffic to prefix p from router i is given by:

$$\begin{aligned} S_{i,p} &= 0, & p \in P_i \\ &= (d_{i,k} + d_{k,j} - d_{i,j}), & p \in (T - P_i),\ v = B(p) \\ && k = U(i,v)\ \&\ j = E(k,p) \end{aligned}$$

The **Worst Stretch** and **Average Stretch** are defined as follows:

$$\begin{aligned} \text{Worst Stretch} &= \max_{i \in R, p \in T}(S_{i,p}) \\ \text{Average Stretch} &= \sum_{i \in R, p \in T} (S_{i,p})/(|R| * |T|) \end{aligned}$$

Problem: ViAggre, through the use of aggregation points, trades off an increase in path length for a reduction in routing state. The ISP can use the assignment of aggregation points as a knob to tune this trade-off. Here we consider the simple goal of minimising the FIB Size on the ISP's routers while bounding the stretch. Specifically, the ISP needs to assign aggregation points by determining a mapping A that

$$\begin{aligned} &\text{min} \quad \text{Worst FIB Size} \\ &\text{s.t.} \quad \text{Worst Stretch} \leq C \end{aligned}$$

where C is the specified constraint on Worst Stretch. Note that much more complex formulations are possible. Our focus on worst-case metrics is guided by practical concerns – the Worst FIB Size dictates how the ISP's routers need to be provisioned while the Worst Stretch characterizes the most unfavorable impact of the use of ViAggre. Specifically, bounding the Worst Stretch allows the ISP to ensure that its existing SLAs are not breached and applications sensitive to increase in latency (example, VOIP) are not adversely affected.

B. A Greedy Solution

The problem of assigning aggregation points while satisfying the conditions above can be mapped to the MultiCommodity Facility Location (MCL) problem [33]. MCL is NP-hard and [33] presents a logarithmic approximation algorithm for it. Here we discuss a greedy approximation solution to the problem, similar to the algorithm in [25].

The first solution step is to determine that if router i were to aggregate virtual prefix v, which routers can it serve without violating the stretch constraint. This is the $can\_serve_{i,v}$ set and is defined as follows:

$$\begin{aligned} can\_serve_{i,v} &= \{j \mid j \in R, (\forall p \in T, B(p) = v, E(i,p) \\ &\qquad = k, (d_{j,i} + d_{i,k} - d_{j,k}) \leq C)\} \end{aligned}$$

Given this, the key idea behind the solution is that any assignment based on the can_serve relation will have Worst Stretch less than C. Hence, our algorithm designates routers to aggregate virtual prefixes in accordance with the can_serve relation while greedily trying to minimise the Worst FIB Size. The algorithm, shown below, stops when each router can be served by at least one aggregation point for each virtual prefix.

```
$Worst\_FIB\_Size$=0
for all r in R do
    for all v in V do
        Calculate $can\_serve_{r,v}$
Sort V in decreasing order of $n_v$
for all v in V do
    Sort R in decreasing order of $|can\_serve_{r,v}|$
    repeat
        for all r in R do
            if $(F_r + n_v) \leq Worst\_FIB\_Size$ then
                A[r]=A[r] ∪ v                    # Assign v to r
                $F_r = F_r + n_v$        # r's FIB size increases
                Mark all routers in $can\_serve_{r,v}$ as served
            if All routers are served for v then
                break
        if All routers are not served for v then
                    # $Worst\_FIB\_Size$ needs to be raised
            for all r in R do
                if v ∉ A[r] then
                        # r is not an aggregation point for v
                    A[r]=A[r] ∪ v
                    $F_r = F_r + n_v$
                    $Worst\_FIB\_Size$ = $F_r$
                    break
    until All Routers are served for virtual prefix v
```

IV. Evaluation

In this section we evaluate the application of ViAggre to a few Internet ISPs.

A. Metrics of Interest

We defined (**Average** and **Worst**) **FIB Size** and **Stretch** metrics in section III-A. Here we define other metrics that we use for ViAggre evaluation.

1) *Impact on Traffic:* Apart from the stretch imposed, another aspect of ViAggre's impact is the amount of traffic affected. To account for this, we define **traffic impacted** as the fraction of the ISP's traffic that uses a different router-level path than the native path. Note that in many cases, a router will use an aggregation point for the destination virtual prefix in the same PoP and hence, the packets will follow the same PoP-level path as before. Thus, another metric of interest is the **traffic stretched**, the fraction of traffic that is forwarded along a different PoP-level path than before. In effect, this represents the change in the distribution of traffic across the ISP's inter-PoP links and hence, captures how ViAggre interferes with the ISP's inter-PoP traffic engineering.

2) *Impact on Router Load:* The extra hops traversed by traffic increases the traffic load on the ISP's routers. We define the **load increase** across a router as the extra traffic it needs to forward due to ViAggre, as a fraction of the traffic it forwards natively.

B. Tier-1 ISP Study

We analysed the application of ViAggre to a large tier-1 ISP in the Internet. For our study, we obtained the ISP's router-level topology (to determine router set *R*) and the routing tables of routers (to determine prefix set *T* and the Egress *E* and BelongsTo *B* relations). We used information about the geographical locations of the routers to determine the Distance matrix *D* such that $d_{i,j}$ is 0 if routers i and j belong to the same PoP (and hence, are in the same city) else $d_{i,j}$ is set to the propagation latency corresponding to the great circle distance between i and j. Further, we did not have information about the ISP's link weights. However, guided by the fact that intra-domain traffic engineering is typically latency-driven [36], we use the Distance matrix *D* as the Weight matrix *W*. We also obtained the ISP's traffic matrix; however, in order to characterise the impact of vanilla ViAggre, the first part of this section assumes that the ISP does not consider any prefixes as popular.

1) *Deployment decisions:* The ISP, in order to adopt ViAggre, needs to decide what virtual prefixes to use and which routers aggregate these virtual prefixes. We describe the approaches we evaluated.

– *Determining set V.* The most straightforward way to select virtual prefixes while satisfying the two conditions specified in section II is to choose large prefixes (/6s, /7s, etc.) as virtual prefixes. We assume that the ISP uses /7s as its virtual prefixes and refer to this as the "/7 allocation".

However, such selection of virtual prefixes could lead to a skewed distribution of (real) prefixes across them with some virtual prefixes containing a large number of prefixes. For instance, using /7s as virtual prefixes implies that the largest virtual prefix (202.0.0.0/7) contains 22,772 of the prefixes in today's BGP routing table or 8.9% of the routing table. Since at least one ISP router needs to aggregate each virtual prefix, such large virtual prefixes would inhibit the ISP's ability to reduce the Worst FIB size on its routers. However, as we mentioned earlier, the virtual prefixes need not be of the same length and so large virtual prefixes can be split to yield smaller virtual prefixes. To study the effectiveness of this approach, we started with /7s as virtual prefixes and split each of them such that the resulting virtual prefixes were still larger than any prefix in the Internet routing table. This yielded 1024 virtual prefixes with the largest containing 4,551 prefixes or 1.78% of the BGP routing table. We also use this virtual prefix allocation for our evaluation and refer to it as "Uniform Allocation".

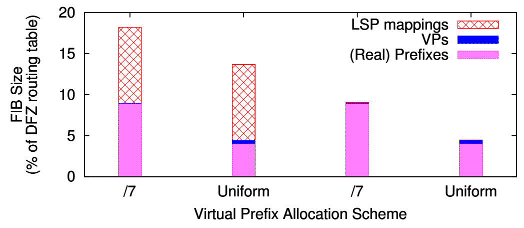


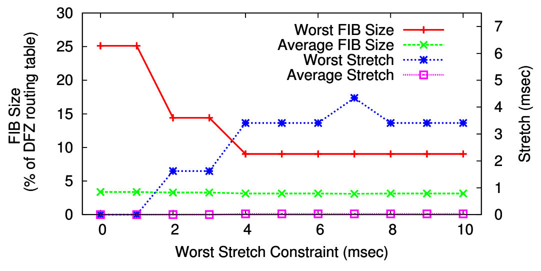
Fig. 2. FIB composition for the router with the largest FIB, C=4ms and no popular prefixes.

– *Determining mapping A.* We implemented the algorithm described in section III-B and use it to designate routers to aggregate virtual prefixes.

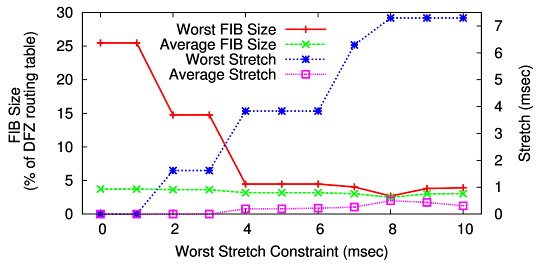
2) *Router FIB:* We first look at the size and the composition of the FIB on the ISP's routers with a ViAggre deployment. Specifically, we focus on the router with the largest FIB for a deployment where the worst-case stretch (C) is constrained to 4ms. The first two bars in figure 2 show the FIB composition for a /7 and uniform allocation respectively. With a /7 allocation, the router's FIB contains 46,543 entries which represents 18.2% of the routing table today. This includes 22,772 prefixes, 128 virtual prefixes, 23,643 LSP mappings and 0 popular prefixes. As can be seen, in both cases, the LSP mappings for tunnels to the external routers contribute significantly to the FIB. This is because the ISP has a large number of customer routers that it has peerings with.

However, we also note that customer ISPs do not advertise the full routing table to their provider. Hence, edge routers of the ISP could maintain routes advertised by customer routers in their FIB, advertise these routes onwards with themselves as the BGP NEXT_HOP and only initiate LSP advertisements for themselves and for peer and provider routers connected to them. With such a scheme, the number of LSP mappings that the ISP's routers need to maintain and the MPLS overhead in general reduces significantly. The latter set of bars in figure 2 shows the FIB composition with such a deployment for the router with the largest FIB. For the /7 allocation, the Worst FIB size is 23,101 entries (9.02% of today's routing table) while for the Uniform allocation, it is 10,226 entries (4.47%). In the rest of this section, we assume this model of deployment.

3) *Stretch Vs. FIB Size:* We ran the assignment algorithm with Worst Stretch Constraint (C) ranging from 0 to 10 ms and determined the (Average and Worst) Stretch and FIB Size of the resulting ViAggre deployment. Figure 3(a) plots these metrics for the /7 allocation. The Worst FIB size, shown as a fraction of the DFZ routing table size today, expectedly reduces as the constraint on Worst Stretch is relaxed. However, beyond C=4ms, the Worst FIB Size remains constant. This is because the largest virtual prefix with a /7 allocation encompasses 8.9% of the DFZ routing table and the



(a) With /7 allocation



(b) With Uniform allocation

Fig. 3. Variation of FIB Size and Stretch with Worst Stretch constraint and no popular prefixes.

		Today	ViAggre			
	Worst Stretch (ms)	–	0	2	4	8
239K FIB	Quad. Fit	Expired	2015	2020	2039	2051
	Expo. Fit	Expired	2018	2022	2031	2035
1M FIB	Quad. Fit	2015	2033	2044	2081	2106
	Expo. Fit	2018	2029	2033	2042	2046

TABLE I
ESTIMATES FOR ROUTER LIFE WITH VIAGGRE

Worst FIB Size cannot be any less than 9.02% (0.12% overhead is due to virtual prefixes and LSP mappings). Figure 3(b) plots the same metrics for the Uniform allocation and shows that the FIB can be shrunk even more. The figure also shows that the Average FIB Size and the Average stretch are expectedly small throughout. The anomaly beyond C=8msec in figure 3(b) results from the fact that our assignment algorithm is an approximation that can yield non-optimal results.

Another way to quantify the benefits of ViAggre is to determine the extension in the life of a router with a specified memory due to the use of ViAggre. As proposed in [21], we used data for the DFZ routing table size from Jan'02 to Dec'07 [20] to fit a quadratic model to routing table growth. Further, it has been claimed that the DFZ routing table has seen exponential growth at the rate of 1.3x every two years for the past few years and will continue to do so [30]. We use these models to extrapolate future DFZ routing table size. We consider two router families: Cisco's Cat6500 series with a supervisor 720-3B forwarding engine that can hold upto 239K IPv4 FIB entries and hence, was supposed to be phased out by mid-2007 [6], though some ISPs still continue to use them. We also consider Cisco's current generation of routers with a supervisor 720-3BXL engine that can hold 1M IPv4 FIB entries.

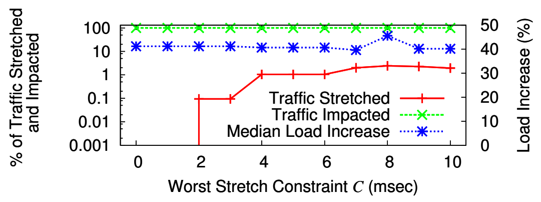


Fig. 4. Variation of the percentage of traffic stretched/impacted and load increase across routers with Worst Stretch Constraint (Uniform Allocation) and no popular prefixes.

For each of these router families, we calculate the year to which they would be able to cope with the growth in the DFZ routing table with the existing setup and with ViAggre. Table I shows the results for the Uniform Allocation.

For ViAggre, relaxing the worst-case stretch constraints reduces FIB size and hence, extends the router life. The table shows that if the DFZ routing table were to grow at the aforementioned exponential rate, ViAggre can extend the life of the previous generation of routers to 2018 with no stretch at all. We realise that estimates beyond a few years are not very relevant since the ISP would need to upgrade its routers for other reasons such as newer technologies and higher data rates anyway. However, with ViAggre, at least the ISP is not forced to upgrade due to growth in the routing table.

Figure 4 plots the impact of ViAggre on the ISP's traffic and router load. The percentage of traffic stretched is small, less than 1% for $C \leq 6$ ms. This shows that almost all the traffic is routed through an aggregation point in the same PoP as the ingress. However, the fact that no prefixes are considered popular implies that almost all the traffic follows a different router-level path as compared to the status quo. This shows up in figure 4 since the traffic impacted is ≈100% throughout. This, in turn, results in a median increase in load across the routers by ≈39%. In the next section we discuss how an ISP can use the skewed distribution of traffic to address the load concern while maintaining a small FIB on its routers.

4) *Popular Prefixes:* Past studies of ISP traffic patterns from as early as 1999 have observed that a small fraction of Internet prefixes carry a large majority of ISP traffic [10,13,34,38]. We used Netflow records collected across the routers of the same tier-1 ISP as in the last section for a period of two months (20th Nov'07 to 20th Jan'07) to generate per-prefix traffic statistics and observed that this pattern continues to the present day. The line labeled "Day-based, ISP-wide" in figure 5 plots the average fraction of the ISP's traffic destined to a given fraction of popular prefixes when the set of popular prefixes is calculated across the ISP on a daily basis. The figure shows that 1.5% of most popular prefixes carry 75.5% of the traffic while 5% of the prefixes carry 90.2% of the traffic.

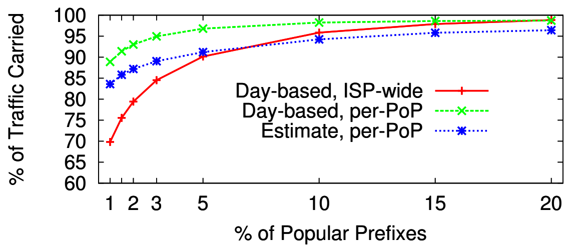
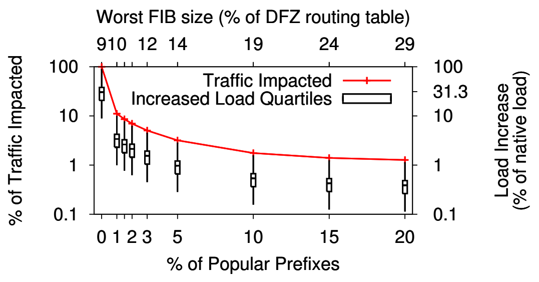


Fig. 5. Popular prefixes carry a large fraction of the ISP's traffic.

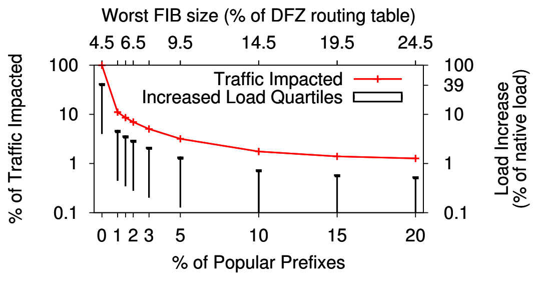
ViAggre exploits the notion of prefix popularity to reduce its impact on the ISP's traffic. However, the ISP's routers need not consider the same set of prefixes as popular; instead the popular prefixes can be chosen per-PoP or even per-router. We calculated the fraction of traffic carried by popular prefixes, when popularity is calculated separately for each PoP on a daily basis. This is plotted in the figure as "Day-based, per-PoP" and the fractions are even higher.

When using prefix popularity for router configuration, it would be preferable to be able to calculate the popular prefixes over a week, month, or even longer durations. The line labeled "Estimate, per-PoP" in the figure shows the amount of traffic carried to prefixes that are popular on a given day over the period of the next month, averaged over each day in the first month of our study. As can be seen, the estimate based on prefixes popular on any given day carries just a little less traffic as when the prefix popularity is calculated daily. This suggests that prefix popularity is stable enough for ViAggre configuration and the ISP can use the prefixes that are popular on a given day for a month or so. However, we admit that that these results are very preliminary and we need to study ISP traffic patterns over a longer period to substantiate the claims made above.

5) *Load Analysis:* We now consider the impact of a ViAggre deployment involving popular prefixes, i.e. the ISP populates the FIB on its routers with popular prefixes. Specifically, we focus on a deployment wherein the aggregation points are assigned to constrain Worst Stretch to 4ms, i.e. C = 4ms. Figure 6 shows how the traffic impacted and the quartiles for the load increase vary with the percentage of popular prefixes for both allocations. Note that using popular prefixes increases the router FIB size by the number of prefixes considered popular and thus, the upper X-axis in the figure shows the Worst FIB size. The large fraction of traffic carried by popular prefixes implies that both the traffic impacted and the load increase drops sharply even when a small fraction of prefixes is considered popular. For instance, with 2% popular prefixes in case of the uniform allocation (figure 6(b)), 7% of the traffic follows a different router-level path than before while the largest load increase is 3.1% of the original router load. With 5% popular prefixes, the largest load increase is 1.38%. Note that the more even distribution of prefixes across virtual prefixes in the uniform allocation results in a more even distribution of the excess traffic load across the ISP's routers – this shows up in the load quartiles being much smaller in figure 6(b) as compared to the ones in figure 6(a).



(a) With /7 allocation



(b) With Uniform allocation

Fig. 6. Variation of Traffic Impacted and Load Increase (0-25-50-75-100 percentile) with percentage of popular prefixes, C=4ms.

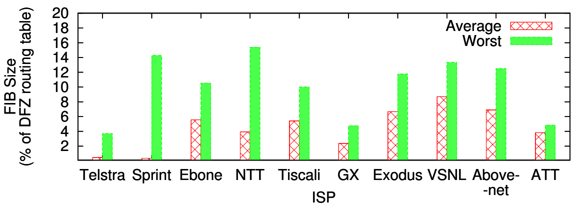


Fig. 7. FIB size for various ISPs using ViAggre.

C. Rocketfuel Study

We studied the topologies of 10 ISPs collected as part of the Rocketfuel project [37] to determine the FIB size savings that ViAggre would yield. Note that the fact we don't have traffic matrices for these ISPs implies that we cannot analyze the load increase across their routers. For each ISP, we used the assignment algorithm to determine the worst FIB size resulting from a ViAggre deployment where the worst stretch is limited to 5ms. Figure 7 shows that the worst FIB size is always less than 15% of the DFZ routing table. However, the Rocketfuel topologies are not complete and are missing routers. Hence, while the results presented here are encouraging, they should be treated as conservative estimates of the savings that ViAggre would yield for these ISPs.

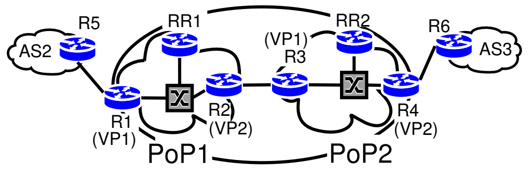


Fig. 8. WAIL topology used for our deployment. All routers in the figure are Cisco 7300s. RR1 and RR2 are route-reflectors and are not on the data path. Routers R1 and R3 aggregate virtual prefix VP1 while routers R2 and R4 aggregate VP2.

D. Discussion

The analysis above shows that ViAggre can significantly reduce FIB size. Most of the ISPs we studied are large tier-1 and tier-2 ISPs. However, smaller tier-2 and tier-3 ISPs are also part of the Internet DFZ. Actually, it is probably more important for such ISPs to be able to operate without needing to upgrade to the latest generation of routers. The fact that these ISPs have small PoPs might suggest that ViAggre would not be very beneficial. However, given their small size, the PoPs of these ISPs are typically geographically close to each other. Hence, it is possible to use the cumulative FIB space across routers of close-by PoPs to shrink the FIB substantially. And the use of popular prefixes ensures that the load increase and the traffic impact is still small. For instance, we analyzed router topology and routing table data from a regional tier-2 ISP (AS2497) and found that a ViAggre deployment with worst stretch less than 5ms can shrink the Worst FIB size to 14.2% of the routing table today.

Further, the fact that such ISPs are not tier-1 ISPs implies they are a customer of at least one other ISP. Hence, in many cases, the ISP could substantially shrink the FIB size on its routers by applying ViAggre to the small number of prefixes advertised by their customers and peers while using default routes for the rest of the prefixes.

V. Deployment

To verify the claim that ViAggre is a configuration-only solution, we deployed both ViAggre designs on a small network built on the WAIL testbed [3]. The test network is shown in figure 8 and represents an ISP with two PoPs. Each PoP has two Cisco 7301 routers and a route-reflector.[4] For the ViAggre deployment, we use two virtual prefixes: 0.0.0.0/1 (VP1) and 128.0.0.0/1 (VP2) with one router in each PoP serving as an aggregation point for each virtual prefix. Routers R1 and R4 have an external router connected to them and exchange routes using an eBGP peering. Specifically, router R5 advertises the entire DFZ routing table and this is, in turn, advertised through the ISP to router R6. We use OSPF for intra-domain routing. Beyond this, we configure the internal distribution of BGP routes according to the following three approaches:

1). **Status Quo.** The routers use a mesh of iBGP peerings to exchange the routes and hence, each router maintains the entire routing table.

2). **Design-I.** The routers still use a mesh of iBGP peerings to exchange routes. Beyond this, the routers are configured as follows:

– *Virtual Prefixes.* Routers advertise the virtual prefix they are aggregating to their iBGP peers.

– *FIB Suppression.* Each router only loads the routes that it is aggregating into its FIB. For instance, router R1 uses an `access-list` to specify that only routes belonging to VP1, the virtual prefix VP2 itself and any popular prefixes are loaded into the FIB. A snippet of this access-list is shown below.

```
! R5's IP address is 198.18.1.200
distance 255 198.18.1.200 0.0.0.0 1

! Don't mark anything inside 0.0.0.0/1
access-list 1 deny 0.0.0.0 128.255.255.255
! Don't mark virtual prefix 128.0.0.0/1
access-list 1 deny 0.0.0.0 128.0.0.0
! Don't mark popular prefix 122.1.1.0/24
access-list 1 deny 122.1.1.0 0.0.0.255
! ... other popular prefixes follow ...

! Mark the rest with admin distance 255
access-list 1 permit any
```

Here, the `distance` command sets the administrative distance of all prefixes that are accepted by `access-list 1` to "255" and these routes are not loaded by the router into its FIB.

– *LSPs to external routers.* We use MPLS for the tunnels between routers. To this effect, LDP [1] is enabled on the interfaces of all routers and establishes LSPs between the routers. Further, each edge router (R1 and R4) initiates a Downstream Unsolicited tunnel [1] for each external router connected to them to all their IGP neighbors using LDP. This ensures that packets to an external router are forwarded using MPLS to the edge router which strips the MPLS header before forwarding them onwards.

Given this setup and assuming no popular prefixes, routers R1 and R3 store 40.9% of today's routing table (107,943 prefixes that are in VP1) while R2 and R4 store 59.1%.

3). **Design-II.** The routers in a PoP peer with the route-reflector of the PoP and the route-reflectors peer with each other. External routers R1 and R6 are reconfigured to have eBGP peerings with RR1 and RR2 respectively. The advertisement of virtual prefixes and the MPLS configuration is the same as above. Beyond this, the route-reflectors are configured to ensure that they only advertise the prefixes being aggregated by a router to it. For instance, RR1 uses a `prefix-list` to ensure that only prefixes belonging to VP1, virtual prefix VP2 itself and popular prefixes are advertised to router R1. The structure of this prefix-list is similar to the access-list shown above. Finally, route-reflectors use a route-map on their eBGP peerings to change the BGP NEXT_HOP of the advertised routes to the edge router that the external peer is connected too. This ensures that the packets don't actually flow through the route-reflectors.

A. Configuration Overhead

A drawback of ViAggre being a "configuration-only" approach is the overhead that the extra configuration entails. The discussion above details the extra configuration that routers need to participate in ViAggre. Based on our deployment, the number of extra configuration lines needed for a router r to be configured according to design-I is given by $(r_{int} + r_{ext} + 2|A(r)| + |P_r| + 6)$ where r_{int} is the number of router interfaces, r_{ext} is the number of external routers r is peering with, $|A(r)|$ is the number of virtual prefixes r is aggregating and $|P_r|$ is the number of popular prefixes in r. Given the size of the routing table today, considering even a small fraction of prefixes as popular would cause the expression to be dominated by $|P_r|$ and can represent a large number of configuration lines.

However, quantifying the extra configuration lines does not paint the complete picture since given a list of popular prefixes, it is trivial to generate an access or prefix-list that would allow them. To illustrate this, we developed a configuration tool as part of our deployment effort. The tool is 334 line python script which takes as input a router's existing configuration file, the list of virtual prefixes, the router's (or representative) Netflow records and the percentage of prefixes to be considered popular. The tool extracts relevant information, such as information about the router's interfaces and peerings, from the configuration file. It also uses the Netflow records to determine the list of prefixes to be considered popular. Based on these extracted details, the script generates a configuration file that allows the router to operate as a ViAggre router. We have been using this tool for experiments with our deployment. Further, we use *clogin* [41] to automatically load the generated ViAggre configuration file onto the router. Thus, we can reconfigure our testbed from status quo operation to ViAggre operation (design-I and design II) in an automated fashion. While our tool is specific to the router vendor and other technologies in our deployment, its simplicity and our experience with it lends evidence to the argument that ViAggre offers a good trade-off between the configuration overhead and increased routing scalability.

B. Control-plane Overhead

Section IV evaluated the impact of ViAggre on the ISP's data plane. Beyond this, ViAggre uses control-plane mechanisms to divide the routing table amongst

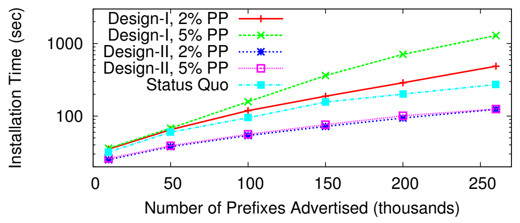
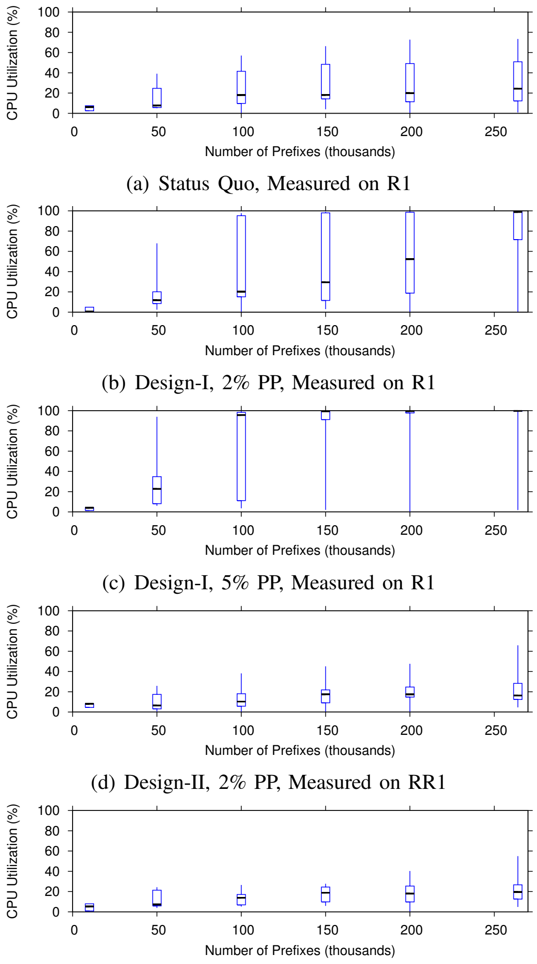


Fig. 9. Installation time with different approaches and varying fraction of Popular Prefixes (PP).



(a) Status Quo, Measured on R1

(b) Design-I, 2% PP, Measured on R1

(c) Design-I, 5% PP, Measured on R1

(d) Design-II, 2% PP, Measured on RR1

(e) Design-II, 5% PP, Measured on RR1

Fig. 10. CPU Utilization quartiles (0-25-50-75-100 percentile) for the three approaches and different fraction of Popular Prefixes (PP).

the ISP's routers – Design-I uses `access-lists` and Design-II uses `prefix-lists`. We quantify the performance overhead imposed by these mechanisms using our deployment. Specifically, we look at the impact of our designs on the propagation of routes through the ISP.

To this effect, we configured the internal distribution of BGP routes in our testbed according to the three approaches described above. External router R5 is configured to advertise a variable number of prefixes through its eBGP peering. We restart this peering on router R5 and measure the time it takes for the routes to be installed into the FIB of the ISP's routers and then advertised onwards; hereon we refer to this as the *installation time*. During this time, we also measure the CPU utilization on the routers. We achieve this by using a clogin script to execute the "*show process cpu*" command on each router every 5 seconds. The command gives the average CPU utilization of individual processes on the router over the past 5 seconds and we extract the CPU utilization of the "BGP router" process.

We measured the installation time and the CPU utilization for the three approaches. For status quo and design-I, we focus on the measurements for router R1 while for design-II, we focus on the measurements for route-reflector RR1. We also varied the number of popular prefixes. Here we present results with 2% and 5% popular prefixes. Figures 9 and 10 plot the installation time and the quartiles for the CPU utilization respectively.

Design-I Vs Status Quo. Figure 9 shows that the installation time with design-I is much higher than that with status quo. For instance, with status quo, the complete routing table is transferred and installed on router R1 in 273 seconds while with design-I and 2% popular prefixes, it takes 487 seconds. Further, the design-I installation time increases significantly as the number of popular prefixes increases. Finally, figures 10(b) and 10(c) show that design-I leads to a very high CPU load during the transfer which increases as more prefixes are considered popular. This results from the fact that access-lists with a large number of rules are very inefficient and would obviously be unacceptable for an ISP deploying ViAggre. We are currently exploring ways to achieve FIB suppression without the use of access-list.

Design-II Vs Status Quo. Figure 9 shows that the time to transfer, install and propagate routes with design-II is lesser than status quo. For instance, design-II with 2% popular prefixes leads to an installation time of 124 seconds for the entire routing table as compared to 273 seconds for status quo. Further, the installation time does not change much as the fraction of popular prefixes increases. Figures 10(d) and 10(e) show that the CPU utilization is low with median utilization being less than 20%. Note that the utilization shown for design-II was measured on route-reflector RR1 which has fewer peerings than router R1 in status quo. This explains the fact that the utilization with design-II is less than status quo. While preliminary, this experiment suggests that design-II can also help with route convergence within the ISP.

C. Failover

As detailed in section II-E, as long as alternate aggregation points exist, traffic in a ViAggre network is automatically re-routed upon failure of the aggregation point being used. We measured this failover time using our testbed. In the interest of space, we very briefly summarise the experiment here. We generated UDP traffic between PCs connected to routers R5 and R6 (figure 8) and then crashed the router being used as the aggregation point for the traffic. We measured the time it takes for traffic to be re-routed over 10 runs with each design. In both cases, the maximum observed failover time was 200 usecs. This shows that our designs ensure fast failover between aggregation points.

VI. Discussion

Pros. ViAggre can be *incrementally deployed* by an ISP since it does not require the cooperation of other ISPs and router vendors. The ISP does not need to change the structure of its PoPs or its topology. What's more, an ISP could experiment with ViAggre on a limited scale (a few virtual prefixes or a limited number of PoPs) to gain experience and comfort before expanding its deployment. None of the attributes in the BGP routes advertised by the ISP to its neighbors are changed due to the adoption of ViAggre. Also, the use of ViAggre by the ISP does not restrict its routing policies and route selection. Further, at least for design-II, control-plane processing is reduced. Finally, there is *incentive for deployment* since the ISP improves its own capability to deal with routing table growth.

Management Overhead. As detailed in section V-A, ViAggre requires extra configuration on the ISP's routers. Beyond this, the ISP needs to make a number of deployment decisions such as choosing the virtual prefixes to use, deciding where to keep aggregation points for each virtual prefix, and so on. Apart from such one-time or infrequent decisions, ViAggre may also influence very important aspects of the ISP's day-to-day operation such as maintenance, debugging, etc. All this leads to increased complexity and there is a cost associated with the extra management.

In section V-A we discussed a configuration tool that automates ViAggre configuration. It is difficult to speculate about actual costs and so we don't compare the increase in management costs against the cost of upgrading routers. While we hope that our tools will actually lead to cost savings for a ViAggre network, an ISP might just be inclined to adopt ViAggre because it breaks the dependency of various aspects of its operation on the size of the routing table. These aspects include its upgrade cycle, the per-byte forwarding cost, the per-byte forwarding power, etc.

Popular Prefixes. As mentioned earlier, ViAggre represents a trade-off between FIB shrinkage on one hand and increased router load and traffic stretch on the other. The fact that Internet traffic follows a power-law distribution makes this a very beneficial trade-off. This power-law observation has held up in measurement studies from 1999 [10] to 2008 (in this paper) and hence, Internet traffic has followed this distribution for at least the past nine years in spite of the rise in popularity of P2P and video streaming. We believe that, more likely than not, future Internet traffic will be power-law distributed and hence, ViAggre will represent a good trade-off for ISPs.

Other design points. The ViAggre proposal presented in this paper represents one point in the design space that we focussed on for the sake of concreteness. Alternative approaches based on the same idea include

– *Adding routers.* We have presented a couple of techniques that ensure that only a subset of the routing table is loaded into the FIB. Given this, an ISP could install "slow-fat routers", low-end devices (or maybe even a stack of software routers [16]) in each PoP that are only responsible for routing traffic destined to unpopular prefixes. These devices forward a low-volume of traffic, so it would be easier and cheaper to hold the entire routing table. The popular prefixes are loaded into existing routers. This approach can be seen as a variant of route caching and does away with a lot of deployment complexity. In fact, ViAggre may allow us to revisit route caching [24].

– *Router changes.* Routers can be changed to be ViAggre-aware and hence, make virtual prefixes first-class network objects. This would do away with a lot of the configuration complexity that ViAggre entails, ensure that ISPs get vendor support and hence, make it more palatable for ISPs. We, in cooperation with a router vendor, are exploring this option [15].

– *Clean-slate ViAggre.* The basic concept of virtual networks can be applied in an inter-domain fashion. The idea here is to use cooperation amongst ISPs to induce a routing hierarchy that is more aggregatable and hence, can accrue benefits beyond shrinking the router FIB. This involves virtual networks for individual virtual prefixes spanning domains such that even the RIB on a router only contains the prefixes it is responsible for. This would reduce both the router FIB and RIB and in general, improve routing scalability. We intend to study the merits and demerits of such an approach in future work.

VII. Related Work

A number of efforts have tried to directly tackle the routing scalability problem through clean-slate designs. One set of approaches try to reduce routing table size by dividing edge networks and ISPs into separate address spaces [7,11,29,32,40]. Our work resembles some aspects of CRIO [40] which uses virtual prefixes and tunneling to decouple network topology from addressing. However, CRIO requires adoption by all provider networks and like [7,11,29,32], requires a new mapping service to determine tunnel endpoints. APT [22] presents such a mapping service. Alternatively, it is possible to encode location information into IP addresses [8,14,18] and hence, reduce routing table size. Finally, an interesting set of approaches that trade-off stretch for routing table size are *Compact Routing* algorithms; see [26] for a survey of the area.

The use of tunnels has long been proposed as a routing scaling mechanism. VPN technologies such as BGP-MPLS VPNs [9] use tunnels to ensure that only PE routers need to keep the VPN routes. As a matter of fact, ISPs can and probably do use tunneling protocols such as MPLS and RSVP-TE to engineer a BGP-free core [35]. However, edge routers still need to keep the full FIB. With ViAggre, none of the routers on the data-path need to maintain the full FIB. Router vendors, if willing, can use a number of techniques to reduce the FIB size, including FIB compression [35] and route caching [35]. Forgetful routing [23] selectively discards alternative routes to reduce RIB size. [2] sketches the basic ViAggre idea. In recent work, Kim et. al. [25] use relaying, similar to ViAggre's use of aggregation points, to address the VPN routing scalability problem.

Over the years, several articles have documented the existing state of inter-domain routing and delineated requirements for the future [5,12,28]; see [12] for other routing related proposals. RCP [4] and 4D [17] argue for logical centralization of routing in ISPs to provide scalable internal route distribution and a simplified control plane respectively. We note that ViAggre fits well into these alternative routing models. As a matter of fact, the use of route-reflectors in design-II is similar in spirit to RCSs in [4] and DEs in [17].

VIII. Summary

This paper presents ViAggre, a technique that can be used by an ISP to substantially shrink the FIB on its routers and hence, extend the lifetime of its installed router base. The ISP may have to upgrade the routers for other reasons but at least it is not driven by DFZ growth over which it has no control. While it remains to be seen whether the use of automated tools to configure and manage large ViAggre deployments can offset the complexity concerns, we believe that the simplicity of the proposal and its possible short-term impact on routing scalability suggest that is an alternative worth considering.

Acknowledgements

This work was partially supported by NSF Grants 0338750 and 0626978 and a Cisco grant.

Notes

[1]Hereon, we follow the terminology used in [39] and use the term "routing table" to refer to the Forwarding Information Base or FIB, commonly also known as the forwarding table. The Routing Information Base is explicitly referred to as the RIB.

[2]All other attributes for the routes to a virtual prefix are the same and hence, the decision is based on the IGP metric to the aggregation points. Hence, "closest" means closest in terms of IGP metric.

[3]With design-II, a router chooses the aggregation point closest to the router's route-reflector in terms of IGP metrics and so a similar formulation works for the second design too.

[4]These are used only for the design-II deployment. We used both a Cisco 7301 and a Linux PC as a route-reflector.

References

[1] Andersson, L., Minei, I., and Thomas, B. RFC 5036 - LDP Specification, Jan 2006.

[2] Ballani, H., Francis, P., Cao, T., and Wang, J. ViAggre: Making Routers Last Longer! In *Proc. of Hotnets* (Oct 2008).

[3] Barford, P. Wisconsin Advanced Internet Laboratory (WAIL), Dec 2007. `http://wail.cs.wisc.edu/`.

[4] Caesar, M., Caldwell, D., Feamster, N., Rexford, J., Shaikh, A., and van der Merwe, J. Design and Implementation of a Routing Control Platform . In *Proc. of Symp. on Networked Systems Design and Implementation (NSDI)* (2005).

[5] Davies, E., and Doria, A. Analysis of Inter-Domain Routing Requirements and History. Internet Draft draft-irtf-routing-history-07.txt, Jan 2008.

[6] de Silva, S. 6500 FIB Forwarding Capacities. NANOG 39 meeting, 2007. `http://www.nanog.org/mtg-0702/presentations/fib-desilva.pdf`.

[7] Deering, S. The Map & Encap Scheme for scalable IPv4 routing with portable site prefixes, March 1996. `http://www.cs.ucla.edu/~lixia/map-n-encap.pdf`.

[8] Deering, S., and Hinden, R. IPv6 Metro Addressing. Internet Draft draft-deering-ipv6-metro-addr-00.txt, Mar 1996.

[9] E. Rosen and Y. Rekhter. RFC 2547 - BGP/MPLS VPNs, Mar 1999.

[10] Fang, W., and Peterson, L. Inter-As traffic patterns and their implications. In *Proc. of Global Internet* (1999).

[11] Farinacci, D., Fuller, V., Oran, D., and Meyer, D. Locator/ID Separation Protocol (LISP). Internet Draft draft-farinacci-lisp-02.txt, July 2007.

[12] Feamster, N., Balakrishnan, H., and Rexford, J. Some Foundational Problems in Interdomain Routing. In *Proc. of Workshop on Hot Topics in Networks (HotNets-III)* (2004).

[13] Feldmann, A., Greenberg, A., Lund, C., Reingold, N., Rexford, J., and True, F. Deriving traffic demands for operational IP networks: methodology and experience. *IEEE/ACM Trans. Netw. 9*, 3 (2001).

[14] Francis, P. Comparision of geographical and provider-rooted Internet addressing. *Computer Networks and ISDN Systems 27*, 3 (1994).

[15] Francis, P., Xu, X., and Ballani, H. FIB Suppression with Virtual Aggregation and Default Routes. Internet Draft draft-francis-idr-intra-va-01.txt, Sep 2008.

[16] Gillian, B. VYATTA: Linux IP Routers, Dec 2007. `http://freedomhec.pbwiki.com/f/linux_ip_routers.pdf`.

[17] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Meyers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A clean slate 4D approach to network control and management. *ACM SIGCOMM Computer Communications Review* (October 2005).

[18] Hain, T. An IPv6 Provider-Independent Global Unicast Address Format. Internet Draft draft-hain-ipv6-PI-addr-02.txt, Sep 2002.

[19] Hughes, D., Dec 2004. PACNOG list posting `http://mailman.apnic.net/mailing-lists/pacnog/archive/2004/12/msg00000.html`.

[20] Huston, G. BGP Reports, Dec 2007. `http://bgp.potaroo.net/`.

[21] Huston, G., and Armitage, G. Projecting Future IPv4 Router Requirements from Trends in Dynamic BGP Behaviour. In *Proc. of ATNAC* (2006).

[22] Jen, D., Meisel, M., Massey, D., Wang, L., Zhang, B., and Zhang, L. APT: A Practical Transit Mapping Service. Internet Draft draft-jen-apt-01.txt, Nov 2007.

[23] Karpilovsky, E., and Rexford, J. Using forgetful routing to control BGP table size. In *Proc. of CoNext* (2006).

[24] Kim, C., Caesar, M., Gerber, A., and Rexford, J. Revisiting route caching: The world should be flat. In *Proc. of PAM* (2009).

[25] Kim, C., Gerber, A., Lund, C., Pei, D., and Sen, S. Scalable VPN Routing via Relaying. In *Proc. of ACM SIGMETRICS* (2008).

[26] Krioukov, D., and kc claffy. Toward Compact Interdomain Routing, Aug 2005. `http://arxiv.org/abs/cs/0508021`.

[27] Li, T. Router Scalability and Moore's Law, Oct 2006. `http://www.iab.org/about/workshops/routingandaddressing/Router_Scalability.pdf`.

[28] Mao, Z. M. Routing Research Issues. In *Proc. of WIRED* (2003).

[29] Massey, D., Wang, L., Zhang, B., and Zhang, L. A Proposal for Scalable Internet Routing & Addressing. Internet Draft draft-wang-ietf-efit-00, Feb 2007.

[30] Meyer, D., Zhang, L., and Fall, K. Report from the IAB Workshop on Routing and Addressing. Internet Draft draft-iab-raws-report-02.txt, Apr 2007.

[31] Narten, T. Routing and Addressing Problem Statement. Internet Draft draft-narten-radir-problem-statement-02.txt, Apr 2008.

[32] O'Dell, M. GSE–An Alternate Addressing Architecture for IPv6. Internet Draft draft-ietf-ipngwg-gseaddr-00.txt, Feb 1997.

[33] Ravi, R., and Sinha, A. Multicommodity facility location. In *Proc. of ACM-SIAM SODA* (2004).

[34] Rexford, J., Wang, J., Xiao, Z., and Zhang, Y. BGP routing stability of popular destinations. In *Proc. of Internet Measurment Workshop* (2002).

[35] Scudder, J. Router Scaling Trends. APRICOT Meeting, 2007. `http://submission.apricot.net/chatter07/slides/future_of_routing`.

[36] Spring, N., Mahajan, R., and Anderson, T. Quantifying the Causes of Path Inflation. In *Proc. of ACM SIGCOMM* (2003).

[37] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP topologies with Rocketfuel. In *Proc. of ACM SIGCOMM* (2002).

[38] Taft, N., Bhattacharyya, S., Jetcheva, J., and Diot, C. Understanding traffic dynamics at a backbone PoP. In *Proc. of Scalability and Traffic Control and IP Networks SPIE ITCOM* (2001).

[39] Y. Rekhter and T. Li and S. Hares, Ed. RFC 4271 - A Border Gateway Protocol 4 (BGP-4), Jan 2006.

[40] Zhang, X., Francis, P., Wang, J., and Yoshida, K. Scaling Global IP Routing with the Core Router-Integrated Overlay. In *Proc. of ICNP* (2006).

[41] clogin Manual Page, Oct 2008. `http://www.shrubbery.net/rancid/man/elogin.1.html`.

[42] Foundry Router Reference, Jul 2008. `http://www.foundrynetworks.co.jp/services/documentation/srcli/BGP_cmds.html`.

[43] JunOS Route Preferences, Jul 2008. `http://www.juniper.net/techpubs/software/junos/junos60/swconfig60-routing/html/protocols-overview4.html`.

Symbiotic Relationships in Internet Routing Overlays

Cristian Lumezanu, Randy Baden, Dave Levin, Neil Spring, Bobby Bhattacharjee
University of Maryland

Abstract

We propose to construct routing overlay networks using the following principle: that overlay edges should be based on mutual advantage between pairs of hosts. Upon this principle, we design, implement, and evaluate PeerWise, a latency-reducing overlay network. To show the feasibility of PeerWise, we must show first that mutual advantage exists in the Internet: perhaps contrary to expectation, that there are not only "haves" and "have nots" of low-latency connectivity. Second, we must provide a scalable means of finding promising edges and overlay routes; we seek embedding error in network coordinates to expose both shorter-than-default "detour" routes and longer-than-expected default routes.

We evaluate the cost of limiting PeerWise to mutually advantageous links, then build the intelligent components that put PeerWise into practice. We design and evaluate "virtual" network coordinates for destinations not participating in the overlay, neighbor selection algorithms to find promising relays, and relay selection algorithms to choose the neighbor to traverse for a good detour. Finally, we show that PeerWise is practical through a wide-area deployment and evaluation.

1 Introduction

We propose *mutual advantage* as a principle for the construction of routing overlay networks: overlay edges should exist only between hosts that benefit from each other's resources or position in the network. Hosts negotiate connections based strictly on mutual advantage, and overlay paths follow only these connections.

Several distributed protocols and applications use *mutual advantage* as part of their design. BitTorrent [5] peers that download the same file trade blocks the other is missing. In backup systems [7], nodes store replicas of files for each other. Autonomous systems in the Internet negotiate peer-to-peer agreements to provide low-cost connectivity to each other's customers [9].

Bringing *mutual advantage* into the design of routing overlays has several benefits. First, mutual advantage induces better cooperation among nodes. Incentives to participate become simpler, and long-lived, fair connections appear. Building systems grounded in incentives for cooperation makes them robust to misbehavior and selfishness [23, 29]. Second, users could freely discriminate among the connections that they allow and would have the ability to explicitly say how much service they want to contribute. Finally, mutual advantage avoids the tragedy of the commons in routing overlays, when only a few, well-connected nodes provide transit. It keeps the trades of connectivity fair, in contrast to file-sharing where universities are net providers of content [27].

In this paper, we present the design, implementation, and evaluation of PeerWise, a latency-reducing routing overlay based on *mutual advantage*. PeerWise scalably discovers detour routes: "indirect" one-hop paths that have lower round trip latency than the "direct" path.

In a previous paper [17], we presented ideas that support a mutually advantageous latency-reducing overlay: that mutual advantage is common in the context of Internet latencies and that embedding error in network coordinate systems, such as Vivaldi [8] or GNP [20], could be used to scalably discover detours. However, we did not evaluate the potential and limitations of mutual advantage, nor did we design or implement a system to take advantage of the existing detour routes. In this work, we show that a mutually advantageous latency-reducing overlay is feasible and efficient, and that detours toward popular destinations are available. We design, implement, and evaluate a system that finds these detours.

We describe our contributions next.

First, we use a measurement-driven simulation to show the potential of PeerWise (§4). We collect two latency data sets to find what fraction of detours exist subject to the mutual advantage requirement and, independently, can be found by embedding error. The mutual advantage requirement reduces the number of destinations reachable via detour by approximately half, yet even popular websites, using content distribution services such as Akamai, are reachable by PeerWise-found detours. Only 5% of potential detours are missed by embedding error.

We next describe the design of PeerWise in two main parts: mechanism (§5) and policies (§6). We implement a *virtual network coordinate* approach to find coordinates for the destinations that do not participate directly in the overlay. *Neighbor tracking* determines the set of nodes that are more likely to offer detours by remembering those neighbors with high embedding error in the coordinate space. *Pairwise negotiation* establishes connections promising mutual benefit while the *maintenance* component ensures that each node receives approximately as much benefit as it provides.

The second part of the design focuses on the decisions that each PeerWise node makes. We evaluate neighbor selection and relay selection algorithms. We show that coordinates can be used to choose among detours. Our environment is quite different from previous work on latency prediction using coordinates. Instead of focusing on source-to-destination, we must choose a source-to-relay-to-destination path based on a relay coordinate known to have high embedding error and a destination coordinate that may be stale or inaccurate.

Finally, we describe the implementation of PeerWise and its evaluation on PlanetLab (§7). We show that PeerWise nodes find detours to popular destinations, that these detours are stable, and that they offer significant latency reductions. Most detours last for a long time and are obtained using only one mutually advantageous peering. We then show how PeerWise detours translate into real life and whether user applications can benefit.

2 Related Work

Routing overlays, such as RON [2], Detour [28], SOSR [11], and OverQoS [31], promise to provide more-reliable or faster paths through the Internet. They forward packets along links in self-constructed meshes and make routing decisions without support from routers or ASes. RON [2] builds a fully connected mesh and monitors all edges. When the direct path between two nodes fails or has performance problems, communication is established through the other overlay nodes. Nakao *et al.* [19, 18] use static AS-level topology and geographical distance information to eliminate redundant overlay edges and improve scalability. Gummadi *et al.* show that all-to-all measurements are not necessary to find reliable paths: routing through a randomly chosen intermediary node is enough [11]. Similarly, we show that faster-than-default paths can be discovered with limited information: network coordinates and latencies to a few other nodes are sufficient.

Various file-swarming systems [5, 15, 30] apply tit-for-tat-like schemes to induce cooperation among peers. Tit-for-tat applies when there is a *mutual interest* among peers, which is common in file swarming; for any pair of peers, one may have blocks the other does not. We show that, perhaps surprisingly, mutual interest is common in low-latency routing in the Internet as well, and that locating nodes of mutual interest can be done in a decentralized fashion.

The requirements imposed by PeerWise on who can connect to whom are reminiscent of the bilateral connection game (BCG) [6], a special case of network formation game. In BCG, a link between two nodes is established only with the consent of both nodes, similarly to PeerWise. However, nodes construct links that minimize the cost of reaching other participating nodes, whereas in PeerWise, nodes create peerings that offer detours to destinations that do not necessarily participate.

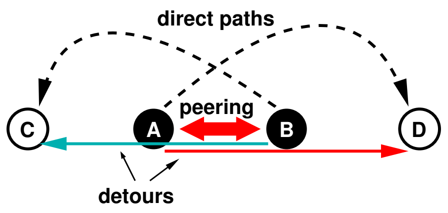


Figure 1: Obtaining faster paths with PeerWise: A discovers a detour to D through B; B also finds that it can reach C faster if it traverses A; A and B create a mutually advantageous peering that they both use to reach their destinations more quickly.

3 PeerWise Philosophy

In this section, we present an overview of PeerWise. We outline the two properties on which PeerWise is based: that mutual advantage is common in the Internet latency space and that network coordinate systems can help indicate detour routes. A previous paper [17] describes these properties in more detail. We then argue that it has the potential to be applied to a wide range of applications.

3.1 Overview

The key idea of PeerWise is that two nodes can cooperate to obtain faster end-to-end paths without either being compelled to offer more service than they receive. Peers negotiate and establish pairwise connections to each other based strictly on mutual advantage. Figure 1 shows an example. The default Internet path between two nodes is the *direct path*. A shorter, alternate path having one intermediate hop is a *detour*, using terminology from Detour [28]. Node A discovers a faster path to D via B. However, B will not help A unless A provides a detour in exchange. Since there is a shorter path from B to C going through A, A and B can help each other communicate faster with their intended destinations.

Mutual Advantage Each participant in overlay networks contributes resources in exchange for the resources of others. Unfortunately, free access and unrestricted demand may lead to over-utilization of certain resources, especially those of well-provisioned nodes. This tragedy of the commons occurs because the benefits of using common resources accrue to individuals, while the costs of exploitation are shared by the resource providers.

Pairwise peerings based on mutual benefit offer users an effective way to resolve the tragedy of the commons, as they can freely discriminate among the connections they allow. However, such decentralized policy may be costly: if nodes accept only peerings that are mutually

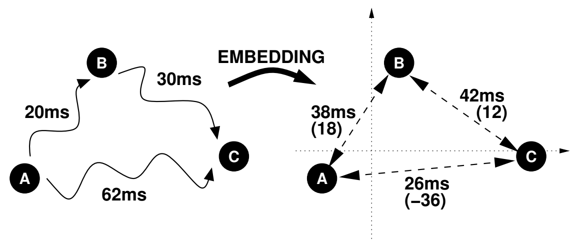


Figure 2: Embedding three points that form a TIV into a metric space introduces inaccuracies. The numbers in parentheses represent embedding errors.

advantageous, but mutual advantage is rare, the benefit of the overlay is lost. In Section 4, we show that mutual benefit is common and that a majority of nodes are in a position to both provide and receive service.

Network Coordinates Embedding Error Measuring and distributing the all-pairs latencies required to find detours would limit the scalability of a latency-reducing overlay. Instead, PeerWise detects triangle inequality violations (TIVs) and uses them to predict good detours.

Three nodes in the Internet form a TIV when the RTT between two of them (the *long side* of the TIV) is greater than the sum of the RTTs to the third node (the *short sides* of the TIV). The left side of Figure 2 shows an example TIV. Pairs of nodes that are long sides in TIVs may benefit from detours; pairs that are short sides may be part of detours.

To find TIVs scalably, PeerWise uses network coordinates. A network coordinate system associates nodes with points in a metric space such that the distance between the points estimates the real latency between nodes. Since TIVs are not allowed in metric spaces by definition, this embedding may result in high errors on the edges of the triangle (see Figure 2). The error for the long side of the TIV will be very negative, or the error for the sum of short sides of the TIV will be very positive. Thus, a pair of nodes with a negative estimation error has a higher chance of benefiting from a shorter path; conversely, when the nodes have a large estimation error between them, they are more likely to be part of a shorter path for another node.

3.2 Where does PeerWise apply?

We expect PeerWise has the most utility for latency-sensitive traffic such as HTTP HEAD requests that check for updates to a cached file before rendering, XML-RPC requests for rapid updates of existing content such as train status or sports scores, voice traffic relayed to bypass firewalls, and online games such as first-person shooters, whose playability hinges on low-latency updates among players [3]. Existing overlay networks could benefit from using PeerWise as a latency-reducing substrate by guiding PeerWise's neighbor- and relay-selection algorithms to better suit the application's needs.

Because PeerWise focuses on reducing latency, it can find and use the low-latency paths that may not support high-bandwidth use—that is, the low-latency paths that the default routing, likely tuned for high-bandwidth, misses. Going through a peer is likely to traverse another access link that might have low bandwidth. This means that bandwidth-intensive applications, such as video streaming, are unlikely to benefit from latency reduction with PeerWise.

4 Limitations of Mutual Advantage

We assess the potential performance of a mutually advantageous latency-reducing overlay. Because we restrict detour paths to mutually advantageous peerings, we would not expect PeerWise to find the shortest detours or find detours to all destinations. We simulate using two latency data sets to show that nodes can find shorter paths to the majority of destinations for which a shorter detour exists, despite the requirement of mutual advantage. We find that mutually advantageous detours exist even for popular destinations hosted on many prefixes.

4.1 Collected Data Sets

We collected two real-world latency data sets and computed all one-hop detours between each pair of nodes.

PW-King Data Set The first data set, PW-King, contains RTTs between 1,953 DNS servers of hosts in the Gnutella network. The list of hosts was gathered by Dabek *et al.* for the Vivaldi [8] project. We use King [10] to measure all-to-all latencies between the servers. King uses recursive DNS queries to estimate the propagation delay between two hosts as the delay between their authoritative name servers. The 1,953 servers were chosen for being in the same subnet as their hosts so that better-connected DNS servers would not influence the estimates of inter-client latencies. For each pair of nodes, we kept the median of all latencies measured at random intervals for a week in February 2008. Of the 1,953 servers, we removed 238 that appeared to experience high load during the measurement, as described by Dabek *et al.* [8]. A heavily-loaded DNS server can cause King to underestimate latencies to other nodes, which can lead to false triangle inequality violations.

Popular Destinations Data Set The second data set, PL-Dest, contains RTTs from 389 PlanetLab nodes to 500 popular web servers, measured in January 2008. We selected the servers based on a ranking by the Alexa Internet Company [1] using expected and measured client access. For faster content delivery, many of the websites have multiple IP addresses; users in different geographic regions see different IPs for the same server. To gather the IP addresses associated with a website, as visible from PlanetLab, we performed DNS lookups on each of the 500 names from the 389 PlanetLab nodes. We obtained 2932 distinct IP addresses in 796 /24 prefixes. We

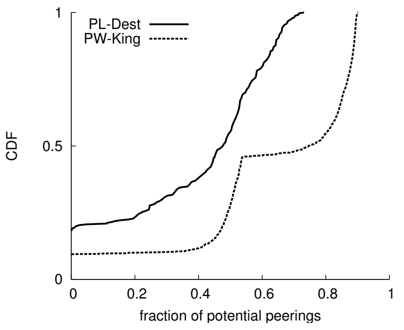


Figure 3: Distribution of the fraction of nodes with which a potential peering exists. In PL-Dest, 18% of nodes have no potential peerings; in PW-King, 50% of nodes have potential peerings with at least 75% of the other nodes.

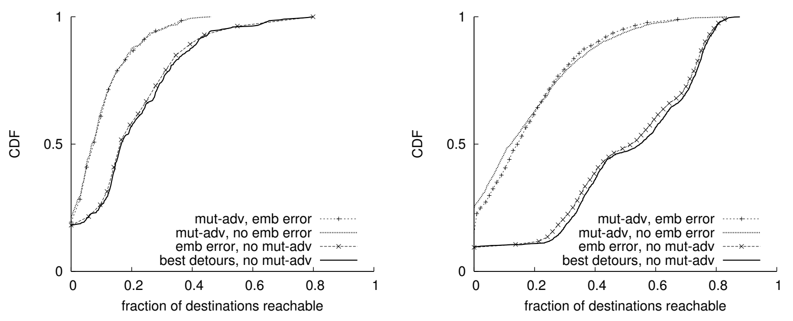


Figure 4: Distribution of the fraction of destinations reachable through mutually advantageous peerings for PL-Dest data set (left), and PW-King data set (right). In PL-Dest, few destinations can be reached by detour at all, some sources need no detours, and approximately half of the detours that could be used are lost by the mutual advantage restriction. In PW-King, all nodes have many detours available, and mutual advantage is less costly. In both, embedding error finds nearly all detours.

probed each prefix and each PlanetLab node from every PlanetLab node at random times over a week. We used the median RTT values to represent the link.

The latency collection process can produce incorrect data that may bias our results. We removed 52 servers from the final data set because we could not measure any RTT to them. Further, several PlanetLab nodes had very low latencies (< 1 ms) to most destinations. These latencies are likely caused by connection-tracking firewalls or "transparent" proxies near the PlanetLab nodes that generate spoofed responses as if from the destination. We removed those nodes from the data set since they would artificially overstate the potential of PeerWise. Our final latency matrix contains RTT values from 325 PlanetLab nodes to 718 prefixes corresponding to 448 websites.

The PW-King and PL-Dest data sets illustrate two scenarios in which PeerWise can be useful. Latency reduction on PW-King shows the potential benefit to applications a set of peers may run, such as a distributed multi-player network game or VoIP application. On PL-Dest, reduced latency shows benefit for users accessing popular servers that would not participate in PeerWise.

4.2 Methodology

We built a simulation prototype of PeerWise to study how well it finds detours with mutual advantage and embedding error. To find network coordinates for nodes, we use Vivaldi [8]. We allow each node to communicate with all other nodes, to better study mutual advantage in isolation. When requesting detours for its destinations, a node starts with the neighbor that has the highest embedding error [17]. We evaluate alternative relay selection methods in Section 6.2.

For each pair of nodes in our data sets, we find all one-hop detours. We define a *good detour* as a detour that provides at least 10 ms and 10% latency reduction over the direct path. We consider only good detours. This cutoff helps avoid impractical or dubious detours due to measurement error. In the PL-Dest data set, we may find detours by server name: The detour path may end at a different IP address associated with the same name.

4.3 Mutual Advantage

How much mutual advantage exists in our data sets? We define a *potential peering* to exist between two nodes that can provide a detour to each other, for at least one destination, as between A and B in Figure 1. The number of potential peerings for a node represents the number of neighbors with which the node can construct mutually advantageous peerings. In Figure 3, we show a cumulative distribution of the fraction of nodes for which a potential peering exists. Each point represents a node, and its placement on the x-axis what fraction of the other nodes it shares a potential peering with. At least 50% of the nodes in either data set have have potential peerings with at least 50% of the rest of the nodes. The figure also shows that there is more mutual advantage in the PW-King data set than in PL-Dest.

Next, we show that mutual advantage sacrifices few detours. We study the fraction of destinations that each node can reach more quickly via mutually advantageous peerings in Figure 4. Each graph considers four cases to isolate the two main potential performance sacrifices: the requirement of mutual advantage (that could make detours unavailable) and relay choice by positive embedding error (that might not find them despite being possible). The solid line represents an unconstrained detour overlay. Considering mutual advantage eliminates over half of the potential destinations for many nodes. For some, mutual advantage eliminates *all* detours; trivially, these are the nodes that cannot provide service to others. Choosing among either set (constrained to mutual

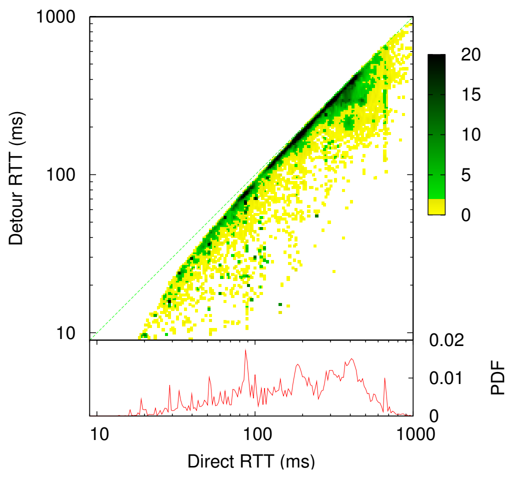


Figure 5: PL-Dest: When a detour exists, density plot of detour path RTT versus the direct path RTT (top, the color of each point represents the number of TIVs with the corresponding direct and detour RTTs), and PDF of direct path RTTs (bottom).

advantage or not) via embedding error between source and relay sacrifices very few detours (the corresponding lines are almost indistinguishable from each other in Figure 4). Mutual advantage does not impact the latency reduction to the destinations that are still reachable: only at most 12% of the median latency reduction is lost due to the requirement of mutual advantage.

4.4 Detours to Nearby Destinations

The destinations in PL-Dest include both regionally and globally popular websites. We expect that a regional website serves its pages from within the region of interest, so the direct path latencies to the destination from PeerWise nodes in that region should be small. Since the PlanetLab nodes are globally diverse, some "detours" may be for destinations unpopular in that node's region. For example, detours to popular websites in China may be less useful for nodes in Europe or North America. In Figure 5, we show that latency reduction is not limited to distant destinations. Because our rule to define a "good" detour requires at least 10 ms of reduction, few very short paths are featured. However, mutually-advantageous detours are found for direct paths too short to cross the Atlantic or Pacific oceans (< 100 ms).

4.5 Multiple-IP Websites

For faster content delivery, around 20% of the popular websites in the PL-Dest data set are served from geographically distributed locations. User requests are transparently directed to the geographically (or administratively) nearest IP address.

Using the PL-Dest data set, we compute how many nodes can find detours to each of the 448 websites and plot it against the total number of /24 prefixes of each website. Figure 6 presents the results. Each point in the plot is associated with one server name. Most websites with IP addresses in at least two prefixes can be reached faster from at least one PlanetLab node. We divide the plot into six regions and describe each in the accompanying table.

Figure 6 shows that PeerWise has the potential to be effective in reducing latency to most popular websites, even when they employ other latency-reducing techniques such as mirroring or DNS redirection.

4.6 Simulation Limitations

First, our pairwise peerings are established expecting that each destination will be accessed as often as any other. Clearly, not all destinations are equally popular, but we cannot estimate how often peers will use the peering. Our evaluation might favor VoIP applications where the endpoints are well distributed and no endpoint is orders of magnitude more popular than the others. In Section 7, we experiment with different access patterns, including random and zipf, to try to apply likely relative popularity models to traffic.

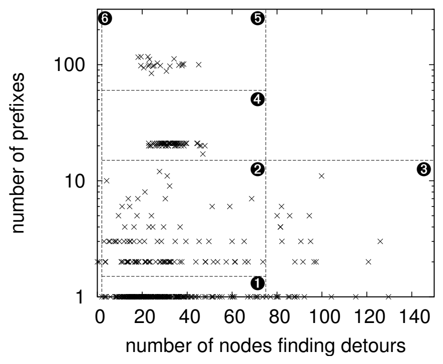
Second, the latencies between DNS servers or PlanetLab nodes may underestimate the latencies between end-hosts in the Internet. Although the latency matrix between DNS servers and PlanetLab hosts may represent the locations of hosts in the coordinate space, these data sets may not represent the latencies seen by such hosts.

Third, using PlanetLab nodes to reach popular destinations may raise questions about the validity of our evaluation. Connecting to a commercial site via a PlanetLab relay may reveal detours that would not be discovered had the relay been on the commercial network. However, Abilene and NLR, research networks that are part of Internet2, use wavelengths on fiber leased from other providers along rights-of-way shared with commercial networks. We believe that this sharing prevents research networks from providing an unfair advantage in latency reduction. We have even observed detours between PlanetLab nodes—routing within the academic network is not so latency-optimal as to prevent detours.

Finally, we do not model the bandwidth of the connection. Even though mutual latency reductions lead to a pairwise peering, limited bandwidth may prevent it from helping. As described in Section 3, we expect to use PeerWise only with latency-sensitive applications that do not require high bandwidth.

5 Design, Part I: Mechanisms

We present next the design of the PeerWise routing overlay network. In this section, we focus on the key features of PeerWise: detour detection using network coordinates for scalability, neighbor tracking for improving efficiency, and pairwise negotiations for fairness. In



	Region	Sites	Notes
❶	$p = 1$ $0 < c \leq 75$	239 (53%)	Under-provisioned, PeerWise expected to be useful.
❷	$1 < p \leq 12$ $0 < c \leq 75$	95 (21%)	Have many prefixes, PeerWise finds detours.
❸	$0 < p \leq 12$ $c > 75$	29 (6%)	Regional websites, most from China, many nodes find detours to them; perhaps not designed for global access.
❹	$12 < p \leq 60$ $0 < c \leq 100$	62 (14%)	All www.google.* websites; which IP address is chosen depends on the source, not the country suffix.
❺	$p > 60$ $0 < c \leq 75$	21 (6%)	Akamai-type destinations; finding a detour to any replica hosting center provides a detour to all sites hosted there.
❻	$p \geq 1$ $c = 0$	2 (0%)	PeerWise finds no detours to these multi-prefix sites; includes www.it168.com and www.sohu.com.

Figure 6: Detours to mirrored websites: The figure presents number of nodes that find detours (c) versus number of prefixes for each website (p). The table describes the six regions in the figure.

Section 6, we describe and evaluate the policies of each PeerWise node. We present the implementation and evaluation details in Section 7.

5.1 Virtual Network Coordinates

Every PeerWise node must compute its own network coordinate before searching for detours. We use Vivaldi [8] for network coordinates. Every node maintains a set of neighbors that it probes periodically. It uses the round trip time and the network coordinate of these neighbors to update its own coordinate. After each probe, the node computes the coordinate that minimizes the squared estimation error to all of its neighbors. To help the system converge quickly, nodes with uncertain coordinates can move farther with each measurement. Figure 7(a) shows the coordinate computation process.

A node in PeerWise must learn the coordinates of destinations to discover long or short sides of a TIV. However, if a destination is not participating in the overlay, it will not provide its own network coordinate. We therefore extend Vivaldi to allow a node to compute a virtual network coordinate for any non-participating host. We refer to non-participating Internet nodes as hosts and to PeerWise participants simply as nodes.

To generate virtual network coordinates for non-participating hosts in Vivaldi, a participating node chooses to become temporarily responsible for that host. The node runs Vivaldi on behalf of the host with one minor adjustment. Since the host is not participating in the system, it cannot manage its own neighbor set or actively gather the round trip times needed to compute the coordinate. Instead, the participating node uses its own neighbor set as the neighbor set for the host, and requests that those neighbors measure the latency to the host, as shown in Figure 7(b). Our extensions are similar to those recently described by Ledlie *et al.* [13].

Requiring all nodes to compute virtual coordinates for all non-participating destinations would limit the scalability of PeerWise. We include a gossip mechanism to disseminate the calculated coordinates throughout the

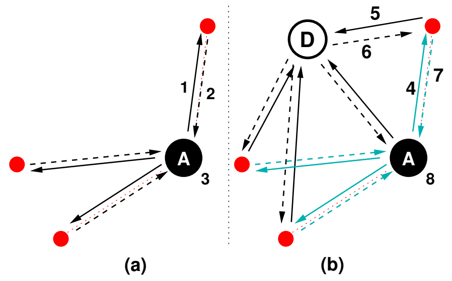


Figure 7: (a) *Computing network coordinates for a PeerWise node*: A measures RTT to its neighbors and asks for their coordinates (1); after it receives the replies (2) it computes the coordinate that minimizes the squared estimation error (3); (b) *Computing network coordinates for a non-PeerWise node D*: A asks each of its neighbors (4) to measure RTTs to D (5,6); after it receives the replies from the neighbors (7), A runs the network coordinate algorithm on behalf of D (8).

system. At fixed intervals (10s in our experiments), each node picks one of its neighbors at random, then selects a random destination and sends to the neighbor the IP address, name and virtual coordinate of the destination.

A node decides to take responsibility for a destination to which it wants to find a detour when the destination's coordinate does not yet exist, becomes too old (1 day in our experiments), or becomes unstable (where stability depends on the embedding error to other nodes). Any node can generate coordinates independently; this decentralization may allow simultaneous, redundant work. Rather than try to enforce a single consistent view of the coordinate, we allow any of these coordinates to be considered valid estimates. When a node receives a new virtual coordinate through the gossip protocol, it uses that new coordinate only if it is more stable and it was updated by the node responsible for it.

Virtual network coordinates are useful if a host is popular. If the host is not popular, a node trying to discover a detour to that host will need to compute its virtual coordinate. Since this requires that the node's neighbors measure the round trip time to the host, the node would know all three sides of the triangle, so it would trivially discover TIVs. However, if the node knows the virtual coordinate of a host already (because the host is popular and its coordinate has been gossiped), it will only know the two adjacent sides of the triangle, and it will be able to make predictions about the third side between the neighbor and the destination. We evaluate these predictions in Section 6.3.

5.2 Neighbor Tracking

The success of our protocol depends on the ability of nodes to find other nodes to establish pairwise peerings. There are many possible relays for a node, any of which may have high embedding error with respect to the node. Recall that high embedding error for a pair of nodes indicates a higher probability that the pair is part of a detour. We use neighbor tracking to find the nodes that are more likely to offer detours. With neighbor tracking, a PeerWise node remembers extra neighbors and learns about good potential relays from its neighbors or from nearby (in latency) nodes. The *neighbors* in this section are not *relays*; they are only candidates for becoming so.

When joining PeerWise, a node bootstraps its potential neighbor set from a known PeerWise node and uses it to compute its network coordinate. Once the network coordinate is stable, the node asks its neighbors about their own neighbors, remembering those nodes with high embedding error. For example, in Figure 8, A asks for the neighbor set of B, formed of B_1, B_2 and B_3. Node A then computes the embedding error from itself to each of B_1, B_2 and B_3 and adds those nodes to which the error is most positive to its neighbor list. These nodes are the most likely to form a short side of a TIV with A.

For scalability, we limit the number of neighbors of each node. Neighbors with higher potential to offer the best detours replace less-efficient neighbors. We consider and evaluate different methods for ranking potential neighbors in Section 6.1. Because PeerWise allows a node to exchange information about neighbors with neighbors, we expect each node to have ample choices.

5.3 Pairwise Negotiation

PeerWise nodes negotiate with their neighbors to request or advertise alternate routes. As discussed in Section 3, a detour to a destination is likely to exist if the estimated distance to the destination is much smaller than the measured latency. In this case, a node asks its neighbors with high embedding errors whether they can offer a faster path (Figure 9(c)). Nodes are not limited to this simple strategy. In Section 6.2, we evaluate different policies for choosing relays and deciding whether to request detours for a destination.

Actively requesting detours may be inefficient, especially if the connection to the destination is short-lived.

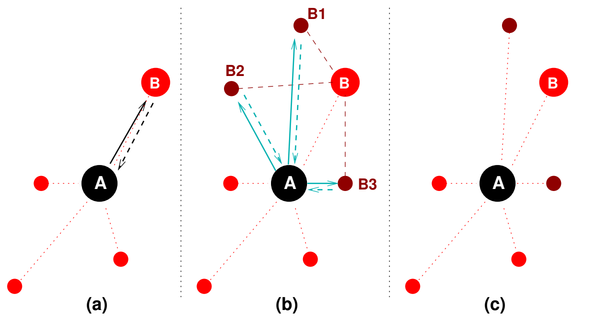


Figure 8: Neighbor Tracking. (a) A chooses the neighbor to which it has the highest embedding error and requests its neighbor set; (b) A measures RTTs to each of the nodes received from B; (c) A adds to its neighbor set those nodes to which it has a positive embedding error.

In addition, the time to find a detour may dominate the latency reduction achieved. To encourage fast detour discovery, PeerWise nodes also proactively advertise paths to popular destinations. For example, in Figure 9(d), node A observes that the link to node D, which may or may not be running PeerWise, has a high estimation error. This means that AD may be a short side in a TIV. A advertises D on all other potential short sides (*i.e.*, to all neighbors to which it has a high estimation error).

Finding detours is not enough: PeerWise is based on mutual agreements between nodes. A sender node can use a detour only if the relay that offers it also finds value in the sender. When requesting a detour from a neighbor, a PeerWise node includes a list of possible destinations to which it has high embedding error. The path to these destinations is more likely to be part of a detour for another node, as described in Section 3. Requests for detours are accepted only when both the sender and the receiver find mutual advantage in forwarding each other's traffic.

5.4 Maintenance

Each PeerWise node maintains two tables: a peering table and a negotiation table. The peering table tracks established, mutually advantageous peering relationships. The negotiation table is an antechamber for the peering table and tracks the nodes with which no peering has been established, but which are candidates for mutually beneficial peerings. Once a peering is established, the peer moves from the negotiation table to the peering table. An entry in either table is associated with a node i in the system and contains i's IP address, network coordinate, and a history of round trip times to i. The peering table adds the SLA and the utilization of the peering.

The SLA specifies the benefit that each node is expected to receive and offer through the peering. We allow different measures for the mutual benefit of a connection as long as the peering nodes both agree upon them. Two nodes can form a peering and agree that each of them

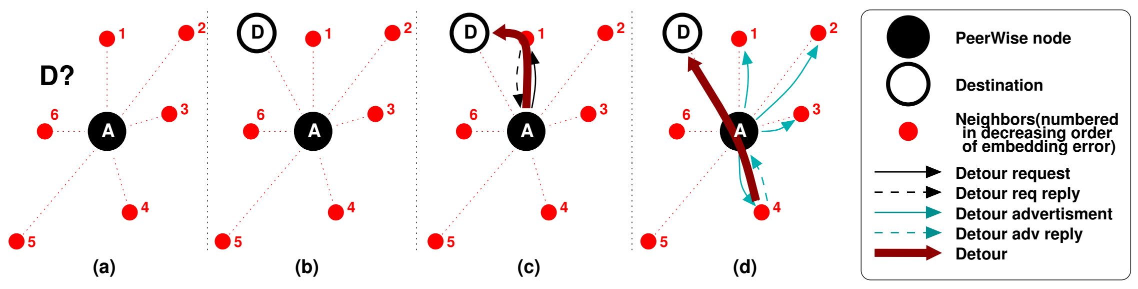


Figure 9: Detour Requests and Advertisements. (a) A wants to connect to destination D; (b) A discovers the network coordinate of D using Vivaldi or Virtual Vivaldi; (c) A requests a detour to D from the neighbor to which it has the highest embedding error; (d) A advertises its path to D to all neighbors that have positive embedding error to A.

uses the other for the same number of detours. Alternatively, they may decide that their benefit is measured in the average latency reduction obtained through each other. For example, in Figure 1, nodes A and B may establish an SLA that promises an average latency reduction of 30 ms from A to D and from B to C. In addition, two peers may establish an imbalanced peering, in which one peer benefits more than the other, if both consider the agreement to be *fair*.

Peerings may become imbalanced in time. This happens because latencies change due to failures or congestion, because peers do not respect the agreement, or because they have different connection rates to their destinations. PeerWise nodes renegotiate existing peerings to account for latency changes and to find the best detours available, as we describe in Section 7. However, we do not monitor the byte-level usage of a peering. Our focus is on finding and taking advantage of mutual latency reductions. In a previous paper [14], we describe a monitoring and accounting mechanism that ensures long-lived and mutually advantageous peerings, even when nodes are selfish or traffic demands differ.

6 Design, Part II: Policies

PeerWise is designed to be a scalable overlay for finding low-latency detours. For scalability, each node must *choose* which neighbors to maintain peerings with, *choose* among neighbors to find a relay, and *predict* whether to seek a relay for a destination.

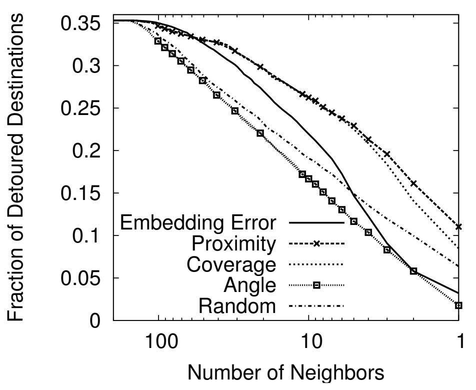
PeerWise nodes must *learn*. Nodes compute coordinates for new destinations to help other nodes predict detours. Newly used relay paths can be instrumented so that they can be dropped if the prediction of their utility was incorrect or preserved if their utility is clear. Finally, nodes must remember a recent destinations so that a neighbor set can be customized to the likely traffic stream. Learned behavior will depend on practical deployment: for example, how frequently nodes return to the same latency-sensitive destination. In fact, as a destination is contacted again and again, PeerWise might lower its standards for a "good" detour to provide improved application performance, or try reaching the destination via relays that are not obvious candidates. In this section, we make no assumptions about the utility of learned information, and instead focus on establishing a broad base of PeerWise connections for reaching all destinations.

To study neighbor and relay selection algorithms, we collected latency measurements and coordinates for 262 PlanetLab nodes and the 448 popular web servers. We considered only the PlanetLab nodes responsive at the time of the measurement. To gather this PL-Dest-Pyxida data set, we used Pyxida [24], an implementation of the Vivaldi coordinate system. To compute coordinates for the web servers, we extended Pyxida with our virtual coordinate algorithm. Every 30 seconds, for 18 hours on January 14, 2008, we took a snapshot containing RTT measurements and coordinates (virtual and non-virtual). We use only a subset of this data: median latency over the past 10 measurements, and network coordinates, all observed after Pyxida ran for two hours (to converge).

6.1 Choosing Neighbors

Each PeerWise node must be able to decide whether a new node would offer better detours than existing neighbors. A new neighbor may provide relays toward a region of coordinate space or directly to known destinations. Deciding upon future mutual advantage is a prediction of future accesses and future performance. In this section, we evaluate the ability of a PeerWise node to predict, from coordinates and measurement, whether a neighbor will contribute.

If nodes were to contact only a few, known destinations, choosing neighbors would be simple: replace a neighbor if the new one provides a better path to an interesting destination. However, we do not expect access patterns to be nearly so predictable. Instead, we wish to determine, when a new neighbor arrives, whether it is likely to provide a shortcut to a useful region in coordinate space.



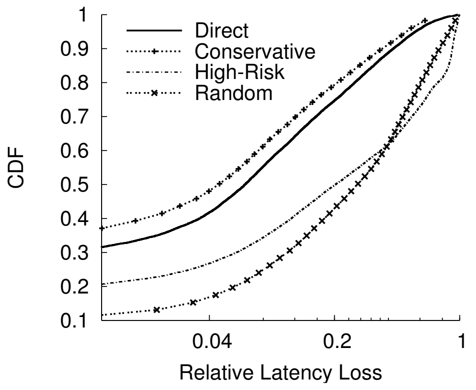


Figure 10: **(left)** *Neighbor selection algorithms*. As the number of legitimate neighbors is restricted, coverage, proximity and embedding error (for 32 or more neighbors) algorithms preserve the most detours. **(right)** *Relay selection algorithms*. Best detours are found through relays selected using the direct and conservative algorithms.

We consider a few traffic-independent neighbor selection policies, expecting that a combination of schemes would perform best. We separate them into two classes: *value* schemes are likely to provide the best detours, but may overlap; *diversity* schemes prefer relays that are different from those already chosen.

Value schemes include *embedding error* and *proximity*. *Embedding error* prefers neighbors with the largest positive error in the embedding of the source to potential neighbor edge: these nodes are likely to traverse the most coordinate distance with the lowest latency. *Proximity* prefers neighbors with the smallest absolute latency between the source and a potential neighbor.

By choosing the best neighbors exclusively, a node may miss diversity. *Coverage* uses the relay's coordinate and latency to determine the region in coordinate space that that relay covers. We split the space with a 2^4-tree structure (for scalability) and prefer neighbors that minimize the expected detour latency to every point in space. *Angle* prefers neighbors in different directions in the coordinate space. For all pairs of potential neighbors, a node computes the angle between the line segments from itself to the neighbors, and selects the neighbors with the largest angles. *Random* chooses neighbors at random to provide a point of comparison.

In Figure 10(left), we compare these neighbor selection algorithms. We vary how many neighbors a node can have from 1 to 200. At each step, we add a new neighbor based on one of the five schemes. Proximity and coverage perform the best, but embedding error also performs well with 32 or more neighbors. We choose proximity as our primary neighbor selection metric because it performs similarly to coverage and is easier to use.

6.2 Choosing Relays

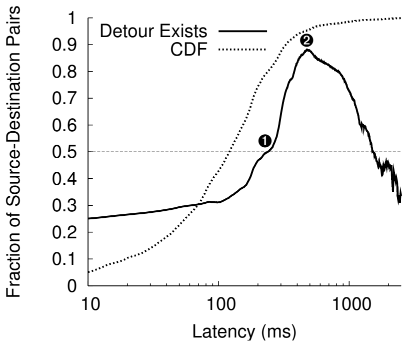
Neighbor selection determines the set of neighbors that may provide a detour path. With relay selection, a node attempts to discover quickly the neighbor that offers the best detour to a specific destination. Like server-selection problems solved by network coordinates, relay selection seeks the shortest combination of the direct path to the relay and the predicted path between relay and destination. Over time, this performance can be measured, but to minimize latency, detour performance should be predicted. At the very least, we hope to reduce the number of relays that we need to simultaneously contact to find a good detour when contacting a destination for the first time.

We consider the following policies for choosing relays for a destination. *Direct prediction* adds the measured source-to-relay latency to the estimated relay-to-destination distance in coordinate space, then chooses the relay with the lowest sum. Because latency measurements may be more reliable than coordinates, we evaluated a *conservative prediction*, which adds the source-to-relay latency measurement again to increase its influence in the prediction. This is based on the expectation that coordinates are inaccurate and seeks greater likelihood of a good detour in preference to the best detour at the top of the list. A *high-risk* scheme chooses the neighbor with the highest embedding error. Finally, *random* provides a baseline.

We select 32 neighbors for each node using the proximity-based algorithm and evaluate the four relay-selection algorithms. In Figure 10(right), we show the quality of predictions made using these algorithms in terms of relative performance lost compared to the best choice. The conservative approach performs best: approximately 80% of the detours chosen are only 20% longer than the best detour between the same pair of nodes.

6.3 Deciding Whether to Relay

Deciding whether to use a detour depends on a prediction of whether it will improve application performance.



❶	
CDF	0.80
Frac	0.50
ms	236

❷	
CDF	0.95
Frac	0.88
ms	474

Figure 11: As the latency to a destination increases, so does the probability that there is a detour.

	Correct decision		Incorrect decision	
	w/o probing	with probing	w/o probing	with probing
Detour exists	7.3%	11.1%	16.6%	12.8%
Detour absent	55.8%	57.3%	20.3%	18.8%
Total	63.1%	68.4%	36.9%	31.6%

Table 1: Using coordinates alone or coordinates with a latency probe to the destination, nodes can predict whether to use PeerWise. Probing the destination slightly increases the probability of making a correct decision.

This has two components: whether the traffic is sensitive to latency and whether a known neighbor is likely to provide a detour path. We evaluate the latter. Whether traffic is latency sensitive can be crudely inferred by ports, by commercial packet scheduling products, or by application-based proxies that can differentiate classes of traffic. In this section, we assume that the traffic is latency sensitive and attempt to predict whether to relay.

The decision of whether to relay depends first on whether virtual coordinates for the relay are available and recent. If there are no coordinates available for the destination, a node may choose to seek a relay by probing. If there are coordinates for the new destination, it may speculatively use a predicted relay, collect more information, or go directly to the destination without probing.

6.3.1 If the destination has no coordinates

If the destination lacks coordinates, the node should forward the packet directly, and if the destination is somewhat distant, *i.e.*, latency is long enough that a good detour is possible, the node may trigger latency probing from neighbors. The latency measurements by neighbors will, first, allow coordinates to be estimated and, second, provide direct latency measurements of the potential detour paths. Conveniently, if a detour path is available, the node may learn about it before the end of the second round trip (by starting the latency probing as soon as 10 ms have elapsed in the first contact).

The distance to the destination may be an indicator of whether the destination has a detour. In Figure 11, we show how often a destination has a relay within the neighbor set, given that the latency to the destination is above some value. For 95% of the edges, as the latency increases, so does the probability of a detour for the edge. The plot suggests that, after sending a probe to the destination, the longer a node waits to receive a response, the more likely it is that a detour exists for that destination. For 15% of destinations (between 236 ms and 1054 ms of latency), there is more than a 50% chance that a detour exists. We expect that actual node behavior, in terms of when to seek out a detour, will be application dependent. For instance, a node may *always* try to find a detour for frequently contacted destinations.

6.3.2 If the destination has coordinates

If the destination has known coordinates that have been gossiped, a node can decide before sending the first packet: is there likely to be a detour among its neighbors? Assuming that all coordinates are accurate, except for the measured latencies to neighbors, the node can find a shortcut without direct contact to the destination.

For certain uses of PeerWise, getting the relay right before contacting a destination is useful. If the destination will be reached with a TCP connection, the first choice can stick: the source address on the SYN packet is fixed, and the connection cannot be easily migrated to a relay. For interactive applications over long TCP connections—shell, game, chat, perhaps voice—this decision may be important.

We show that, most of the time, when the coordinates of the destination are known, a node makes the correct decision on whether to use PeerWise. We define a correct decision as finding a good relay (within 25% of the best latency reduction) when a detour exists, or not attempting to find one when a detour does not exist. All other decisions of a node (*i.e.*, attempting to find a relay when a detour does not exist or finding a bad relay) are considered incorrect. We summarize all possible situations in Table 1. We used the *proximity* policy for neighbor selection and the *conservative* policy for relay selection. Using coordinates alone, nodes make a correct decision 63.1% of the time. The prediction accuracy improves to 68.4% if the latency to the destination is known. We consider the frequency of correct and incorrect decisions to be acceptable; a more ambitious node might try to discover detours more often at the expense of making more mistakes.

7 Implementation and Evaluation

We implement PeerWise and run it under real network conditions on PlanetLab. In this section, we briefly describe our implementation, then show that this implementation can *quickly* find mutually advantageous de-

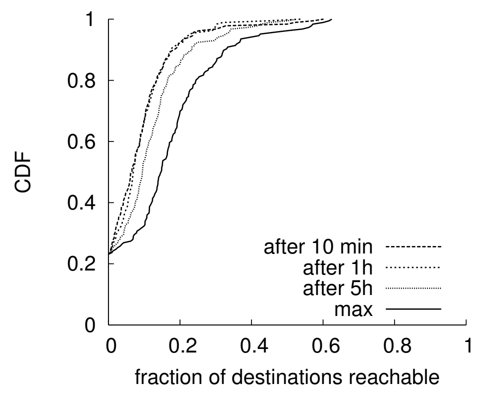


Figure 12: Fraction of the popular destinations reachable through mutually advantageous detours from PlanetLab.

tours that offer *significant* and *continuous* latency reduction. We then confirm that PeerWise detours can speed short web transfers in practice.

7.1 Implementation

We divide the functionality of PeerWise into two parts: the network coordinate system and a stand-alone daemon that includes all other components described in Section 5. We use Pyxida [24] for computing coordinates, since it is the only network coordinate system implementation we are aware of that is tested extensively under realistic network conditions [12]. Pyxida is written in Java and uses the Vivaldi algorithm [8] to compute coordinates for nodes. Each Pyxida node maintains a variable number of neighbors, updated constantly, and probes them at regular intervals. We augmented Pyxida to compute virtual coordinates for hosts that do not participate as described in Section 5.1.

We wrote the PeerWise daemon in approximately 3,000 lines of Ruby. The daemon listens for connections from other PeerWise nodes, and negotiates, establishes, and maintains mutually advantageous peerings. It communicates with Pyxida regularly, using RPC over TCP, to update the measured latencies and coordinates of the current set of neighbors as well as of the destinations that are currently served. By relying on the latency measurement and coordinate computation performed by Pyxida, we minimize the communication overhead. On the average, every node consumes less than 1KB/s (including Pyxida traffic).

7.2 Finding Detours

We ran PeerWise on 189 PlanetLab nodes, chosen for their stability, in September 2008. We focus on what detours PeerWise can find, where a detour is determined by the pings not by actual transfers. We express mutual advantage between two nodes as the number of detours that each offers the other. We experimented with three scenarios:

- **All-dest:** Each node tries to find detours to all 500 popular websites (described in Section 4) to which it can measure an RTT.
- **Rand-dest:** Each node tries to find detours to a random subset of the 500 websites.
- **Zipf-dest:** The popularity of destinations follows a Zipf distribution.

Our discussion focuses on the **All-dest** experiment, but we summarize the results from **Rand-dest** and **Zipf-dest** in Table 2. Recall that the destinations are already very popular servers, many of which use content distribution. Therefore, **All-dest** is not a best case scenario.

We describe the behavior of each node next. Nodes start looking for detours, after their network coordinates have stabilized, by successively sending detour requests to their neighbors. We limit the number of neighbors of each node to 32 for scalability and use the *proximity* policy for selecting neighbors. We make sure that no two detour requests are simultaneous: a new request is sent only when a reply (either positive or negative) has arrived for a previous one or a timeout has occurred. Each request tries to find detours to as many destinations as possible. Requests are sent continuously, even to the nodes with which peerings have been established or to the nodes that, in the past, could not offer detours. In this way, we are constantly renegotiating the peerings and are always ready to adapt to changes in latency.

PeerWise relies on the latency measurements and coordinate computations performed by Pyxida. We update both every 10 minutes. To avoid instability due to varying latencies, the updated values for latencies represent moving medians across the last 10 samples collected.

We present results for the first 36 hours of the experiment, counting from the time when nodes start requesting detours. For ease of exposition and to study startup behavior, all nodes start requesting detours simultaneously. We show that most nodes find mutually advantageous detours and that these detours lead to significant and stable latency reductions.

7.2.1 PeerWise finds detours

For each node, we count the destinations that can be reached using a mutually advantageous detour for the duration of the experiment. Figure 12 shows the distribution of the fraction of reachable destinations. Focus only on the line labeled “max” for now. Each point corresponds to a node, and its projection on the horizontal axis represents the fraction of destinations for which the node finds detours. Around 25% of the nodes cannot find any detours, while most nodes find detours to at least 10% of the popular destinations. Our results are consistent with those of the evaluation in Section 4 (see Figure 4). For **Rand-dest** and **Zipf-dest**, fewer nodes (around 50%) are able to find detours at all. This is because the number of destinations is much smaller than in **All-dest**.

	Latency reduction (§ **7.2.3**) relative (absolute)			**Longevity** (§ **7.2.4**) % of (src,dest) pairs			**Variability** (§ **7.2.4**) % of (src,dst) pairs		
	median	**10 percentile**	**90 percentile**	**$\geq$0.9**	**0.5-0.9**	**$<$0.5**	**1**	**2-10**	**$>$10**
All-dest	26% (29ms)	12% (12ms)	63% (131ms)	54%	18%	28%	67%	2%	31%
Rand-dest	25% (33ms)	12% (13ms)	60% (115ms)	36%	19%	45%	51%	23%	26%
Zipf-dest	24% (27ms)	12% (13ms)	59% (76ms)	31%	31%	38%	48%	23%	29%

Table 2: Characteristics of PeerWise detours: latency reduction, longevity and variability.

7.2.2 PeerWise finds detours quickly

How quickly are the detours discovered? We compute the fraction of destinations to which a detour is discovered by PeerWise within the first 10 minutes, 1 hour and 5 hours. Figure 12 shows the results as cumulative distributions. Many detours are discovered within the first 10 minutes of the experiment and the majority after less than an hour. Fewer and fewer detours are discovered afterward. These are mostly the detours that appear due to varying latencies—they are discovered because PeerWise constantly adapts to new latencies and coordinates.

7.2.3 PeerWise offers significant latency reduction

The detours discovered by PeerWise would not be very useful if they offered minimal latency reductions compared to the direct paths. We show that this is not the case. Recall that we have set a threshold: we consider only those detours that offer reduction of more than 10 ms and 10% of the direct-path latency. Here we focus on the latency reductions negotiated by PeerWise. In Section 7.3, we show how these reductions hold when user traffic traverses the detour path.

We compute all latency reductions for each (source, destination) pair for which a detour exists, both as absolute (milliseconds) and relative (fraction of the direct path latency) values. We show the median, 10th and 90th percentiles in Table 2. The median latency reduction is 29 ms or 26% of the latency of the direct path. 10% of the pairs have a reduction of more than 131 ms. This is caused by unusually high direct-path latencies, possibly due to traffic shaping. By circumventing these slow links, PeerWise can offer significant latency reduction.

7.2.4 Longevity and variability

PeerWise nodes may offer continuous latency reduction to a destination using several peerings. For each (source, destination) pair, we evaluate how long PeerWise offers reduction and with how many different relays. Ideally, every destination will be reached continuously through the same peering. Long-lived reductions through the same peering offer nodes more choices in when to use the mutually advantageous connection.

We consider two metrics: *longevity* and *variability*. Longevity captures how PeerWise nodes maintain latency reduction once a detour is discovered. We define the longevity of a destination D from a node S as the fraction of time that PeerWise offers S a detour to D, after PeerWise first learns about a shorter path from S to D. A longevity of 1 for the pair (S, D) means that, after PeerWise discovers the first detour between S and D, it will always offer some detour between S and D. Variability represents the number of different relays that S uses to obtain continuous reduction to D. The lower the variability, the easier it is to maintain latency reduction.

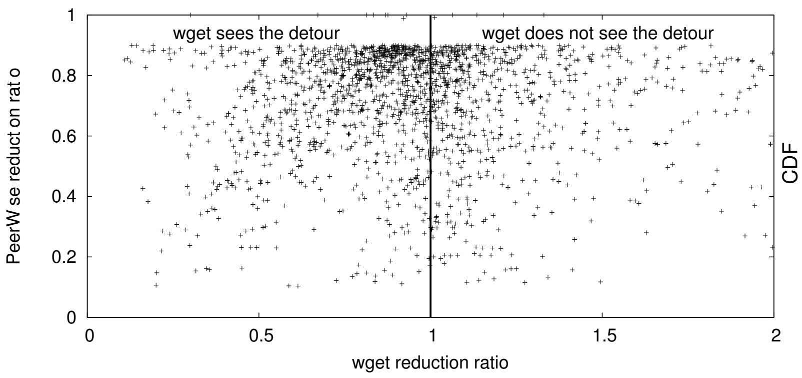
Table 2 summarizes longevity and variability for all (source, destination) pairs for which PeerWise offers latency reduction. For **All-Dest**, more than half of the pairs have a longevity higher than 0.9. 67% of the pairs use only one relay. When fewer destinations are selected at random or using a Zipf distribution, the number of detours, their longevity, and variability are reduced. However, about half of the (source, destination) pairs still have longevity higher than 0.5 and variability of 1.

7.3 Using Detours

We show how the detours discovered by PeerWise translate in real life. Can user-level applications benefit from the network-level detours of PeerWise? From each PlanetLab node running PeerWise, we download the front page of each of the 500 popular websites to which a mutually-advantageous detour exists. We use *wget* to perform two transfers every time it is called: one using the direct path and one using the PeerWise detour. To make the web request follow the detour path, we install the *tinyproxy* HTTP proxy on every PlanetLab node that can be used as a relay. We run each transfer 100 times, alternating whether detour or direct comes first, and record the individual completion times.

We verify whether the detours promised by PeerWise are seen by the web transfers. For each (source, destination) pair with a detour in PeerWise, we compute the *wget* reduction ratio—the ratio between the median relay transfer time and the median direct transfer time—and plot it against the PeerWise reduction ratio—the latency reduction ratio promised by PeerWise. Figure 13(left) presents the results. For 58% of the pairs, the *wget* reduction is less than 1; web transfers take less time through the relay than through the direct path, as predicted by PeerWise. However, many PeerWise detours do not materialize for the *wget* transfers.

We explain the dissonance between the PeerWise view and the application view next. PeerWise detours are determined by network-level pings. On the other hand, the *wget* end-to-end latency includes server and proxy wait



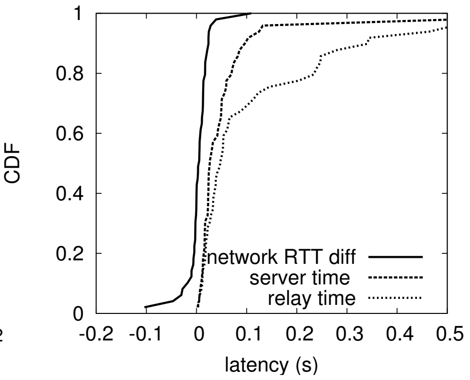


Figure 13: **(left)** *Wget* latency reduction versus PeerWise latency reduction: 58% of all PeerWise detours achieve latency reduction in real life. **(right)** Distributions of average server wait times, relay times, and difference between *wget* and PeerWise RTTs for all detour transfers. Relay times inflate application latencies the most.

times and thus may be larger than network latency. Further, PeerWise detours are based on medians of latencies gathered over long periods of time. Due to potential latency variations, these medians may differ from the RTTs at the time of the transfer.

To quantify the factors that inflate the application latency, we instrument our experiment as follows. During the web transfers, we run *tcpdump* on every relay node and log all proxy traffic. Using the packet timestamps, we compute, for each detour transfer, the network latency (from the TCP connection setup), the time spent at the relay and the time waiting for the server. Figure 13(right) shows the distributions of average server time, relay time and of the difference between network latency at transfer time and latency promised by PeerWise. The time spent at the relay and at the server accounts for most of the inflation in application latency: half of the relays induce an additional average latency of at least 50 ms. PeerWise predicts the network part of the *wget* transfer time well.

All relays are PlanetLab nodes; PlanetLab does not always reflect the realities of the Internet. We believe that the slowness of PlanetLab is the main factor that contributes to the unusually high relay time for our transfers. To confirm, we set up *tinyproxy* on a computer with minimal load, located at University of Maryland and run web transfers through it. The average relay time for all transfers through the UMD proxy is 5ms, less than 95% of all PlanetLab relays. If we consider the hypothetical situation in which all PlanetLab relay times were replaced by the average UMD relay time—effectively minimizing the time spent by a transfer at the relay node—then 78% of our web transfers would see the detours promised by PeerWise. We conclude that PeerWise has the potential to improve application performance.

8 Discussion

We discuss some of the implications that wide adoption of PeerWise would have for both ISPs and users.

8.1 Implications for ISPs

Overlay networks violate routing polices. How then would inter-domain routing policy and traffic engineering practices coexist with widespread PeerWise deployment? Routing overlay networks enable rule violations: customers and peers provide transit, and selfish routing [25] can subvert traffic engineering decisions. We discuss each in turn.

Customers provide transit, which is forbidden in inter-domain routing [9]. Even when a detour AS path precisely matches the direct (because an overlay node lies within the address space of one of the autonomous systems of the path), the overlay node is still a customer and a customer still provides transit. Whether that customer has an autonomous system or instead pays a monthly fee for a residential connection hardly matters.

Overlay networks bypass traffic engineering decisions. It is unclear to what extent the excessively long latency paths are deliberately chosen by network administrators. One might worry that a successful deployment of PeerWise would hamper ISP efforts to shape traffic toward slower, but less utilized, links. PeerWise is not intended for high-bandwidth transfers. Its structure discourages bandwidth consumption: we intend to shave packet transmission latency and, because each pair of nodes must strive to maintain the fairness of the application-level SLA that connects them, they may not consume unnecessarily. Downloading a large file through PeerWise may not reduce the download time significantly, considering the many other bottlenecks in the network (loss, client-side queuing, server load, etc.) [21, 4, 22].

8.2 Implications for Users

Forwarding traffic *through* and *on behalf of* others raises issues of privacy and liability for PeerWise users. Although unencrypted traffic is "public" regardless of the path it takes, it is reasonable to assume that users would be more reluctant to forward their traffic through other users than directly through the "faceless" ISPs. Another concern is being held liable for forwarding potentially illicit traffic on behalf of another user. While some such traffic may be straightforward to filter (and negotiate in a PeerWise SLA), say by mechanisms similar to parental controls, such an approach requires knowing "questionable" destinations ahead of time, and leads to both false negatives and positives. A more general mechanism for *non-repudiation*—a means of verifiably proving to authorities the source of forwarded traffic—may be more appropriate, but is beyond the scope of this paper.

A potential extension of PeerWise would be to limit one's neighbors to a set of trusted users, determined for example via friend-of-friend links in an online social network, similar to the *f2f* file store [16]. While such an extension may obviate the concerns of non-repudiation, it may exacerbate privacy concerns; users may be less inclined to forward private traffic through their friends.

Interestingly, PeerWise can assist in *securing* an end-user's traffic. Reis *et al.* demonstrated that some ISPs modify users' web pages in transit [26]. PeerWise could assist in routing around such ISPs, or perhaps in lending greater credence to a page's authenticity.

9 Conclusions

PeerWise is based on building overlay networks from *mutually advantageous peerings*; we show that such a simple, locally enforced mechanism is sufficient to provide detour routes in the Internet. Surprisingly, pairs of nodes can help each other: few nodes are so well positioned that they need no help, and few are so poorly positioned that they can help no one. Our evaluation of PeerWise on two sets of real world latencies and on PlanetLab shows that most nodes can find good detours, reducing latency by at least 10 ms and 10%. PeerWise finds detours to both regionally and globally popular destinations, as well as to websites that use other latency-reduction techniques such as mirroring or DNS redirection. Most detours are long-lived and stable and reflect well the performance of applications using them.

Acknowledgments

We are grateful to our shepherd, Venugopalan Ramasubramanian, and to the NSDI reviewers for their help in improving this paper. We also thank Peter Druschel, Bo Han, Jay Lorch, Harsha Madhyastha, Justin McCann, Larry Michele, Alan Mislove, Vivek Pai, and Angie Wu for their comments. This work was supported by NSF grants CNS-0435065, CNS-0643443 and CNS-0626629, and by a Microsoft Live Labs fellowship.

References

[1] Alexa. http://www.alexa.com/.

[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient overlay networks. In *SOSP*, 2001.

[3] A. Bharambe, J. R. Douceur, J. R. Lorch, T. Moscibroda, J. Pang, S. Seshan, and X. Zhuang. Donnybrook: Enabling large-scale, high-speed, peer-to-peer games. In *SIGCOMM*, 2008.

[4] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *IEEE Infocom*, 2000.

[5] B. Cohen. Incentives build robustness in BitTorrent. In *P2PEcon*, 2003.

[6] J. Corbo and D. Parkes. The price of selfish behavior in bilateral network formation. In *PODC*, 2005.

[7] L. Cox and B. Noble. Samsara: Honor among thieves in peer-to-peer storage. In *SOSP*, 2003.

[8] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: a decentralized network coordinate system. In *SIGCOMM*, 2004.

[9] L. Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM Transactions on Networking*, 9(6):733–745, 2001.

[10] K. Gummadi, S. Saroiu, and S. Gribble. King: Estimating latency between arbitrary Internet end hosts. In *IMW*, 2002.

[11] K. P. Gummadi, H. Madhyastha, S. D. Gribble, H. M. Levy, and D. J. Wetherall. Improving the reliability of internet paths with one-hop source routing. In *OSDI*, 2004.

[12] J. Ledlie, P. Gardner, and M. Seltzer. Network coordinates in the wild. In *NSDI*, 2007.

[13] J. Ledlie, M. Seltzer, and P. Pietzuch. Proxy network coordinates. Tech. rep., Imperial College London, 2008.

[14] D. Levin, R. Baden, C. Lumezanu, N. Spring, and B. Bhattacharjee. Motivating participation in Internet routing overlays. In *NetEcon*, 2008.

[15] D. Levin, R. Sherwood, and B. Bhattacharjee. Fair file swarming with FOX. In *IPTPS*, 2006.

[16] J. Li and F. Dabek. F2F: reliable storage in open networks. In *IPTPS*, 2006.

[17] C. Lumezanu, D. Levin, and N. Spring. PeerWise discovery and negotiation of faster paths. In *HotNets*, 2007.

[18] A. Nakao and L. Peterson. Scalable routing overlay networks. In *ACM SIGOPS Operating Systems Review*, 2006.

[19] A. Nakao, L. Peterson, and A. Bavier. A routing underlay for overlay networks. In *SIGCOMM*, 2003.

[20] T. S. E. Ng and H. Zhang. Predicting Internet network distance with coordinates-based approaches. In *INFOCOM*, 2002.

[21] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP throughput: A simple model and its empirical validation. In *SIGCOMM*, 1998.

[22] J. Padhye and S. Floyd. Identifying the TCP behavior of web servers. In *SIGCOMM*, 2001.

[23] M. Piatek, T. Isdal, T. Anderson, A. Krishnamurthy, and A. Venkataramani. Do incentives build robustness in BitTorrent? In *NSDI*, 2007.

[24] Pyxida. http://pyxida.sourceforge.net/.

[25] L. Qiu, Y. R. Yang, Y. Zhang, and S. Shenker. On selfish routing in Internet-like environments. In *SIGCOMM*, 2003.

[26] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver. Detecting in-flight page changes with web tripwires. In *NSDI*, 2008.

[27] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and H. M. Levy. An analysis of Internet content delivery systems. In *OSDI*, 2002.

[28] S. Savage, T. Anderson, A. Aggarwal, D. Becker, N. Cardwell, A. Collins, E. Hoffman, J. Snell, A. Vahdat, G. Voelker, and J. Zahorjan. Detour: A case for informed Internet routing and transport. *IEEE Micro*, 19(1):50–59, 1999.

[29] S. Savage, N. Cardwell, D. Wetherall, and T. Anderson. TCP congestion control with a misbehaving receiver. *ACM CCR*, 29(5):71–78, 1999.

[30] M. Sirivianos, J. H. Park, X. Yang, and S. Jarecki. Dandelion: Cooperative content distribution with robust incentives. In *USENIX*, 2007.

[31] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An overlay based architecture for enhancing Internet QoS. In *NSDI*, 2004.

The USENIX Association

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

For more information about membership and its benefits, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Corporate Supporters

USENIX Patron

Microsoft® Research

USENIX Benefactors

Google™

hp® invent

Infosys® POWERED BY INTELLECT DRIVEN BY VALUES

LINUX PRO MAGAZINE

NetApp™

Sun® microsystems The Network is the Computer™

vmware®

USENIX & SAGE Partners

Ajava Systems, Inc.
DigiCert® SSL Certification
FOTO SEARCH Stock Footage and Stock Photography
Hyperic Systems Monitoring
Splunk
Zenoss

USENIX Partners

Cambridge Computer Services, Inc.
GroundWork Open Source Solutions
Xirrus

SAGE Partner

MSB Associates

ISBN-13: 978-1-931971-67-6
90000
9 781931 971676