Project no. 609551
Project acronym: SyncFree
Project title: *Large-scale computation without synchronisation*

European Seventh Framework Programme
ICT call 10

Deliverable reference number and title:	D.3.3 Protocols for invariant preservation and security
Due date of deliverable:	March 23, 2016
Actual submission date:	March 23, 2016
Start date of project:	October 1, 2013
Duration:	36 months
Name and organisation of lead editor for this deliverable:	NOVA
Revision:	0.1
Dissemination level:	CO

Contents

1 **Executive Summary** 1

2 **Milestones in the Deliverable** 3
- 2.1 Status of the work . . . 4

3 **Contractors Contributing to the Deliverable** 5
- 3.1 KL . . . 5
- 3.2 INRIA . . . 5
- 3.3 Louvain . . . 5
- 3.4 Nova . . . 5
- 3.5 Basho . . . 5
- 3.6 Trifork . . . 5

4 **Results** 6
- 4.1 Security . . . 6
  - 4.1.1 Access control in weakly consistent systems . . . 6
  - 4.1.2 Secure dissemination . . . 9
- 4.2 Invariants . . . 10
  - 4.2.1 Enforcing Numeric Invariants . . . 11
  - 4.2.2 Explicit Consistency . . . 11
- 4.3 Extensions to Works Previously Reported . . . 15
  - 4.3.1 Quality-of-data . . . 15
  - 4.3.2 Delta State-based CRDTs . . . 16
  - 4.3.3 Efficient Support of Large CRDTs in Riak . . . 16
  - 4.3.4 Causality with partial knowledge . . . 17
  - 4.3.5 Other works . . . 17

5 **Publications** 18

6 **Published papers** 23
- 6.1 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pages 371– 384, New York, NY, USA, 2016. ACM. . . . 23
- 6.2 Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict- free partially replicated data types. In Proceedings of the 7th IEEE Interna- tional Conference on Cloud Computing Technology and Science (CloudCom 2015). IEEE, Nov 2015. . . . 38
- 6.3 Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Queue, January 2016. ACM. . . . 47
- 6.4 Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Commun. ACM, April 2016. ACM. . . . 65

6.5 Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. Geo-replication: Fast if possible, consistent if necessary. IEEE Data Engineering Bulletin (to appear), 2016. 71
6.6 Valter Balegas, Sérgio Duarte, Carla Ferreira, Nuno Preguiça, and Rodrigo Rodrigues. Making Weak Consistency Great Again. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. . 84
6.7 Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. The problem with embedded CRDT counters and a solution. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. 88
6.8 Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. 92
6.9 Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE Tool: Proving Weakly-Consistent Applications Correct. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. 96

7 Submitted papers 101
7.1 Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. Access control for weakly consistent cloud-storage systems. Submitted for publication, 2016. 101
7.2 Christopher Meiklejohn. Loquat: A partially replicated, secure, broadcast protocol for edge computation. Submitted for publication, 2016. 111
7.3 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. Submitted for publication, 2016. 120
7.4 Seyed H. Haeri (Hossein), Peter Van Roy, Carlos Baquero, and Christopher Meiklejohn. Deduction with partial knowledge about causality. Submitted for publication, 2016. 148

1 Executive Summary

The SyncFree project aims to enable trustworthy large-scale distributed applications in geo-replicated settings. The core concepts are replicated yet consistent data types (CRDTs) which allow information dissemination and sharing without the need for global synchronization.

Within the project, Work Package 3 (WP3) coordinates the work on extending the safety, quality and security guarantees provided by a system that uses minimal synchronisation. This deliverable, *Protocols for invariant preservation and security*, reports results of task 2, *Divergence control and quality-of-data* and task 3, *Security*. Regarding invariant preservation, the research focused on providing mechanisms for ensuring that global invariants are preserved in the presence of concurrent updates. Regarding security, the research focused on providing access control in weakly consistent environments, where access control policies can be updated concurrently with the execution of updates.

The specific requirements addressed in our work were driven by the use cases studied in WP1, but also from previous experience of the project partners, both industrial and academic. We now briefly overview the results achieved during the reporting period. Although the solutions presented address the problems that we promised to address, we expect to continue improving some of the works until the end of the project.

Invariants Although a large number of applications can work correctly under weak consistency models, other applications need to maintain global invariants that cannot be enforced using such models.

To address this problem, we have proposed explicit consistency, as a novel consistency model that extends weak consistency with the enforcement of global application invariants. We have proposed a number of techniques to enforce explicit consistency, notably a methodology for extending applications with reservation mechanisms [3, 5], CISE logic for proving that a given reservation system enforces invariants [12], and the Bounded Counter for enforcing numeric invariants in systems that do not implement the full Explicit Consistency approach [2].

In this period, we have published the work on CISE logic [12] and we have completed a tool that uses CISE logic for helping programmers identifying the operations that can lead to problems [17].

We have additionally worked on a solution for enforcing invariants by repairing invariants [4], which is complementary to the reservation-based approach. This solution combines several new CRDT designs with a tool that proposes extensions to application operations that guarantee that invariants are preserved even in the presence of concurrent operations.

Security Access control is an important aspect of information systems as these systems store sensitive information. In weakly consistent systems, however, concurrent modifications and out-of-order delivery of data updates and policy changes impose security threats due to inconsistencies among policies and data operations.

We have proposed an access control model for eventually consistent data stores that avoids information leakage, unauthorized modifications and guarantees con-

vergence of the copies of the security policy among replicas [22]. In particular, we address the problem of policy changes which modify the permissions being executed concurrently with the execution of updates. We consider two settings: a data-center setting, where all replicas and clients run in the data-center and can be assumed to be trusted; and an extension of this that includes clients that can synchronize among them in peer-to-peer interactions.

We have additionally addressed the problem of providing secure data dissemination among replicas, by proposing changes to the dissemination protocols used in Lasp [16]. The proposed protocols create dissemination trees where only nodes with the same security permissions participate.

Extensions to Works Previously Reported A number of works that started being developed in the context of Task 3.1 and 3.2 have continued during this period. The following works have led to publications in this period: (i) the model and designs of CRDTs with delta-mutations [1, 20] for efficient synchronization; (ii) the model of conflict-free partially replicated data types for partially caching large objects [10]; (iii) fundamental techniques for tracking causality [7, 8]. We have also continued our research on how to inform applications about the potential divergence of the data being read.

2 Milestones in the Deliverable

WP3, tasks 2 and 3 contribute to the following milestone, shared with other work packages:

Mil. no	Mil. name	Date due	Actual date	Lead contractor
MS3	Results consolidation and validation plan	M30	M360	INRIA

Task 3.2 has contributed to this deliverable and milestone by focusing mainly on solutions for enforcing global invariants, as stated in the project proposal:

> This deliverable will report on the protocols for divergence control and Quality-of-Data (QoD). This deliverable includes protocols for decentralised invariant preservation and divergence control. Synchronisation-freedom comes at the price of divergence among replicas. While many applications can work properly in this context, others require additional information, e.g., QoD metrics such as an estimate of the amount of divergence, or bounding the divergence, or ensuring global invariants. Compared to previous work [8, 50, 122], extreme-scale replication poses new challenges, both in the definition of divergence metrics and in the scalability of the divergence measurement and control algorithms. We will identify sub-classes of CRDTs according to the guarantees they provide, and formally analyse the degree of synchronisation that these sub-classes require. We will also explore the design space of CRDTs and associated protocols for efficiently preserving global invariants, from using decentralised solution such as escrow, reservation, and exo-leasing [74, 82], to solutions that use some synchronisation. The main challenge is to push the limits of the efficiency of the implementation of CRDTs and supporting systems for various classes of invariants, and the seamless integration of different solutions in the same platform. [month 18]

Task 3.3 addresses the specific security aspects of systems adopting weakly consistent data models, in particular those using CRDTs. In particular, we focused mainly on the problem of access control, as stated in the project proposal:

> This deliverable contains final results, including a security approach for the deployment of a CRDT-based platform. In applications such as the Danish Shared Medication Records, security is paramount. In particular, we will focus on access control in a decentralized environment. Access control is fundamental to ensure that information is disclosed only to authorised users has been widely studied, both in centralised [57, 90] and distributed databases [2, 10, 11]. However, extreme scale replication of mutable shared data raises new issues. Access control policies must be propagated and enforced in all replicas, but the use of different policies may compromise eventual consistency. Wobber et al. [120] propose an EC access control mechanism, assuming a separate

security layer with a single root of authority, and restricting policy expressiveness. Recent approaches [26, 28] integrate access control with data entities, verifying conformance of arguments with access control policies statically, using refinement types. In particular, Caires et al. [26] supports access control that depends on the state of data. This task will extend this approach to replicated data and combine it with the EC guarantees of Wobber et al. [120].

2.1 Status of the work

This deliverable reports the proposals made in the context of WP3 (in collaboration with other Work Packages) for invariant preservation and security. Although the work presented herein matches the expected outcomes when the project has been proposed, some of the works are still on-going and we expect to developed improved versions until the end of the project.

In particular, regarding security we are still finishing the implementation of the proposed access control model for settings where not only servers synchronize their replicas, but where also clients can engage in peer-to-peer interactions.

Regarding protocols for enforcing global invariants, we are finishing the development of a solution that enforces invariants by allowing any concurrent operations to execute without coordination and automatically repairs invariants when merging the state of replicas. This approach is complementary to the other solution developed, where invariants are preserved by restricting the execution of operations that may lead to an invariant violation.

Finally, regarding our work on QoD for providing divergence information for applications, we continued our work on evaluating different protocols for disseminating information for computing divergence metrics, which we will also evaluate in the context of the FMK experiment that we have planned (as described in D5.2).

3 Contractors Contributing to the Deliverable

The following contractors contributed to the deliverables:

3.1 KL

Annette Bieniusa, Mathias Weber.

3.2 INRIA

Mahsa Najafzadeh, Jordi Martori, Marc Shapiro, Alejandro Tomsic, Tyler Crain, Pascal Urso, Marek Zawirski, Michał Jabczyński.

3.3 Louvain

Christopher Meiklejohn, Seyed H. HAER, Iwan Briquemont, Manuel Bravo, Zhongmiao Li, Peter van Roy.

3.4 Nova

Valter Balegas, Sérgio Duarte, João Leitão, Ali Shoker, Carla Ferreira, Paulo Sérgio Almeida, Rodrigo Rodrigues, Carlos Baquero, Nuno Preguiça.

3.5 Basho

Russell Brown, Engel Sanchez.

3.6 Trifork

Kresten Thorup.

4 Results

This section presents the results obtained in WP3, during the reporting period. We organize the results in three groups: *Security* (§ 4.1), discussing the work on security features; *Invariants* (§ 4.2), detailing how to enforce invariants while minimizing the required coordination; and *Other works* (§ 4.3), describing extensions to works preciously reported.

4.1 Security

In the context of security, as planned, our main focus has been on defining an access control mechanism that can work in weakly consistent systems. We describe our proposal in section 4.1.1. We have additionally addressed the problem of guaranteeing secure dissemination of information among replicas to avoid leaking information. This work is described in section 4.1.2.

4.1.1 Access control in weakly consistent systems

Access control is important to protect sensitive information stored in information systems. Security policies describe the rules applied to determine whether a user is allowed to perform a specific operation. The policy of the organization running the information system usually changes over time. These changes need to be reflected in the implementation of the access control system. Therefore, access control systems support updating the policy rules at runtime. For strongly consistent systems, the implementation of access control is well understood since the order in which operations are processed is the same on all replicas. In weakly consistent systems however, a global order over the operations does not exist. Concurrent modifications and message reordering impose security threats due to inconsistencies among policies and data operations.

A typical situation in the evolution of policies is the reduction of the permissions of a user. In a social network example, this reduction can be achieved by removing a user from the list of friends of some other user. As a result of this reduction, the user may no longer perform operations, that he was previously allowed to perform. In the social network example, this may mean that the user is no longer allowed to observe changes like the upload of new photos. In the strongly consistent case, the semantics of this policy modification is clear because of the global order of operations. In a weakly consistent data store, the policy modification and a subsequent upload of a picture may arrive in different order on different replicas. This leads to inconsistency between the intended policy semantics and the fact that the new photo can be accessed by the user between the arrival of the upload operation and the arrival of the policy modification. For weakly consistent systems, the semantics and also the correctness of the access control system can only be defined based on the local order of operations for each replica.

In our work we assume that (server) replicas of the system are trusted, which is a common assumption in cloud environments. We further assume that the system includes a mechanism to authenticate clients, which can be implemented by using secret sharing (e.g. user/password) or certificates. Clients can be assured of being

communicating with the servers by using secure channels (e.g. using SSL).

4.1.1.1 Correctness of Access Control in Weakly Consistent Systems

We denote the happens-before relation between operations op_1 and op_2 based on the order of execution on a replica as $op_1 \rightarrow op_2$. The policy consists of right assignments $(r, s, o) \in \mathit{Rights} \times \mathit{Subjects} \times \mathit{Objects}$. The access control system checks all operations performed on the system. A rights assignment (r, s, o) *permits* an operation op performed by subject s', written $(r, s, o) \models (op, s')$ iff $\mathsf{target}(op) = o$, $s = s'$ and the capabilities r allow operation op. The policy can be modified at runtime using *policy modifying operations* which assign new capabilites for a subject-object pair, also written (r, s, o) for assigning capabilities r for object o to subject s.

For strongly consistent systems, we can assume that the happens-before relation is a total order on the operations. We can define the correctness of access control as follows: If $(r, s, o) \rightarrow (op, s')$, $\mathsf{target}(op) = o$ and $(r, s, o) \not\models (op_R, s)$ and $\mathsf{target}(op_R) = o$ then $(r, s, o) \not\rightarrow (op_R, s)$. If the rights assignment (r, s, o) restricts the capabilities of s such that s may not perform op_R on object o then op_R cannot be allowed by the access control system after performing the rights assignment (r, s, o). In particular, this means that $(op, s') \not\rightarrow (op_R, s)$, assuming op_R to be a read operation shows that modifications performed by op on object o are not visible to s.

The reasoning above shows that there is a relation between rights assignments and subsequent data operations that has to be retained by the access control system to be correct. We call this relation the *protection relation* and write $(r, s_1, o) \lhd (op_W, s_2)$ to mean the rights assignment (r, s_1, o) protects the operation op_W by subject s_2. The protection relation is a subset of the happens-before relation. $(r, s_1, o) \lhd (op_W, s_2)$ holds if $(r, s_1, o) \rightarrow (op_W, s_2)$ and $\mathsf{target}(op_W) = o$. We claim that the protection relation has to be retained on every replica in order for an access control system to be correct in the context of weakly consistent systems.

A more complete presentation of the model is available in the submitted publication 7.1 ([22]).

4.1.1.2 Implementation on Antidote

We are currently building a prototype implementation of an access control system that preserves the protection relation called ProPreAC. The underlying idea is to represent all concurrent rights assignments and require the set of capabilities *Rights* to form a lattice. A rights assignment (r, s, o) replaces all rights assignments visible when performing (r, s, o). All rights assignments that happened concurrently are retained. To compute the capabilities a subject s has on object o, we consider all of these concurrent rights assignments $\{(r_1, s, o), (r_2, s, o), ..., (r_k, s, o)\}$ and compute the minimum capabilities $\min(r_1, r_2, ..., r_k)$. This way, a possible reduction of capabilities is retained. The system has to retain the protection relation when applying downstream operations. If $(r, s_1, o) \lhd (op_W, s_2)$ then (r, s_1, o) has to be applied before (op_W, s_2) on all replicas.

The current implementation is based on Antidote. We implemented a CRDT implementation suitable for modeling the concurrent rights assignments. The capabilities are modeled as sets of operations that may be performed by the subject. The minimum, in this case, is the intersection of all concurrently assigned capability

sets. When generating the downstream operations, the current policy is added as an additional parameter to the operation. Applying the downstream operation on a replica removes all rights assignments of this old policy and adds the new rights assignment. Because Antidote preserves the happens-before relation between operations and the protection relation is a subset of this happens-before relation, this CRDT is sufficient to retain the correctness of the policy handling.

We are currently working on using ProPreAC as the access control mechanism for Antidote. In such case, whenever a new operation is issued, the system tags the operation with the user that issued the operation, and the verification of permissions is executed as explained before.

4.1.1.3 Implementation on Titan Titan is a system that allows clients to replicate shared objects and propagate modifications in a peer-to-peer fashion (ad detailed in deliverable 2.3). Titan integrates with cloud databases, by forwarding updates executed in the clients to the data centers and retrieving new updates from the servers. Although Titan has been designed to allow integration with different cloud databases, its use of CRDTs makes it a natural extension for using data stored in Antidote in client machines.

We are currently extending and implementing the proposed access control model for Titan. While in the implementation on Antidote we assume that replicas (servers) are trustable and will only execute correct code without the possibility of forging different causal relations among operations, it is not reasonable to assume the same when replicas execute in the clients.

Our approach to extend the proposed access control model relies on using servers to certify causal dependencies. In this case, when a server receives an operation from a client, it certifies the causal dependencies of the received operation to be the the current state in the server.

This avoids that a client, after seeing that she no longer has rights to execute some operation, tries to submit the operation by stating that it was executed without having the knowledge that her permission had changed. In the case where an operation is executed in a client concurrently with a change in the associated permission in the server, this approach takes a conservative approach that denies the execution of the operations.

We should note that this approach still allows the attack that a client, after observing a change in her permissions, received from a given server, can contact some other server for submitting an operation she is no longer allowed to execute. If the server has no knowledge of the change in permissions when certifying the causal dependencies, the operation will be accepted.

We are considering alternatives to this approach, including the use of wall clocks in the process. In this case, a server receiving an operation from a client would certify the time when the operations was received. The time of operations would be compared with the time when the changes in the authorisation rules were performed to decide whether an operation should be accepted or not. Obviously, we would need to consider that it is impossible to synchronize the clocks in all machines and that a time interval should be considered for the operation times, as in Spanner [11]. This approach had the potential to reduce the vulnerability window.

For avoiding reads from clients that are no longer allowed to read the state of

an object, information propagated among client nodes is ciphered with a symmetric key. For each access control configuration (or view), the system generates a new symmetric key. This keys are maintained in the servers, with client requiring a key having to ask for the key of a given view to a server.

We are also considered the possibility of adopting the dissemination approach that will be presented in the next section. However, we currently believe that in a setting where the number of clients might be small, involving all clients in the propagation of information might be a better approach. We intend to evaluate this experimentally.

4.1.2 Secure dissemination

In the context of supporting Lasp in a large number of clients, Meiklejohn [16] has proposed Loquat for providing secure dissemination of information. Loquat provides security by guaranteeing that nodes that cannot access some information will not belong to the dissemination tree for that information. This work is complementary of the work presented before.

Loquat includes two main features. First, an extension to the Plumtree [14] epidemic broadcast protocol that supports partial replication of data. Second, an extension to the protocol that supports information flow control, where nodes that are not authorized to receive confidential information will not receive dissemination of that information.

4.1.2.1 System model This work assumes a dynamic set of nodes, where each node in the system has a globally unique node identifier. We assume the the crash-stop failure model, and nodes that crash recover by rejoining the cluster with a new globally unique node identifier and no state: identical to a new node. We assume non-Byzantine network and node behavior.

We assume a set of unique tokens representing both the level of secrecy and integrity placed on scopes of messages. We assume that messages within a given scope can only move in two directions: messages can decrease in secrecy, or increase in integrity (through additional declassification, or increased authentication.) We assume actors in the system do not act maliciously, by lying about the security contexts they are privileged to (non-Byzantine behaviour).

4.1.2.2 Information flow control To support information flow control, Loquat alters the Plumtree protocol by assuming that there is a single spanning tree computed for each security context s and scope γ. For instance, the set of nodes that can read a given scope s with the same security context γ will compute their own spanning tree. Each node keeps a map for a given scope and security context to a pair containing the "eager" and "lazy" sets.

We assume that the lazy set for all scopes and security contexts is pre-populated with either the entire set of nodes in the cluster, or the result of a peer sampling services, for larger clusters. We augment each of the four message types (IHAVE, GRAFT, PURGE, BCAST) to carry the scope and security context information for every request.

We enumerate the steps in the initial spanning tree construction for a scope s and a security context γ, which may occur over several rounds of broadcast.

1. At the beginning of the execution of the protocol, no members exist in the "eager set" of the broadcasting node, and a random set of nodes is placed into the "lazy set".
2. Given this, the first message that is broadcast for scope s will be the IHAVE message, containing the scope s, a unique message identifier derived from the version vector c, and the security context for the scope γ.
3. Receiving nodes will not receive an eager broadcast within the time interval, given these messages were never transmitted with eager broadcast.
4. When the timeout expires, nodes for which the information exchange predicate holds (i.e., for which the integrity labels and secrecy labels are compatible) will send the receiving node the GRAFT message to move nodes into the broadcasting node's "eager set".
5. Finally, the message will be delivered by BCAST from the broadcasting node containing the actual message.

This will cause a minimal spanning tree to be computed and maintained that contains nodes that are only allowed to view the confidential information for a given scope s.

4.1.2.3 Next steps This work in on-going and we are completing its implementation and evaluation as the dissemination infrastructure for running Lasp applications that address Internet-of-things scenarios. We will also evaluate whether the proposed protocols are suitable for disseminating information among "heavy" clients, as targeted by Titan.

4.2 Invariants

Systems that adopt weak consistency models have to deal with concurrent operations not seeing the effects of each other. If CRDTs can be used to guarantee eventual convergence in these cases, they cannot be used to guarantee that application invariants are enforced, which can lead to non-intuitive and undesirable semantics.

To address this problem, we have researched solutions on how to to maintain application invariants while minimizing coordination. Most of the work performed in this context have already been reported in deliverable 3.2. For completeness, we make a brief overview of our proposals for maintaining global invariants, detailing only the work that has been performed since M24.

The first work addresses numeric invariants, which accounts for an important class of application invariants. The second is more general and can efficiently address generic application invariants by moving coordination outside of the normal flow of operation execution. Finally, we are developing an approach that allows to enforce some invariants without any coordination, by applying the ideas of CRDTs over a set of objects that can be modified independently.

4.2.1 Enforcing Numeric Invariants

Our first work focused on enforcing numeric invariants [2] in the presence of concurrent updates to counter objects. To this end, we proposed a novel abstract data type called Bounded Counter. This replicated object, like conventional CRDTs, allows for operations to execute locally, automatically merges concurrent updates, and, in contrast to previous counter CRDTs, also enforces numeric invariants while avoiding any coordination in most cases. This work builds on the ideas of escrow transactions [18] and the demarcation protocol [9].

We implemented the Bounded Counter in Antidote, and integrated it with the Antidote transactional model, Transactional Causal+ Consistency. This allows to provide much stronger guarantees to applications by enforcing numeric invariant. We are still working on submitting this novel approach for publication.

4.2.2 Explicit Consistency

Our second work proposes a general approach for maintaining applications invariants, based on *Explicit Consistency* [3]. *Explicit Consistency* is a novel consistency semantics for replicated systems. The high level idea is to let programmers define the application-specific correctness rules that should be met at all times. These rules are defined as invariants over the database state.

Given the invariants expressed by the programmer, we propose a methodology for enforcing explicit consistency that has three steps: (i) detect the sets of operations that may lead to invariant violation when executed concurrently (we call these sets *I-offender sets*); (ii) select how to handle *I-offender sets*, by selecting either violation-avoidance or invariant-repair techniques; (iii) instrument the application code to use the selected mechanism on top of a weakly consistent database system.

4.2.2.1 Invariant Violation Avoidance For avoiding that the concurrent execution of operations will violate invariants, we have proposed a reservation system comprising the following techniques.

UID generator: A very common invariant is uniqueness of identifiers [15]. This problem can be easily solved, without coordination, by statically splitting the space of identifiers per replica. Indigo provides this service by appending a replica-specific suffix to a locally-unique identifier.

Multi-level lock reservation: The multi-level lock reservation (or simply multi-level lock) is our base mechanism to restrict the concurrent execution of operations that can break invariants. A multi-level lock can provide the following rights: *(i) shared forbid*, giving the shared right to forbid some action to occur; *(ii) shared allow*, giving the shared right to allow some action to occur; *(iii) exclusive allow*, giving the exclusive right to execute some action.

When a replica holds one of the above rights, no other replica holds rights of a different type. For instance, if a replica holds a *shared forbid*, no other replica has any form of *allow*. For an *I-offender set*, we can use a multi-level lock reservation for allowing either the execution of one operation or the other for all clients. The *exclusive allow* right can be used to assign the right to a single client.

Multi-level mask reservation: For invariants of the form $P_1 \vee P_2 \vee \ldots \vee P_n$, the concurrent execution of any pair of operations that makes two different predicates

false may lead to an invariant violation if all other predicates were originally false. A multi-level mask reservation, which can be seen as a vector of multi-level locks, can efficiently control the execution of operations.

When a replica obtains a *shared allow* right in one entry, it must obtain a *shared forbid* right in some other entry. For example, an operation that may make P_i false needs to obtain the *shared allow* right on the ith entry and a *shared forbid* right on an entry j for which the predicate is true. At runtime, to find an entry to forbid, it is only necessary to evaluate the current value of the predicate associated with each entry that can be locked.

Escrow reservation: For numeric invariants of the form $x \geq k$, we include an escrow reservation for allowing some decrements to execute without coordination [18]. This is similar to the Bounded Counter described before.

We have implemented a variant called *escrow reservation for conditions* that can be used to control the number of elements that verify a given condition. This addresses the situations where the same element can be removed twice, which could lead to the "leak" of rights.

Partition lock reservation: For some invariants, it is desirable to have the ability to reserve part of a partitionable resource. For example, consider the invariant that forbids two appointment in a calendar to overlap in time. Two operations that schedule different appointments will break the invariant if the time periods overlap. Using a multi-level lock, it would be necessary to obtain an *exclusive allow* right for executing any operation to schedule a new tournament.

However, no invariant violation arises if the time periods of concurrent operations do not overlap. To address this case, we provide a partition lock that allows a replica to obtain an *exclusive lock* on an interval of real values.[1] Replicas can obtain locks on multiple intervals, given that no two intervals reserved by different replicas overlap.

Using Reservations Our static analysis outputs *I-offender sets* and the corresponding invariant violated. A programmer, electing to use the conflict avoidance approach, must select the type of reservation to be used to avoid invariant violations. Figure 1 presents a default mapping between types of invariants and the corresponding reservations. Conservatively, it is always possible to resort to multi-level locks to enforce any invariant, at the expense of admissible concurrency, as discussed earlier. In the context of WP4, it has been shown how to prove that a system that uses a reservation system consisting only of multi-level locks preserves a given invariant by using the CISE logic [12].

In this period, we have worked on improving the CISE tool, which help programmers exploiting explicit consistency. The CISE tool is a static analysis tool for proving integrity invariants of applications using databases with hybrid consistency models. The tool helps a programmer to find minimal consistency guarantees sufficient for application correctness.

With the feedback provided by the static analysis of his program, a programmer must decide which reservations will be used to restrict concurrency in a way that invariants are not violated. Each operation is extended to acquire the appropriate

[1] Partition locks are a simplified version of partitionable objects [21] and slot reservations [19].

Invariant type	Formula (example)	Reservation
Numeric	$x < K$	Escrow(x)
Referential	$p(x) \Rightarrow q(x)$	Multi-level lock
Disjunction	$p_1 \vee \ldots \vee p_n$	Multi-level mask
Overlapping	$t(s_1, e_1) \wedge t(s_2, e_2) \Rightarrow s_1 \geq e_2 \vee e_1 \leq s_2$	Partition lock
Default	—	Multi-level lock

Table 1: Default mapping from invariants to reservations.

rights before executing its code, and to release appropriate rights afterwards.

4.2.2.2 Invariant Repair The results of our evaluation with the invariant violation avoidance approach show most operation can execute locally. However, in some cases, the execution of some operations requires obtaining reservations from remote site. This leads to high latency, with operation execution taking even longer than in strong consistency settings, and lower availability as fault may make it impossible to obtain the necessary reservations.

We now show how to support invariant repair. In our presentation we use the example of referential integrity, but our approach can address other invariants.

Running example We chose referential integrity as running example due to its importance in relational databases and concurrent programming in general. We consider a toy database composed of two entities, A and B. We assume, without loss of generality, that each entity has a single attribute. There is a one-to-many relationship $R_{a \to b}$ from elements of A to elements of B.

Consider the implementation of this example using an object-relational mapping approach, where entities are modeled as two distinct sets and the relationship between them are modeled by a third set of pairs $(a, b) : a \in A, b \in B$.

We assume that the storage system stores each set in separate objects and that it provides causal consistency and atomic updates across multiple objects.

The integrity constraint of this model is broken when $\exists(a, b) \in R_{a \to b} : a \notin A \vee b \notin B$, i.e. there is a relationship between entity a and b, but one or both of them do not exist. We consider, for simplicity, that the application is correct under strong consistency, i.e. any sequential execution of the program does not violate the invariant. An invariant violation only occurs when a client issues an operation to create a new relation (a, b) while another client issues an operation to remove a or b from A or B, respectively.

Better safe than sorry To allow fast execution without constraining concurrency, every replica must be able to reply to a request without depending on remote state. Under these circumstances it is not possible to avoid concurrent executions that might leave the database in an inconsistent state. Since detecting conflicts and fixing invalid database state is expensive, we propose solving conflicts beforehand instead, so that operation execution is always safe. The idea is that an operation can have extra effects in order to avoid generating an invalid state when replicas are reconciled. As a trade-off, the semantics of operations that are implemented this

way is limited, but, as we show next, interesting semantics can be provided with proper use of convergence rules.

In the next section we describe two alternative solutions for the described problem. In the first solution we rely exclusively on existing CRDT semantics, while in the second solution we devise a new convergence rule for concurrent operations to implement an alternative semantics.

Adding missing elements When a new element (a, b) is added to $R_{a \to b}$, the operation that adds this element to the relations set must ensure that $a \in A$ and $b \in B$ to preserve referential integrity. These elements might be removed concurrently at other replicas leading to an invariant violation after replicas reconcile. To avoid this conflict, we modify the operation that adds (a, b) to $R_{a \to b}$ to also add a to A and b to B, atomically, and set the convergence rule of each set to use a Add-Wins policy. This policy ensures that if an add and remove operations execute concurrently for the same element, then the element will be present in the set, cancelling the effects of the remove operation. The consequence of our modifications is that any concurrent remove for elements a or b will be cancelled by the additional effect of the operation that adds (a, b). Therefore, at any time, each replica is in a consistent state.

Ensuring that elements are removed In the previous solution, whenever the conflicting operations execute, the operation that adds the relation takes precedence over the remove operations. We might want the opposite semantics, i.e. that whenever a remove operation for a or b is issued, we want to cancel any concurrent operation that adds and element to $R_{a \to b}$ containing one of those values. This example is different from the previous and cannot be solved in the same way, because we do not know the possible pairs containing a or b that might be added to the set, and it would be too expensive to consider the whole domain of A or B. To this end, we had to design a new set CRDT that prevents concurrently adding elements to a set that match a given criteria, without specifying their values.

The intuition behind this new set is to provide a special *touch(Predicate p)* operation that accepts a predicate that specifies which elements we want to prevent adding concurrently to the set. This way, whenever we execute a remove operation for elements a or b, we also execute a *touch* in $R_{a \to b}$ that prevent the addiction of any pair matching $(a, *)$ or $(*, b)$, where $*$ means any element.

Repair conflicts only when necessary Now lets consider that the number of relations between A and B is limited. If the size of $R_{a \to b}$ exceeds its limit, due to concurrent operations that add elements to this set, we would like to remove the extra elements from the set. But when a replica receives an update that makes the set bigger than the allowed, it is still not possible to determine if there was an invariant violation, because some concurrent operation, that has not been delivered locally, may remove the exceeding elements. The system cannot wait for all concurrent operations to arrive before deciding which elements to remove, however it can postpone this decision, to make it more accurate.

While the client does not read $R_{a \to b}$ the database is not showing any inconsistency, therefore we do not need to restore the size of the set until it is requested for

read. At that point, the replica must check if the size of the set was exceeded and use an arbitrary convergence policy to rule out exceeding elements, even if there is some missing concurrent operations. The decision must be durable, so that other replicas are able to take the same decision when they coordinate with this replica. The strategy allows that some elements that are visible at a moment to be automatically removed later, as consequence of some concurrent operation that was not yet delivered, but this is a trade-off in semantics to preserve availability.

This repair strategy poses new consistency problems to the programmer, since some elements might be removed in the background, leading to the same problems discussed before to maintain integrity across multiple objects.

Tools for programming Weak Consistency In the previous section we have seen how to preserve referential integrity in applications developed on top of weak consistency. Even though the transformations to the operations are easy to explain, it might be difficult for the average programmers to devise them. For this reason, we are also working on tools that can ease identifying invariant violations in applications and proposes possible solutions.

We are building a tool that, given the specification of an application's operations and invariants, identifies conflicts that might arise due to concurrent executions and proposes transformations to the operations to fix them, without strengthening the consistency model employed. For identifying conflicts, it is possible to rely on CISE logic [12, 17] or use the tool included in Indigo [3]. We have been extending this later tool to propose transformations to the operations like the ones we described before.

We are currently evaluating the invariant repair approach, which we expect to submit for publication in late April. Our results show that this approach can address the problem of operations with high latency at the cost of typically minor increases in the latency of other operations.

4.3 Extensions to Works Previously Reported

A number of works that started being developed in the context of Tasks 3.1 and 3.2 continued during this period, some of them leading to publications. We now briefly overview the most relevant work, some of them being developed jointly with other Work Packages.

4.3.1 Quality-of-data

In systems where data is weakly consistent, it might be interesting for an application to have information about how divergent the data is. In deliverable 3.2 we have reported a number of proposal for providing this kind of information. During this period, our work related with this topic focused on two main directions.

First, we continued the work on defining and evaluating algorithms for propagating the necessary information to be able to provide divergence information in the replicas. This work is being done relying on simulation, as discussed in the previous report. For the different divergence metrics proposed, we are currently focusing on the following: *Elapsed Time Since Last Sync*; *# of Operations That*

Are Known to be Missing; *Estimated # of Missing Operations per Replica*; and *Probability of staleness of each replica*. The selection of the metrics to focus on was done based on feedback received about how interesting the different metrics could be. We currently expect to have results for a submission before the Summer.

Second, in the context of the FMK-inspired application being developed, we have studied how divergence metrics could be used to improve the application and how to implement this divergence metrics in Antidote.

The base divergence metric that will be used will be *Elapsed Time Since Last Sync* to provide information on the potential staleness of the local replica. This divergence metric can be applied to every data object and its implementation in Antidote is rather straightforward, as the replication protocols already propagate the necessary information.

The second divergence metric we will be using is the *Estimated # of Missing Operations per Replica*, which we will apply to the objects storing the receipts for a given pharmacy. Unlike other data object, for which updates are infrequent, we expect this object to receive a more regular stream of updates, which will allow to infer the evolution model of data.

In the next period, we expect to report on the implementation of this divergence metrics in Antidote and on its evaluation in the context of the FMK-inspired application as part of the final project experiments.

4.3.2 Delta State-based CRDTs

Delta State-based CRDTs provide an efficient mechanism for synchronizing state-based CRDTs, by propagating only deltas among replicas. In this period, we have proposed additional CRDT designs [6], which are publicly available [2]. Additionally, we have worked on a journal submission of this work – section 7.3.

An extension to the delta state-based model was also proposed [20]. This proposal extends delta-based CRDTs by allowing to query a CRDT for the delta between the current version and a previous version identified by a version vectors. This allows to improve the first synchronisation step, which is specially important in settings where a replica changes its synchronization peers often, as it is the case when using gossip protocols.

4.3.3 Efficient Support of Large CRDTs in Riak

The deployment of CRDTs in Riak 2.0, and their use in multiple production scenario has led to performance problems due to unanticipated usage, such as clients storing huge number of elements inside sets.

Basho continued to address this problem, by redesigning the support for CRDTs inside Riak. This work explores ideas of Delta State-based CRDTs and will be reported in more detail elsewhere at M36.

[2] `https://github.com/CBaquero/delta-enabled-crdts`

4.3.4 Causality with partial knowledge

We have been work on designing efficient techniques for tracking causality among operations. As discussed in previous reports, the key challenge relates with the size of metadata necessary to track causality, which is even more challenging when dealing with resource-limited devices.

We have proposed an approach that keeps partial knowledge on causality and we have shown how to use this partial information [13]. Out work provides the first proof-theoretic causality modelling for distributed partial knowledge. We show that the partial knowledge gives rise to a weaker model than classical causality. We provide rules for offline deduction about causality. We define two notions of bisimilarity between devices, with which we prove two important results. Namely, no matter the order of addition/removal, two devices deduce similarly about causality so long as: (1) the same causal information is fed to both; (2) they start bisimilar and erase the same causal information.

In the next period we will study how the proposed model can be used in the works developed in the context of the project, in particular Antidote.

4.3.5 Other works

During this period, the following results submitted in the previous periods were published:

- The work on Conflict-free Partially Replicated Data Structure, where a CRDT can be partitioned in multiple *particles* was published in CloudCom'2015 [10].
- We have published a paper on logical clocks, which includes Dotted Version Vectors and the Dotted Vector Clock variant in ACM Queue [7] and as a practice paper at ACM Comm. of the ACM [8].

5 Publications

The work performed in the context of WP3 and in collaboration with other work packages has led to several papers. The following papers have been published during this period:

- [12] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pages 371– 384, New York, NY, USA, 2016. ACM. ($ 6.1)
- [10] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free Partially Replicated Data Types. In Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015). IEEE, Nov 2015. ($ 6.2)
- [7] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Queue, January 2016. ACM. ($ 6.3)

The following papers have been accepted and will be published during the next period:

- [8] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Commun. ACM (to appear), April 2016. ACM. ($ 6.4)
- [5] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. Geo-replication: Fast if possible, consistent if necessary. IEEE Data Engineering Bulletin (to appear), 2016. ($ 6.5)
- [4] Valter Balegas, Sérgio Duarte, Carla Ferreira, Nuno Preguiça, and Rodrigo Rodrigues. Making Weak Consistency Great Again. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. ($ 6.6)
- [6] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. The problem with embedded CRDT counters and a solution. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. ($ 6.7)
- [20] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. ($ 6.8)
- [17] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE Tool: Proving Weakly-Consistent Applications Correct. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016. ($ 6.9)

The following paper are under submission or being prepared for submission.

- [22] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. Access control for weakly consistent cloud-storage systems. Submitted for publication, 2016. ($ 7.1)
- [16] Christopher Meiklejohn. Loquat: A partially replicated, secure, broadcast protocol for edge computation. Submitted for publication, 2016. ($ 7.2)
- [1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. Submitted for publication, 2016. ($ 7.3)
- [13] Seyed H. Haeri (Hossein), Peter Van Roy, Carlos Baquero, and Christopher Meiklejohn. Deduction with partial knowledge about causality. Submitted for publication, 2016. ($ 7.4)

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. Submitted for publication, 2016.

[2] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguica. Extending eventually consistent cloud databases for enforcing numeric invariants. In *Reliable Distributed Systems (SRDS), 2015 IEEE 34th Symposium on*, pages 31–36, Sept 2015.

[3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 6:1–6:16, New York, NY, USA, 2015. ACM.

[4] Valter Balegas, Sérgio Duarte, Carla Ferreira, Nuno Preguiça, and Rodrigo Rodrigues. Making Weak Consistency Great Again. Presented at the Second Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC '16, 2016.

[5] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. Geo-replication: Fast if possible, consistent if necessary. *IEEE Data Engineering Bulletin (to appear)*, 2016.

[6] Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. The problem with embedded CRDT counters and a solution. In *Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear)*, PaPoC '16. ACM, 2016.

[7] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. *Queue*, January 2016.

[8] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. *Commun. ACM*, April 2016.

[9] Daniel Barbará-Millá and Hector Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[10] Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict-free partially replicated data types. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, Nov 2015.

[11] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li,

Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8:1–8:22, August 2013.

[12] Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 371–384, New York, NY, USA, 2016. ACM.

[13] Seyed H. Haeri (Hossein), Peter Van Roy, Carlos Baquero, and Christopher Meiklejohn. Deduction with partial knowledge about causality. Submitted for publication, 2016.

[14] Joao Leitao, Jose Pereira, and Luis Rodrigues. Epidemic broadcast trees. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems*, SRDS '07, pages 301–310, Washington, DC, USA, 2007. IEEE Computer Society.

[15] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 265–278, Berkeley, CA, USA, 2012. USENIX Association.

[16] Christopher Meiklejohn. Loquat: A partially replicated, secure, broadcast protocol for edge computation. Submitted for publication, 2016.

[17] Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE Tool: Proving Weakly-Consistent Applications Correct. In *Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear)*, PaPoC '16. ACM, 2016.

[18] Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986.

[19] Nuno Preguiça, J. Legatheaux Martins, Miguel Cunha, and Henrique Domingos. Reservations for conflict avoidance in a mobile database system. In *Proceedings of the 1st International Conference on Mobile Systems, Applications and Services*, MobiSys '03, pages 43–56, New York, NY, USA, 2003. ACM.

[20] Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In *Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear)*, PaPoC '16. ACM, 2016.

[21] G. D. Walborn and P. K. Chrysanthis. Supporting semantics-based transaction processing in mobile database applications. In *Proceedings of the 14TH Symposium on Reliable Distributed Systems*, SRDS '95, pages 31–, Washington, DC, USA, 1995. IEEE Computer Society.

[22] Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. Access control for weakly consistent cloud-storage systems. Submitted for publication, 2016.

6 Published papers

6.1 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, pages 371– 384, New York, NY, USA, 2016. ACM.

'Cause I'm Strong Enough: Reasoning about Consistency Choices in Distributed Systems

Alexey Gotsman
IMDEA Software Institute, Spain

Hongseok Yang
University of Oxford, UK

Carla Ferreira
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa, Portugal

Mahsa Najafzadeh
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Marc Shapiro
Sorbonne Universités, Inria,
UPMC Univ Paris 06, France

Abstract

Large-scale distributed systems often rely on replicated databases that allow a programmer to request different data consistency guarantees for different operations, and thereby control their performance. Using such databases is far from trivial: requesting stronger consistency in too many places may hurt performance, and requesting it in too few places may violate correctness. To help programmers in this task, we propose the first proof rule for establishing that a particular choice of consistency guarantees for various operations on a replicated database is enough to ensure the preservation of a given data integrity invariant. Our rule is modular: it allows reasoning about the behaviour of every operation separately under some assumption on the behaviour of other operations. This leads to simple reasoning, which we have automated in an SMT-based tool. We present a nontrivial proof of soundness of our rule and illustrate its use on several examples.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords Replication; causal consistency; integrity invariants

1. Introduction

To achieve availability and scalability, many modern distributed systems rely on *replicated databases*, which maintain multiple *replicas* of shared data. Clients can access the data at any of the replicas, and these replicas communicate changes to each other using message passing. For example, large-scale Internet services use data replicas in geographically distinct locations, and applications for mobile devices keep replicas locally to support offline use. Ideally, we would like replicated databases to provide *strong consistency*, i.e., to behave as if a single centralised node handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [2, 24].

For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [47]. In these databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately returns to the client; the *effect* of the operation is propagated to the other replicas only *eventually*. This may lead to *anomalies*—behaviours deviating from strong consistency. One of them is illustrated in Figure 1(a). Here Alice makes a post while connected to a replica r_1, and Bob, also connected to r_1, sees the post and comments on it. After each of the two operations, r_1 sends a message to the other replicas in the system with the update performed by the user. If the messages with the updates by Alice and Bob arrive to a replica r_2 out of order, then Carol, connected to r_2, may end up seeing Bob's comment, but not Alice's post it pertains to. The *consistency model* of a replicated database restricts the anomalies that it exhibits. For example, the model of *causal consistency* [33], which we consider in this paper, disallows the anomaly in Figure 1(a), yet can be implemented without any synchronisation. The model ensures that all replicas in the system see *causally dependent* events, such as the posts by Alice and Bob, in the order in which they happened. However, causal consistency allows different replicas to see causally *independent* events as occurring in different orders. This is illustrated in Figure 1(b), where Alice and Bob concurrently make posts at r_1 and r_2. Carol, connected to r_3 initially sees Alice's post, but not Bob's, and Dave, connected to r_4, sees Bob's post, but not Alice's. This outcome cannot be obtained by executing the operations in any total order and, hence, deviates from strong consistency.

Such anomalies related to the ordering of actions are often acceptable for applications. What is not acceptable is to violate crucial well-formedness properties of application data, called *integrity invariants*. Consistency models that do not require any synchronisation are often too weak to ensure these. For example, consider a toy banking application where the database stores the balance of a single account that clients can make deposits to and withdrawals from. In this case, an integrity invariant may require the account balance to be always non-negative. Consider the database compu-

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

POPL'16, January 20–22, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-3549-2/16/01...
http://dx.doi.org/10.1145/2837614.2837625

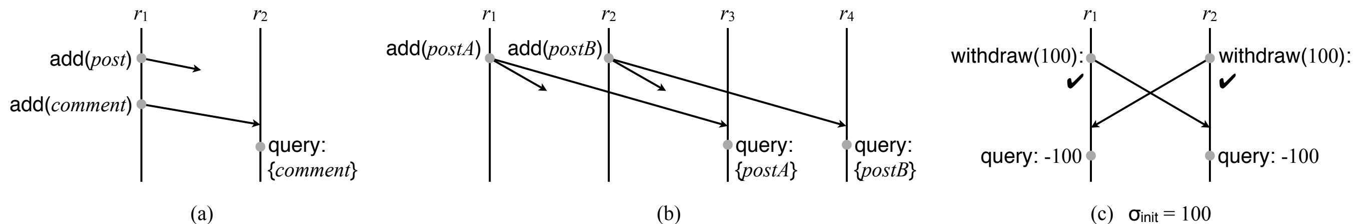


Figure 1. Illustrations of replicated database computations.

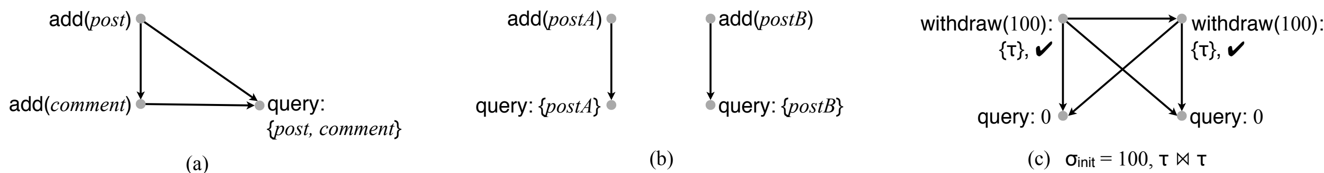


Figure 2. Examples illustrating Definition 1. We omit return values when they are $\perp$ and token sets when they are empty.

tation in Figure 1(c), allowed by causal consistency. Initially all replicas store the same balance of 100. Alice and Bob, connected to r_1 and r_2, both withdraw 100, thinking that there are sufficient funds available. Once the two replicas exchange the updates, the balance becomes -100, violating the integrity invariant. To ensure the integrity invariant in this example, we have to introduce synchronisation between replicas, and, since synchronisation is expensive, we would like to introduce it sparingly. To allow this, some research [9, 32, 42, 44] and commercial [6, 10, 35] databases now provide *hybrid* consistency models that allow the programmer to request stronger consistency for certain operations and thereby introduce synchronisation. For example, a consistency model may execute some operations under causal consistency, and some under strong consistency [32]. To preserve the integrity invariant in our banking application when using this model, only withdrawal operations need to use strong consistency, and hence, synchronise to ensure that the account is not overdrawn; deposit operations may use causal consistency and hence proceed without synchronisation. Requesting stronger consistency in hybrid models is similar to the use of fences in weak memory models of shared-memory multiprocessors and programming languages [11] (see §7 for a comparison).

Even though hybrid consistency models allow the programmer to fine-tune consistency level, using these models effectively is far from trivial. Requesting stronger consistency in too many places may hurt performance and availability, and requesting it in too few places may violate correctness. Striking the right balance requires the programmer to reason about the application behaviour on the subtle semantics of the consistency model, taking into account which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness. This difficulty is compounded by the perennial challenge of reasoning about concurrency, present even with strong consistency—having to consider the huge number of possible interactions between concurrently executing operations.

To help programmers exploit hybrid consistency models, we propose the first proof rule and tool for proving integrity invariants of applications using replicated databases with a range of hybrid models. In more detail, our first contribution is a generic hybrid consistency model (§2) that is flexible enough to encode a variety of consistency models for replicated databases proposed in the literature [9, 32, 33, 42]. It guarantees causal consistency by default and allows the programmer to additionally specify which pairs of operations may not execute without synchronisation by means of a special *conflict relation*. For example, to ensure the non-negativity of balances in the banking application, the conflict relation may require any pair of withdrawals to synchronise, so that one of them is aware of the effect of the other. This is equivalent to executing withdrawals under strong consistency. In general, different instances of the conflict relation correspond to different interfaces for strengthening consistency proposed in the literature. Our proof rule is developed for the generic consistency model and, hence, applies to existing models that can be represented as its instantiations. We specify our consistency model formally (§3) using the approach previously proposed for specifying variants of eventual consistency [15]. In this approach, a database computation is denoted by a partial order on client operations, representing causality, and the conflict relation imposes additional constraints on this order.

Our next, and key, technical contribution is a proof rule for showing that a set of operations preserves a given integrity invariant when executed on our consistency model with a given choice of conflict relation (§4). For example, we can prove that withdrawals and deposits preserve the non-negativity of balances when executed with the conflict relation described above. To avoid explicit reasoning about all possible interactions between operations, our proof rule is *modular*: it allows us to reason about the behaviour of every operation separately under some assumption on the behaviour of other operations, which takes into account the conflict relation. In this way, our proof rule allows the programmer to reason precisely about how strengthening or weakening consistency of certain operations affects correctness.

The modular nature of our proof rule allows it to reason in terms of states of a single database copy, just like in proof rules for strongly consistent shared-memory concurrency. We have proved that this simple reasoning is sound, despite the weakness of the consistency model (§5). As part of this proof we have identified a more general *event-based* rule that reasons directly in terms of partial orders on events representing database computations, instead of database states that these events lead to. The soundness of the original *state-based* rule is proved by deriving it from the event-based one. In this way, the event-based rule explicates the reasons for the soundness of the state-based rule.

We have also developed a prototype tool that automates our proof rule by reducing checking its obligations to SMT queries (§6). Using the tool, we have verified several example applications that require strengthening consistency in nontrivial ways. These

include an extension of the above banking application, an online auction service and a course registration system. In particular, we were able to handle applications using *replicated data types* (aka CRDTs [40]), which encapsulate policies for automatically merging the effects of operations performed without synchronisation at different replicas. The fact that we can reduce checking the correctness properties of complex computations in our examples to querying off-the-shelf SMT tools demonstrates the simplicity of reasoning required by our approach.

2. Consistency Model, Informally

We start by presenting our generic consistency model. Even though this model is not implemented in its full generality by an existing database, it can encode a variety of models that have in fact been implemented. In this section we present the programming interface of our consistency model and describe its semantics informally, from an operational perspective. We give a formal semantics in §3.

2.1 Causal Consistency and Its Implementation

Our hybrid model guarantees at least *causal consistency* [33], already mentioned in §1. We therefore start by presenting informally how a typical implementation of a causally consistent database operates. Let State be the set of possible states of the data managed by the database system. We denote states by σ and let σ_{init} be a distinguished initial state. Applications define a set of operations $\mathsf{Op} = \{o, \dots\}$ on the data and interact with the database by issuing these operations. For simplicity, we assume that an operation always terminates and returns a single value from a set Val. We use a value $\bot \in \mathsf{Val}$ to model operations that return no value. We do not consider operation parameters, since these can be part of the operation name.

The database implementation consists of a set of replicas, each maintaining a complete copy of the database state; we identify replicas by $r_1, r_2, \dots$ For the purposes of the informal explanation, we assume that replicas never fail. A client operation is initially executed at a single replica, which we refer to as its *origin* replica. At this replica, the execution of the operation is not interleaved with that of others. This execution updates the replica state deterministically, and immediately returns a value to the client. After this, the replica sends a message to all other replicas containing the *effect* of the operation, which describes the updates done by the operation to the database state. The replicas are guaranteed to receive the message at most once. Upon receipt, the replicas apply the effect to their state.

In this paper, we abstract from a particular language in which operations may be written and assume that their semantics is given by a function

$$\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to (\mathsf{Val} \times (\mathsf{State} \to \mathsf{State}))). \quad (1)$$

To aid readability, for $o \in \mathsf{Op}$ we write $\mathcal{F}_o$ instead of $\mathcal{F}(o)$ and let

$$\forall o, \sigma.\ \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathsf{eff}}(\sigma)).$$

Given a state σ of o's origin replica, $\mathcal{F}_o^{\mathsf{val}}(\sigma) \in \mathsf{Val}$ determines the return value of the operation and $\mathcal{F}_o^{\mathsf{eff}}(\sigma) \in \mathsf{State} \to \mathsf{State}$ its effect. The latter is a function, to be applied by every replica to its state to incorporate the operation's effect: immediately at the origin replica, and after receiving the corresponding message at all other replicas.

For example, states in the toy banking application of §1 are integers, representing the account balance: $\mathsf{State} = \mathbb{Z}$. We define

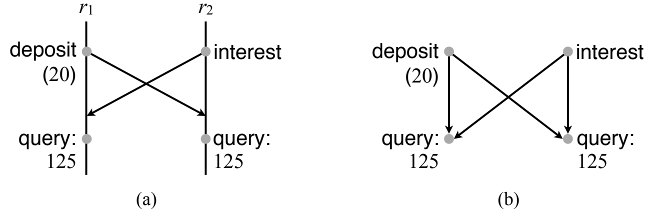


Figure 3. (a) An illustration of a database computation; (b) the corresponding execution of Definition 1. We assume $\sigma_{\mathsf{init}} = 100$.

the semantics of operations for depositing an amount $a > 0$, accruing a 5% interest and querying the balance:

$$\begin{aligned} \mathcal{F}_{\mathsf{deposit}(a)}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + a)); \\ \mathcal{F}_{\mathsf{interest}}(\sigma) &= (\bot, (\lambda\sigma'.\, \sigma' + 0.05 * \sigma)); \\ \mathcal{F}_{\mathsf{query}}(\sigma) &= (\sigma, \mathsf{skip}), \end{aligned} \quad (2)$$

where $\mathsf{skip} = (\lambda\sigma'.\, \sigma')$. Figure 3(a) illustrates a database computation involving these operations. Note that interest first computes the interest $0.05 * \sigma$ based on the balance σ at the origin replica; its effect then adds the resulting amount to the balance at each replica. In particular, in Figure 3(a) interest at r_2 does not take into account the deposit made at r_1. This behaviour is the price to pay for avoiding synchronisation between replicas. The good news is that, once the replicas r_1 and r_2 exchange the effects of deposit and interest, they converge to the same balance, which is returned by the query operations.

Such convergence is not guaranteed for arbitrary operations. For example, we could implement interest so that its effect multiplied the balance by 1.05 at each replica where it is applied:

$$\mathcal{F}_{\mathsf{interest}}^{\mathsf{eff}}(\sigma) = (\lambda\sigma'.\, (1.05 * \sigma')). \quad (3)$$

In the scenario in Figure 3(a), this would lead the query operations to return different values, 126 at r_1 and 125 at r_2. In this case, even after all messages are delivered, replicas end up in different states. This is undesirable for database users: we would like the implementation to be *convergent*, i.e., such that two replicas that see the same set of operations are in the same state. In particular, if users stop performing updates to the database, then once all outstanding messages are delivered, all replicas should reach the same state [47]. To ensure convergence, for now we require that the effects of all operations commute (we relax this condition slightly in §2.2):

$$\forall o_1, o_2, \sigma_1, \sigma_2.\ \mathcal{F}_{o_1}^{\mathsf{eff}}(\sigma_1) \circ \mathcal{F}_{o_2}^{\mathsf{eff}}(\sigma_2) = \mathcal{F}_{o_2}^{\mathsf{eff}}(\sigma_2) \circ \mathcal{F}_{o_1}^{\mathsf{eff}}(\sigma_1). \quad (4)$$

For example, this condition holds of the effects defined by (2). The requirement of commutativity is not very taxing: as we elaborate in §6, to satisfy (4), programmers can exploit ready-made *replicated data types* (aka CRDTs [40]). These encapsulate commutative implementations of policies for merging concurrent updates to the database.

As we explained in §1, asynchronous operation processing may lead to anomalies, and causal consistency disallows some of them. It ensures that message propagation between replicas is *causal*: if a replica sends a message containing the effect of an operation o_2 after it sends or receives a message containing the effect of an operation o_1, then no replica will receive the message about o_2 before it receives the one about o_1. In this case we say that the invocation of o_2 *causally depends* on that of o_1. Causal propagation disallows the computation in Figure 1(a), but allows the one in Figure 1(b).

$$
\begin{aligned}
\mathsf{Token} &= \{\tau\} \\
\bowtie &= \{(\tau, \tau)\} \\
\mathcal{F}_{\mathsf{deposit}(a)}(\sigma) &= (\bot, (\lambda\sigma'.\,\sigma' + a), \emptyset) \\
\mathcal{F}_{\mathsf{interest}}(\sigma) &= (\bot, (\lambda\sigma'.\,\sigma' + 0.05 * \sigma), \emptyset) \\
\mathcal{F}_{\mathsf{query}}(\sigma) &= (\sigma, \mathsf{skip}, \emptyset) \\
\mathcal{F}_{\mathsf{withdraw}(a)}(\sigma) &= \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'.\,\sigma' - a), \{\tau\}) \\
&\quad\ \text{else } (\text{✗}, \mathsf{skip}, \{\tau\})
\end{aligned}
$$

Figure 4. Operation semantics for the banking application. Note that $a > 0$.

2.2 Strengthening Consistency

The guarantees provided by causal consistency are too weak to ensure certain integrity invariants. For example, in our banking application we would like the state at each replica to satisfy the invariant

$$I = \{\sigma \mid \sigma \geq 0\}. \tag{5}$$

To ensure this, an operation for withdrawing an amount $a > 0$ could check whether the account has sufficient funds and return ✓ or ✗ depending on the result:

$$\mathcal{F}_{\mathsf{withdraw}(a)}(\sigma) = \text{if } \sigma \geq a \text{ then } (\checkmark, (\lambda\sigma'.\,\sigma' - a)) \text{ else } (\text{✗}, \mathsf{skip}).$$

This is enough to maintain the invariant when all operations are processed at the same replica, but not when they are processed asynchronously at different replicas. This is illustrated by the computation in Figure 1(c), already explained in §1.

The problem in this example arises because two particular operations update the database concurrently, without being aware of each other. To address this, our consistency model allows the programmer to strengthen causal consistency by specifying explicitly which operations may not be executed in this way. Namely, the model is parameterised by a *token system* $\mathcal{T} = (\mathsf{Token}, \bowtie)$, consisting of a set of *tokens* Token and a symmetric *conflict relation* $\bowtie \subseteq \mathsf{Token} \times \mathsf{Token}$. Tokens are ranged over by τ and their sets, by T. For sets T_1 and T_2 of tokens we let $T_1 \bowtie T_2$ if there exists a pair of conflicting tokens coming from these sets: $\exists\tau_1 \in T_1.\,\exists\tau_2 \in T_2.\,\tau_1 \bowtie \tau_2$.

Each operation may acquire a set of tokens. To account for this, we redefine the type of $\mathcal{F}$ in (1) as

$$\mathcal{F} \in \mathsf{Op} \to (\mathsf{State} \to (\mathsf{Val} \times (\mathsf{State} \to \mathsf{State}) \times \mathcal{P}(\mathsf{Token}))) \tag{6}$$

and let

$$\forall o, \sigma.\, \mathcal{F}_o(\sigma) = (\mathcal{F}_o^{\mathsf{val}}(\sigma), \mathcal{F}_o^{\mathsf{eff}}(\sigma), \mathcal{F}_o^{\mathsf{tok}}(\sigma)).$$

Thus, $\mathcal{F}_o^{\mathsf{tok}}(\sigma) \in \mathcal{P}(\mathsf{Token})$ gives the set of tokens acquired by the operation o when executed in the state σ. Informally, our consistency model guarantees that operations that acquire tokens conflicting according to $\bowtie$ have to be causally dependent one way or another: the origin replica of one operation must have incorporated the effect of the other by the time the former operation executes. Ensuring this in implementations requires replicas to synchronise [9, 32].

In our consistency model, we can guarantee the preservation of invariant (5) in the banking application by defining operation semantics as in Figure 4. Thus, withdraw acquires a token τ conflicting with itself, and all other operations do not acquire any tokens. Then the scenario in Figure 1(c) cannot happen: one withdrawal would have to be aware of the other and would therefore fail. However, deposits and interest accruals can be causally independent with all operations, and replicas can therefore execute them without any synchronisation [9, 32]. In this example, the token τ is analogous to a mutual exclusion lock in shared-memory concurrency. Our proof method (§4) establishes that this use of the token is indeed sufficient to preserve the integrity invariant (5).

Since operations acquiring conflicting tokens have to be causally dependent, causal message propagation (§2.1) ensures that all replicas see such operations in the same order. This allows us to weaken (4) to require commutativity only for operations that do not acquire conflicting tokens:

$$
\begin{aligned}
\forall o_1, o_2, \sigma_1, \sigma_2.\, &(\mathcal{F}_{o_1}^{\mathsf{tok}}(\sigma_1) \bowtie \mathcal{F}_{o_2}^{\mathsf{tok}}(\sigma_2)) \vee \\
&(\mathcal{F}_{o_1}^{\mathsf{eff}}(\sigma_1) \circ \mathcal{F}_{o_2}^{\mathsf{eff}}(\sigma_2) = \mathcal{F}_{o_2}^{\mathsf{eff}}(\sigma_2) \circ \mathcal{F}_{o_1}^{\mathsf{eff}}(\sigma_1)).
\end{aligned} \tag{7}
$$

As we show in §3, this is sufficient to ensure the property of convergence that we introduced in §2.1. For example, the operations in Figure 4 satisfy (7). Furthermore, if all operations except query acquired the token τ, then we would be able to implement interest by the effect given by (3) without compromising convergence.

3. Formal Semantics

We now formally define the semantics of our consistency model, i.e., the set of all client-database interactions it allows. To keep the presentation as simple as possible, we define the semantics declaratively: our formalism does not refer to implementation-level concepts, such as replicas or messages, even though we do use these concepts in informal explanations. We build on an approach previously used to specify forms of eventual consistency [15]. Namely, our denotations of database computations consist of a set of events, representing operation invocations by clients, and a relation on events, describing abstractly how the database processes the corresponding operations.

Assume a countably infinite set Event of *events*, ranged over by e, f, g. A relation is a *strict partial order* if it is transitive and irreflexive. For a relation R we write $(e, f) \in R$ and $e \xrightarrow{R} f$ interchangeably.

DEFINITION 1. *Given a token system* $\mathcal{T} = (\mathsf{Token}, \bowtie)$*, an **execution** is a tuple* $X = (E, \mathsf{oper}, \mathsf{rval}, \mathsf{tok}, \mathsf{hb})$*, where:*

- E *is a finite subset of* Event*;*
- $\mathsf{oper} : E \to \mathsf{Op}$ *gives the operation whose invocation a given event denotes;*
- $\mathsf{rval} : E \to \mathsf{Val}$ *gives the return value of the operation;*
- $\mathsf{tok} : E \to \mathcal{P}(\mathsf{Token})$ *gives the set of tokens acquired by the operation;*
- $\mathsf{hb} \subseteq E \times E$*, called **happens-before**, is a strict partial order such that*

$$\forall e, f \in E.\, \mathsf{tok}(e) \bowtie \mathsf{tok}(f) \implies (e \xrightarrow{\mathsf{hb}} f \vee f \xrightarrow{\mathsf{hb}} e). \tag{8}$$

Operationally, each event represents an invocation of an operation at its origin replica. The applications of the operation's effect at other replicas are not recorded in an execution explicitly. Instead, the happens-before relation records causal dependencies between operations arising from such applications: $e \xrightarrow{\mathsf{hb}} f$ means that either the operations denoted by e and f were executed at the same replica in this order, or they were executed at different replicas and the message containing the effect of e had been delivered to the replica performing f before f was executed. Hence, if we have $e \xrightarrow{\mathsf{hb}} f$, then the effect of e is incorporated into the state to which f is applied and may influence its return value. We give examples of executions in Figures 2 and 3(b). The ones in Figures 2(b) and 3(b) model the computations of the database informally illustrated in Figures 1(b) and 3(a), respectively.

The transitivity of hb in Definition 1 reflects the guarantee of causal message propagation in implementations explained in

§2.1 [15]. For example, in the execution of Figure 2(a), the transitivity of hb mandates the edge between the addition of a post and the query (cf. Figure 1(a)). The condition (8) formalises the stronger consistency guarantee provided by tokens: operations acquiring conflicting tokens have to be causally dependent. For example, since the two withdraw operations in Figure 2(c) acquire a token τ with $\tau \bowtie \tau$, they have to be related by happens-before. Finally, we require executions to contain only finitely many events, because in this paper we are only concerned with safety properties of applications.

We write $\mathsf{Exec}(\mathcal{T})$ for the set of all executions over the token system $\mathcal{T}$. In the following, we denote components of X and similar structures as in $X.E$. We let X_{init} be the unique execution with $X_{\mathsf{init}}.E = \emptyset$.

We now define the semantics of our consistency model as the set of all executions $X \in \mathsf{Exec}(\mathcal{T})$ over a token system $\mathcal{T}$ whose return values $X.\mathsf{rval}$ and token sets $X.\mathsf{tok}$ are computed using $\mathcal{F}$ as informally described in §2. To define this set, we first let the *context* of an event e in an execution X be

$$\mathsf{ctxt}(e, X) = (E, (X.\mathsf{oper})|_E, (X.\mathsf{rval})|_E, (X.\mathsf{tok})|_E, (X.\mathsf{hb})|_E),$$

where $E = (X.\mathsf{hb})^{-1}(e)$ and $\cdot|_E$ is the restriction to events in E. Operationally-speaking, the context consists of those events whose effects have been incorporated into the state of the replica where the operation $X.\mathsf{oper}(e)$ executes; it is these events that influence the outcomes of e—the return value $X.\mathsf{rval}(e)$ and the token set $X.\mathsf{tok}(e)$. For example, the context of each of the query events in Figure 3(b) consists of the deposit and interest events. This reflects the events that the corresponding replica has seen before executing query in Figure 3(a).

It is technically convenient for us to initially formulate definitions without assuming effect commutativity (7). In this case, $X.\mathsf{rval}(e)$ and $X.\mathsf{tok}(e)$ are not determined by $\mathsf{ctxt}(e, X)$ uniquely. In operational terms, this is because the state that a replica will be in after seeing the events in $\mathsf{ctxt}(e, X)$ depends on the order in which the replica finds out about these events: although causal message propagation ensures that messages about causally dependent events in $\mathsf{ctxt}(e, X)$ will be delivered to the replica in the order consistent with $X.\mathsf{hb}$, messages about causally independent events may be delivered in arbitrary order. We therefore first define a function

$$\mathsf{eval}^{\dagger}_{\mathcal{F}} : \mathsf{Exec}(\mathcal{T}) \to \mathcal{P}(\mathsf{State})$$

that yields the *set* of all possible states that a replica may end up in after seeing the events in a given execution, such as $\mathsf{ctxt}(e, X)$. For an execution Y, we define $\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y)$ inductively on the size of $Y.E$. If $Y.E = \emptyset$, then $\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y) = \{\sigma_{\mathsf{init}}\}$. Otherwise,

$$\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y) = \{\mathcal{F}^{\mathsf{eff}}_{Y.\mathsf{oper}(e)}(\sigma')(\sigma) \mid e \in \mathsf{max}(Y) \wedge \sigma \in \mathsf{eval}^{\dagger}_{\mathcal{F}}(Y|_{Y.E-\{e\}}) \wedge \sigma' \in \mathsf{eval}^{\dagger}_{\mathcal{F}}(\mathsf{ctxt}(e, Y))\},$$

where

$$\mathsf{max}(Y) = \{e \in Y.E \mid \neg\exists f \in Y.E.\, (e, f) \in Y.\mathsf{hb}\}. \quad (9)$$

Thus, to compute $\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y)$ for a non-empty Y, we choose an hb-maximal event e in Y. Operationally, this is the event whose effect is incorporated last by the replica r whose state we are determining. We then pick a state σ that r could be in right before incorporating the effect of e. The set of such states is obtained by invoking $\mathsf{eval}^{\dagger}_{\mathcal{F}}$ on the execution $Y|_{Y.E-\{e\}}$, describing the events r knew about when it incorporated e. To determine the effect of e's operation, we pick a state σ' that the replica r' that generated e could be in at the time of this generation. The set of such states is computed by invoking $\mathsf{eval}^{\dagger}_{\mathcal{F}}$ on the execution $\mathsf{ctxt}(e, Y)$, describing the events that replica r' knew about when it generated e. Then the effect of e's operation is $\mathcal{F}^{\mathsf{eff}}_{Y.\mathsf{oper}(e)}(\sigma')$, and we determine the state of the replica r after e by applying this effect to the state σ: $\mathcal{F}^{\mathsf{eff}}_{Y.\mathsf{oper}(e)}(\sigma')(\sigma)$.

To illustrate $\mathsf{eval}^{\dagger}_{\mathcal{F}}$, consider the execution Y consisting of the deposit and interest events in Figure 3(b) and the operation semantics $\mathcal{F}$ in Figure 4. Recall that in this case $\sigma_{\mathsf{init}} = 100$. We can evaluate Y in two ways, corresponding to the orders in which replicas r_1, respectively r_2, apply the effects of the events in the computation in Figure 3(a):

$$\begin{aligned}\mathsf{eval}^{\dagger}_{\mathcal{F}}(Y) &= \{\mathcal{F}^{\mathsf{eff}}_{\mathsf{interest}}(\sigma_{\mathsf{init}})(\mathcal{F}^{\mathsf{eff}}_{\mathsf{deposit}(20)}(\sigma_{\mathsf{init}})(\sigma_{\mathsf{init}})),\\ &\quad\ \mathcal{F}^{\mathsf{eff}}_{\mathsf{deposit}(20)}(\sigma_{\mathsf{init}})(\mathcal{F}^{\mathsf{eff}}_{\mathsf{interest}}(\sigma_{\mathsf{init}})(\sigma_{\mathsf{init}}))\}\\ &= \{100 + 20 + 5, 100 + 5 + 20\} = \{125\}.\end{aligned}$$

Both ways of evaluation lead to the same outcome. This would not be the case if we used a function $\mathcal{F}'$ identical to $\mathcal{F}$, but with the effect of interest defined by (3), which violates (7). In this case,

$$\mathsf{eval}^{\dagger}_{\mathcal{F}'}(Y) = \{100 + 20 + 6, 100 + 5 + 20\} = \{126, 125\},$$

which corresponds to the diverging database computation we explained in §2.1.

We note that, for notational convenience, $\mathsf{eval}^{\dagger}_{\mathcal{F}}$ takes as a parameter a whole execution including return values (rval) and token sets (tok) associated with its events. However, the function as we defined it does not depend on these: the state is determined solely based on the operations performed (oper) and happens-before relationships among them (hb).

DEFINITION 2. *An execution $X \in \mathsf{Exec}(\mathcal{T})$ is* ***consistent*** *with $\mathcal{T}$ and $\mathcal{F}$, denoted $X \models \mathcal{T}, \mathcal{F}$, if*

$$\forall e \in X.E.\, \exists \sigma \in \mathsf{eval}^{\dagger}_{\mathcal{F}}(\mathsf{ctxt}(e, X)).\ (X.\mathsf{rval}(e) = \mathcal{F}^{\mathsf{val}}_{X.\mathsf{oper}(e)}(\sigma)) \wedge (X.\mathsf{tok}(e) = \mathcal{F}^{\mathsf{tok}}_{X.\mathsf{oper}(e)}(\sigma)).$$

We let $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) = \{X \mid X \models \mathcal{T}, \mathcal{F}\}$ be the set of executions allowed by our consistency model.

PROPOSITION 3.

$$\forall X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\, \forall e \in X.E.\, (\mathsf{ctxt}(e, X) \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})).$$

Operationally, $X \models \mathcal{T}, \mathcal{F}$ means that the outcomes in X can be produced by the database implementation sketched in §2 with some order of message delivery. The executions in Figures 2 and 3(b) are consistent with the parameters in Figure 4 or the expected semantics of operations on posts and comments. In particular, the execution in Figure 2(c) is consistent because the context of the right-hand-side withdraw includes the left-hand-side withdraw. Evaluating this context yields a zero balance, which causes the right-hand-side withdraw to generate skip as its effect.

LEMMA 4. *If $X \models \mathcal{T}, \mathcal{F}$, then $\mathsf{eval}^{\dagger}_{\mathcal{F}}(X)$ is a singleton set. Furthermore, so is $\mathsf{eval}^{\dagger}_{\mathcal{F}}(\mathsf{ctxt}(e, X))$ for any $e \in X.E$.*

The lemma shows that in Definition 2 it does not matter how we choose the order of evaluation in $\mathsf{eval}^{\dagger}_{\mathcal{F}}$. When viewed operationally, this independence implies the convergence property from §2.1: two replicas that see the same events will end up in the same state. The proof of Lemma 4, given in [26, §A], exploits properties (7) and (8). This proof is subtle because (7) does not require commutativity for the effects of pairs of operations that acquire conflicting tokens.

Motivated by Lemma 4, we define the evaluation of consistent executions

$$\mathsf{eval}_{\mathcal{F}} : \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \to \mathsf{State}$$

as follows: $\mathsf{eval}_{\mathcal{F}}(X)$ is the unique σ such that $\mathsf{eval}^{\dagger}_{\mathcal{F}}(X) = \{\sigma\}$.

To illustrate the flexibility of our consistency model, we show how it can represent some of the existing models; we provide more instantiations in §6.

Causal consistency [16, 33] is the baseline model we obtain without using any tokens: $\mathsf{Token} = \emptyset$ and $\forall o, \sigma. \mathcal{F}_o^{\mathsf{tok}}(\sigma) = \emptyset$. Then (8) is a tautology and (7) is equivalent to (4), so that all effects have to commute.

Sequential consistency [29] is a form of strong consistency and the strongest consistency model we can obtain from ours. It requires every operation to acquire a mutual exclusion token:

$$\mathsf{Token} = \{\tau\}; \quad \bowtie = \{(\tau, \tau)\}; \quad \forall o, \sigma. \mathcal{F}_o^{\mathsf{tok}}(\sigma) = \{\tau\}.$$

Then in any execution $X \in \mathsf{Exec}((\mathsf{Token}, \bowtie), \mathcal{F})$, the happens-before $X.\mathsf{hb}$ is total, and each event in X is aware of the effects of all events preceding it in $X.\mathsf{hb}$.

RedBlue consistency [32] is a hybrid consistency model that classifies operations as either *red* or *blue*: $\mathsf{Op} = \mathsf{Op}_r \uplus \mathsf{Op}_b$. Red operations are guaranteed sequential consistency, and blue operations, only causal consistency. To express this in our model, we again use a mutual exclusion token: $\mathsf{Token} = \{\tau\}$ and $\bowtie = \{(\tau, \tau)\}$. Red operations acquire τ, and blue operations acquire no tokens:

$$(\forall o \in \mathsf{Op}_r. \forall \sigma. \mathcal{F}_o^{\mathsf{tok}}(\sigma) = \{\tau\}) \wedge (\forall o \in \mathsf{Op}_b. \forall \sigma. \mathcal{F}_o^{\mathsf{tok}}(\sigma) = \emptyset).$$

Then red operations are totally ordered by happens-before, and blue ones are ordered only partially. The token assignment in our banking application (Figure 4) is an instance of the RedBlue consistency, where withdraw operations are red, and all others are blue.

Our framework cannot express some of common consistency models, such as prefix consistency [43], which is stronger than causal consistency. However, the framework could be adjusted to assume prefix consistency as a baseline following [17].

4. State-based Proof Rule

We consider the following verification problem: given a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$, prove that operations $\mathcal{F}$ maintain an integrity invariant $I \subseteq \mathsf{State}$ over database states. Formally, we establish that any execution consistent with $\mathcal{T}$ and $\mathcal{F}$ evaluates to a state satisfying I:

$$\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}_{\mathcal{F}}^{-1}(I).$$

By Proposition 3 this implies that the return value of every event in an execution $X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})$ can be obtained by applying its operation to a state satisfying I:

$$\forall e \in X.E. \exists \sigma \in I. (X.\mathsf{rval}(e) = \mathcal{F}_{X.\mathsf{oper}(e)}^{\mathsf{val}}(\sigma)).$$

For example, we show that any execution consistent with Figure 4 evaluates to a state satisfying the invariant (5). Hence, a query operation will always return a non-negative balance.

The key challenge of the above verification problem is the need to consider infinitely many executions consistent with $\mathcal{T}$ and $\mathcal{F}$. Our main technical contribution is the proof rule for solving this problem that avoids considering all such executions explicitly. Instead, the proof rule is *modular* in that it allows us to reason about the behaviour of every operation separately. Our proof rule is also *state-based* in that it reasons in terms of states obtained by evaluating parts of executions or, from the operational perspective, in terms of replica states.

We give our proof rule in Figure 5 and explain it from the operational perspective. The rule assumes that the invariant I holds of the initial database state σ_{init} (condition S1). Consider a computation of the database implementation from §2 and a state σ of a replica r at some point in this computation. The proof rule assumes that $\sigma \in I$ and aims to establish that executing any operation o at r will preserve the invariant I. This is easy if we only consider how o's effect changes the state of r, since this effect is applied to the state σ where it was generated:

$$\forall \sigma. (\sigma \in I \implies \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma) \in I). \tag{10}$$

The difficulty comes from the need to consider how o's effect changes the state of any other replica r' that receives it; see Figure 6(a). At the time of the receipt, r' may be in a different state σ', due to operations executed at r' concurrently with o. We can show that it is sound to assume that this state σ' also satisfies the invariant. Thus, to check that the operation o preserves the invariant when applied at any replica, it is sufficient to ensure

$$\forall \sigma, \sigma'. (\sigma, \sigma' \in I \implies \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') \in I). \tag{11}$$

However, establishing this without knowing anything about the relationship between σ and σ' is a tall order. In the bank account example, both $\sigma = 100$ and $\sigma' = 0$ satisfy the integrity invariant (5). Then $\mathcal{F}_{\mathsf{withdraw}(100)}^{\mathsf{eff}}(\sigma)(\sigma') = -100$, which violates the invariant. Condition (11) fails in this case because it does not take into account the tokens acquired by withdraw.

The proof rule in Figure 5 addresses the weakness of (11) by allowing us to assume a certain relationship between the state where an operation is generated (σ) and where its effect is applied (σ'), which takes into account the tokens acquired by the operation. To express this assumption, the rule uses a form of rely-guarantee reasoning [27]. Namely, it requires us to associate each token τ with a *guarantee* relation $G(\tau)$, describing all possible state changes that an operation acquiring τ can cause. Crucially, this includes not only the changes that the operation can cause on the state of its origin replica, but also any change that its effect causes at any other replica it is propagated to. We also have a guarantee relation G_0, describing the changes that can be performed by an operation without acquiring any tokens. Condition S2 requires the guarantees to preserve the invariant.

$\exists G_0 \in \mathcal{P}(\mathsf{State} \times \mathsf{State}), G \in \mathsf{Token} \to \mathcal{P}(\mathsf{State} \times \mathsf{State})$ such that

S1. $\sigma_{\mathsf{init}} \in I$

S2. $G_0(I) \subseteq I \wedge \forall \tau. G(\tau)(I) \subseteq I$

S3. $\forall o, \sigma, \sigma'. (\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*) \implies (\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}_o^{\mathsf{tok}}(\sigma))$

$$\overline{\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}_{\mathcal{F}}^{-1}(I)}$$

Figure 5. State-based proof rule for a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$. For $T \subseteq \mathsf{Token}$ we let $G(T) = \bigcup_{\tau \in T} G(\tau)$ and $T^{\perp} = \{\tau \mid \tau \in \mathsf{Token} \wedge \neg\exists\tau' \in T. \tau \bowtie \tau'\}$. We denote by R^* the reflexive and transitive closure of a relation R. For a relation $R \in \mathcal{P}(A \times B)$ and a predicate $P \in \mathcal{P}(A)$, the expression $R(P)$ denotes the image of P under R.

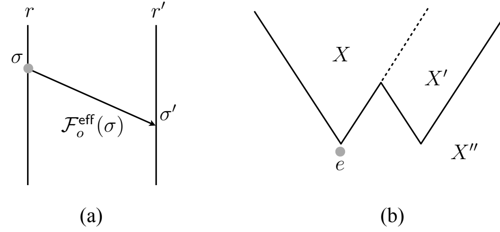


Figure 6. Graphical illustrations of (a) the state-based rule; and (b) the event-based rule.

Like (11), condition S3 considers an arbitrary state σ of o's origin replica r, assumed to satisfy the invariant I. The condition then considers any state σ' of another replica r' to which the effect of o is propagated. The conclusion of S3 requires us to prove that applying the effect $\mathcal{F}_o^{\mathsf{eff}}(\sigma)$ of the operation o to the state σ' satisfies the union of G_0 and the guarantees associated with the tokens $\mathcal{F}_o^{\mathsf{tok}}(\sigma)$ that the operation o acquires. By S2, this implies that the effect of the operation preserves the invariant. Condition S3 further allows us to assume that the state σ' of r' can be obtained from the state σ of r by applying a finite number of changes allowed by G_0 or the guarantees for *those tokens that do not conflict with any of the tokens acquired by the operation* o, i.e., $G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp})$. Informally, acquiring a token denies other replicas permissions to concurrently perform changes that require conflicting tokens.

We now use our proof rule to show that the operations in the banking application (Figure 4) preserve the integrity invariant (5). We assume that the initial state σ_{init} satisfies the invariant. The guarantees are as follows:

$$G(\tau) = \{(\sigma, \sigma') \mid 0 \leq \sigma' < \sigma\}; \quad G_0 = \{(\sigma, \sigma') \mid 0 \leq \sigma \leq \sigma'\}. \tag{12}$$

Since withdrawals acquire the token τ, the guarantee $G(\tau)$ for this token allows decreasing the balance without turning it negative; the guarantee G_0 allows increasing a non-negative balance. Then condition S2 is satisfied. We show how to check the condition S3 in the most interesting case of $o = \mathsf{withdraw}(a)$. Consider σ and σ' satisfying the premiss of S3:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*.$$

Since $\mathcal{F}_o^{\mathsf{tok}}(\sigma) = \{\tau\}$, we have $(\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp} = \emptyset$. Thus, $(\sigma, \sigma') \in G_0^*$. This and $\sigma \in I$ imply

$$0 \leq \sigma \leq \sigma'. \tag{13}$$

If $\sigma < a$, then $\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') = \sigma'$. Furthermore, $\sigma' \geq 0$ by (13). Thus, $(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) = (\sigma', \sigma') \in G_0$, which implies the conclusion of S3.

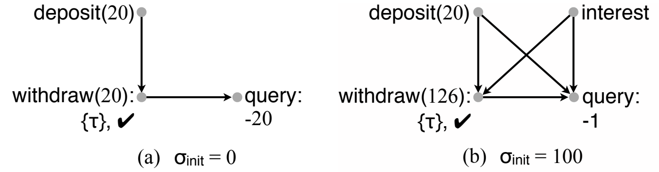
If $\sigma \geq a$, then $\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') = \sigma' - a$. Since $\sigma \leq \sigma'$ by (13), we have $\sigma' \geq a$. Thus, $(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) = (\sigma', \sigma' - a) \in G(\{\tau\})$, which implies the conclusion of S3. Operationally, in this case our proof rule establishes that, if there was enough money in the account at the replica where the withdrawal was made, then there will be enough money at any replica the withdrawal is delivered to. This completes the proof of our example.

In a banking application with multiple accounts, we could ensure non-negativity of balances by associating every account c with a token τ_c such that $\tau_c \bowtie \tau_c$, but $\tau_c \not\bowtie \tau_{c'}$ for another account c'. Thus, withdrawals from the same account would have to synchronise, while withdrawals from different accounts could proceed without synchronisation. Our proof rule easily deals with this generalisation by associating every token τ_c with a guarantee describing the changes to the corresponding account. As we elaborate in §6, the banking application we verify with the aid of our tool allows multiple accounts. There we also provide more complex examples of using our proof rule. For now, it is instructive to see how the proof rule is specialised for some of the simpler instantiations of our consistency model from §3.

Sequential consistency. Recall that for sequential consistency, $\bowtie\ = \{(\tau, \tau)\}$ and we always have $\mathcal{F}_o^{\mathsf{tok}}(\sigma) = \{\tau\}$, so that $(\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp} = \emptyset$. Let $G_0 = \emptyset$, so that we always have $\sigma = \sigma'$ in S3. Then S2 and S3 require us to find $G(\tau)$ such that

$$G(\tau)(I) \subseteq I \wedge \forall o, \sigma.\ (\sigma \in I \implies (\sigma, \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma)) \in G(\tau)).$$

It is easy to show that we can find such a $G(\tau)$ if and only if (10) holds for all o. Thus, in this case it is sufficient to check that the effect of an operation preserves the invariant when applied to the same state where it was generated.



(a) $\sigma_{\mathsf{init}} = 0$ (b) $\sigma_{\mathsf{init}} = 100$

Figure 7. Executions illustrating the unsoundness of the state-based proof rule on weaker consistency models.

Causal consistency. We have $\mathsf{Token} = \emptyset$ and the conditions S2 and S3 become equivalent to

$$G_0(I) \subseteq I \wedge (\forall o, \sigma, \sigma'.\ (\sigma \in I \wedge (\sigma, \sigma') \in G_0^*) \implies (\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) \in G_0).$$

In this case the effects of all operations are described by a single guarantee relation G_0. We need to show that every operation satisfies this guarantee while assuming that concurrently executing operations at other replicas do. Note that (11), for all o, is a special case of the above obligation for $G_0 = I \times I$. Thus, (11) is an invariant-based version of the above rely-guarantee proof rule.

As we elaborate in §7, our proof rule bears a lot of similarity to proof rules for strongly consistent shared-memory concurrency [21, 27, 36]. The reasons for the soundness of our proof in the setting of weak consistency are subtle. Its soundness relies crucially on the fact that our consistency model guarantees at least causal consistency and on the commutativity of operation effects (7). For example, some consistency models do not guarantee the transitivity of happens-before [8, 47] and thus allow the execution in Figure 7(a), which uses the operations in Figure 4. Here a withdrawal hb-follows a deposit; a query sees only the withdrawal, thus violating the integrity invariant (5). Since we have proved these operations to preserve the invariant using our proof rule, this rule is unsound over a consistency model allowing the execution in Figure 7(a). We note that the obligation (11), for all o, establishes the invariant I even for a consistency model where hb is only acyclic, but not necessarily transitive.

To illustrate that our rule becomes unsound if we drop the requirement of effect commutativity (7), consider the operations in Figure 4, but with the effect of $\mathsf{interest}$ defined by (3). It is easy to show that the premiss of the rule holds for the invariant (5) even with this change. At the same time, the execution in Figure 7(b) violates the invariant, yet is consistent with the operations in Figure 4 according to Definition 2. This is because the evaluation determining the effect of $\mathsf{withdraw}(126)$ can order $\mathsf{deposit}(20)$ before $\mathsf{interest}$, whereas the evaluation determining the outcome of query can order these operations the other way round, resulting in a smaller balance. Again, the obligation (11) establishes the invariant even without (7): it ensures

$$\forall X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\ \mathsf{eval}_{\mathcal{F}}^{\dagger}(X) \subseteq I.$$

5. Event-based Proof Rule and Soundness

We now prove the soundness of the state-based proof rule. To this end, we present an *event-based* proof rule (Figure 8), from which the state-based one is derived. This event-based rule highlights the reasons for the soundness of the state-based one. Instead of reasoning about replica states, the event-based rule reasons about executions describing the events that replicas know about; the evaluation of the corresponding effects yields the replica states in the

$\exists \mathbb{G} \in \mathcal{P}(\mathsf{Exec}(\mathcal{T}) \times \mathsf{Exec}(\mathcal{T}))$ such that

E1. $X_{\mathsf{init}} \in \mathbb{I}$

E2. $\mathbb{G}(\mathbb{I}) \subseteq \mathbb{I}$

E3. $\forall X, X', X''.\ \forall e \in X''.E.$
$(X \in \mathbb{I} \wedge X' = X''|_{X''.E-\{e\}} \wedge X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \wedge$
$e \in \max(X'') \wedge X = \mathsf{ctxt}(e, X'') \wedge (X, X') \in \mathbb{G}^*)$
$\implies (X', X'') \in \mathbb{G}$

$\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$

Figure 8. Event-based proof rule.

state-based rule. In particular, we specify the desired integrity invariant as a predicate on executions: $\mathbb{I} \subseteq \mathsf{Exec}(\mathcal{T})$. The event-based rule establishes that any execution consistent with given $\mathcal{T} = (\mathsf{Token}, \bowtie)$ and $\mathcal{F}$ belongs to $\mathbb{I}$: $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$.

As before, we explain the event-based rule from the operational perspective. The rule again uses rely-guarantee reasoning, but with the guarantee $\mathbb{G}$ represented by a relation on executions. The guarantee describes the change to a replica's knowledge brought on by the replica executing a new operation or receiving the effect of an operation originally executed elsewhere.

Conditions E1 and E2 are similar to S1 and S2: E1 requires the invariant $\mathbb{I}$ to allow an empty execution X_{init} (§3), which evaluates to the initial database state σ_{init}; E2 requires the guarantee to preserve the invariant. Condition E3 is graphically illustrated in Figure 6(b). Similarly to S3, the condition E3 considers any operation, denoted by an event e, and checks that the change to the database state made by the operation satisfies the guarantee. This check is done not only at the origin replica r of e, but also at any other replica r' that receives its effect. The execution X can be thought of as describing the events known to the replica r when it executed the operation denoted by e. We assume that the execution X satisfies the invariant $\mathbb{I}$. The execution X' describes the events known to the replica r' just before it receives the effect of e; X'' describes the events known to r' after this, so that $X' = X''|_{X''.E-\{e\}}$. The execution X'' is consistent with $\mathcal{T}$ and $\mathcal{F}$; the conditions in the proof rule imply that so are X and X'. The condition $e \in \max(X'')$ (see (9)) reflects the fact that e is the latest event received by r'. The condition $X = \mathsf{ctxt}(e, X'')$ ensures that X is a part of $X' = X''|_{X''.E-\{e\}}$. This reflects the guarantee of causal message propagation: when r' receives the effect of e, this replica is guaranteed to know about all the events that the replica r knew about when it executed e.

Even though the rule allows us to assume that X is part of X', the latter may contain additional events that the replica r' found out about by the time it received the effect of e. The rule allows us to assume that the changes in the knowledge of r' brought on by adding these events satisfy the guarantee: $(X, X') \in \mathbb{G}^*$. In exchange, the rule requires us to ensure that adding the event e to the knowledge of replica r' will also satisfy the guarantee: $(X', X'') \in \mathbb{G}$.

In the following, we use the fact that the premiss of the implication in E3 entails that all events in $X'.E - X.E$ are causally independent with e.

PROPOSITION 5. *For all X, X', X'' and $e \in X''.E$,*

$$(X' = X''|_{X''.E-\{e\}} \wedge e \in \max(X'') \wedge X = \mathsf{ctxt}(e, X''))$$
$$\implies \neg\exists f \in (X'.E - X.E).\ (e \xrightarrow{X''.\mathsf{hb}} f \vee f \xrightarrow{X''.\mathsf{hb}} e).$$

PROOF. Consider $f \in (X'.E - X.E)$. Since $e \in \max(X'')$, we cannot have $e \xrightarrow{X''.\mathsf{hb}} f$. If $f \xrightarrow{X''.\mathsf{hb}} e$, then $f \in X.E$ due to $X = \mathsf{ctxt}(e, X'')$. But this contradicts $f \in (X'.E - X.E)$. ☐

We now give the proof of soundness of the event-based rule and sketch the derivation of the state-based one (we give a full proof of the latter in [26, §A]).

Let $\sqsubseteq$ be the following partial order on executions:

$$X \sqsubseteq X' \iff (X = X'|_{X.E} \wedge ((X'.\mathsf{hb})^{-1})(X.E) \subseteq X.E). \tag{14}$$

When $X \sqsubseteq X'$, we say that X is a *causal cut* of X'; any event is included into X together with its causal dependencies in X'. Operationally, $X \sqsubseteq X'$ means that X and X' can describe the knowledge of a replica at different points in the same database computation.

PROPOSITION 6.

$$\forall X \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\ \forall Y.\ (Y \sqsubseteq X \implies Y \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})).$$

THEOREM 7. *The event-based proof rule in Figure 8 is sound.*

PROOF. Assume E1-E3 hold. We prove that

$$\forall X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}).\ \forall Y.\ (Y \sqsubseteq X'' \implies (Y, X'') \in \mathbb{G}^*), \tag{15}$$

i.e., that the guarantee $\mathbb{G}$ allows us to transition into a consistent execution X'' from any of its causal cuts Y. The desired conclusion $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$ follows from (15): it implies $(X_{\mathsf{init}}, X'') \in \mathbb{G}^*$, but $X_{\mathsf{init}} \in \mathbb{I}$ (E1) and $\mathbb{G}$ preserves $\mathbb{I}$ (E2).

The proof of (15) is done by induction on the size of X''. In the base case, we must have $Y = X'' = X_{\mathsf{init}}$, which implies $(Y, X'') \in \mathbb{G}^*$. In the induction step, we consider $X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})$ and $Y \sqsubseteq X''$ such that $Y \neq X''$. We pick an event $e \in (X''.E - Y.E)$ such that $e \in \max(X'')$ and define X and X' as in E3:

$$X = \mathsf{ctxt}(e, X'') \wedge X' = X''|_{X''.E-\{e\}}.$$

Then

$$Y \sqsubseteq X' \wedge X \sqsubseteq X'. \tag{16}$$

By Proposition 6 we have $X, X' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F})$. Thus, we can apply the induction hypothesis to X' and its causal cuts X and Y, as well as to X and its causal cut X_{init}, getting:

$$(Y, X') \in \mathbb{G}^* \wedge (X, X') \in \mathbb{G}^* \wedge (X_{\mathsf{init}}, X) \in \mathbb{G}^*.$$

By E1 and E2, $(X_{\mathsf{init}}, X) \in \mathbb{G}^*$ implies $X \in \mathbb{I}$. Together with $(X, X') \in \mathbb{G}^*$, this allows us to apply E3 and obtain $(X', X'') \in \mathbb{G}$. This and $(Y, X') \in \mathbb{G}^*$ imply $(Y, X'') \in \mathbb{G}^*$, as required. ☐

In operational terms, the statement (15) established in the proof ensures that any sequence of changes in the knowledge of a replica during a database computation is described by $\mathbb{G}^*$. The above proof relies crucially on the fact that our consistency model guarantees at least causal consistency. For example, in (16) we can deduce $X \sqsubseteq X'$ from $X = \mathsf{ctxt}(e, X'')$ because happens-before is transitive.

COROLLARY 8. *The state-based proof rule in Figure 5 is sound.*

PROOF SKETCH. Assume a state-based invariant $I \subseteq \mathsf{State}$. We construct the corresponding event-based invariant $\mathbb{I}$ as the set of all executions that evaluate to a state in I: $\mathbb{I} = \mathsf{eval}_{\mathcal{F}}^{-1}(I)$. Then the conclusion $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathbb{I}$ of the event-based rule implies the conclusion $\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}_{\mathcal{F}}^{-1}(I)$ of the state-based rule.

We now show that the premiss of the state-based rule implies that of the event-based rule. Assume state-based guarantees G_0 and G that satisfy S1-S3. We construct the corresponding event-based guarantee $\mathbb{G}$ by describing the change to the knowledge of a replica

brought on by incorporating the effect of an operation satisfying the state-based guarantees G_0 and G:

$$\mathbb{G} = \{(X, Y) \mid \exists e.\ (Y.E - X.E) = \{e\} \wedge X \sqsubseteq Y \wedge (\mathsf{eval}_{\mathcal{F}}(X), \mathsf{eval}_{\mathcal{F}}(Y)) \in G_0 \cup G(Y.\mathsf{tok}(e))\}. \quad (17)$$

Thus, the guarantee $\mathbb{G}$ consists of pairs (X, Y), where Y extends X by a single event e representing the operation, and the two executions evaluate to a pair of states in G_0 or $G(\tau)$ for some token τ acquired by e.

It remains to prove that the event-based guarantee $\mathbb{G}$ satisfies conditions E1-E3. Conditions E1 and E2 trivially follow from conditions S1 and S2; we thus only need to show that S3 implies E3. Assume that for some X, X', X'' and $e \in X''.E$, the premiss of E3 holds:

$$X \in \mathbb{I} \wedge X' = X''|_{X''.E-\{e\}} \wedge X'' \in \mathsf{Exec}(\mathcal{T}, \mathcal{F}) \wedge e \in \mathsf{max}(X'') \wedge X = \mathsf{ctxt}(e, X'') \wedge (X, X') \in \mathbb{G}^*. \quad (18)$$

Let $\sigma = \mathsf{eval}_{\mathcal{F}}(X)$, $\sigma' = \mathsf{eval}_{\mathcal{F}}(X')$ and $o = X''.\mathsf{oper}(e)$. We now show that the premiss of S3 holds:

$$\sigma \in I \wedge (\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*. \quad (19)$$

First of all, $\sigma \in I$ follows from $X \in \mathbb{I}$ by the definition of $\mathbb{I}$. Furthermore, by Proposition 5, all events in $(X'.E - X.E)$ are unrelated to e in $(X''.\mathsf{hb} \cup (X''.\mathsf{hb})^{-1})$. But then by (8), they cannot acquire tokens that conflict with the ones acquired by e:

$$\forall f \in (X'.E - X.E).\ \neg(X''.\mathsf{tok}(e) \bowtie X''.\mathsf{tok}(f)).$$

Using this fact, $(X, X') \in \mathbb{G}^*$ given by (18) and the definition of $\mathbb{G}$ given by (17), we can show that

$$(\sigma, \sigma') = (\mathsf{eval}_{\mathcal{F}}(X), \mathsf{eval}_{\mathcal{F}}(X')) \in (G_0 \cup G((X''.\mathsf{tok}(e))^{\perp}))^* = (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*,$$

thus establishing (19). Then the conclusion of S3 yields $(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) \in G_0 \cup G(\mathcal{F}_o^{\mathsf{tok}}(\sigma))$, so that

$$\begin{aligned}(\mathsf{eval}_{\mathcal{F}}(X'), \mathsf{eval}_{\mathcal{F}}(X'')) &= (\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) \\ &\in G_0 \cup G(\mathcal{F}_o^{\mathsf{tok}}(\sigma)) \\ &= G_0 \cup G(X''.\mathsf{tok}(e)).\end{aligned} \quad (20)$$

This implies the conclusion of E3: $(X', X'') \in \mathbb{G}$. □

The above proof relies crucially on Lemma 4, which allows us to define $\mathsf{eval}_{\mathcal{F}}$. The lemma guarantees that, when evaluating executions, choosing different orders for causally independent events does not affect the resulting state. In (20) this allows us to choose a particular convenient order of evaluating X'' that applies the operation o last. Lemma 4 holds due to the commutativity condition (7), and this illustrates the importance of this condition for the soundness of the state-based rule.

6. Examples and Automation

We have developed a prototype tool that automates the state-based proof rule by reducing its obligations to SMT queries. Using the tool, we have verified three applications: an extended version of the banking application in Figure 4, an auction service and a course registration system. Our results are summarised in Figure 9. In the following, we first show more sophisticated uses of our proof rule using fragments of the auction and courseware applications, as well as the consistency model of parallel snapshot isolation [38, 42]. We then present our automation approach and the complete applications that we verified.

6.1 Auction Service

Figure 10 shows a fragment of an auction application. An auction can be either open or closed. While the auction is open, a client can place a bid with the amount b using the $\mathsf{place}(b)$ operation. A client can also close the auction at any time using the close operation, which declares the winner. Finally, clients can query the database state using query.

Application	# ops	# tokens	# invariants	time (ms)
Banking	5	1	1	385
Auction	14	9	12	5297
Courseware	5	5	2	512

Figure 9. Characteristics of the applications verified and the time taken by the tool. The numbers of operations are given ignoring operation parameters. The numbers of tokens are similarly given without taking into account tokens associated with different instances of the same object, such as different bank accounts. The tool was run on a Mac Mini, 3 GHz Intel Core i7.

$$\begin{aligned}
\mathsf{State} &= \mathcal{P}(\mathbb{N}) \times (\mathbb{N} \cup \{\perp\}) \\
\sigma_{\mathsf{init}} &= (\emptyset, \perp) \\
I &= \{(B, w) \mid w \neq \perp \implies B \neq \emptyset \wedge w = \max(B)\} \\
\mathsf{Token} &= \{\tau_c, \tau_p\} \\
\bowtie &= \{(\tau_c, \tau_c), (\tau_c, \tau_p), (\tau_p, \tau_c)\} \\
\mathcal{F}_{\mathsf{place}(b)}((B, w)) &= \text{if } w \neq \perp \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_p\}) \\
&\quad \text{else } (\checkmark, (\lambda(B', w').\ (B' \cup \{b\}, w')), \{\tau_p\}) \\
\mathcal{F}_{\mathsf{close}}((B, w)) &= \text{if } (w \neq \perp \vee B = \emptyset) \text{ then } (\text{✗}, \mathsf{skip}, \{\tau_c\}) \\
&\quad \text{else } (\checkmark, (\lambda(B', w').\ (B', \max(B))), \{\tau_c\}) \\
\mathcal{F}_{\mathsf{query}}((B, w)) &= ((B, w), \mathsf{skip}, \emptyset)
\end{aligned}$$

Figure 10. A fragment of an auction application.

The database state is of the form (B, w). Here B contains the amounts of the bids placed; for simplicity, we do not distinguish two bids with the same amount. The component w is either $\perp$, signifying that the auction is still open, or the winning bid. A successful $\mathsf{place}(b)$ operation has the effect of adding b to B. The close operation writes the winning bid into w. Note that the effects of two close operations do not commute. To satisfy (7), and to ensure that clients can only close the auction once, we let close operations acquire a token τ_c such that $\tau_c \bowtie \tau_c$.

The integrity invariant I we would like to maintain in the auction application is that, if the auction is closed, then the winning bid is the maximal of all the bids placed. Without using any other tokens than τ_c, this invariant can be violated: Alice can close the auction and declare the winner, e.g., 100, without being aware of a higher bid 105 placed concurrently by Bob. A query aware of both operations will return the bid set containing 105 and 100 but mark 100 as the winning bid in the set.

To preserve the invariant in the RedBlue consistency model (§3), we would have to use strong consistency for both place and close operations, i.e., let them acquire the mutually exclusive token τ_c. To address this inefficiency, Balegas et al. [9] proposed a hybrid model where consistency can be strengthened using *multi-level locks*, analogous to readers-writer locks from shared memory. In our example, we represent such a lock by a pair of tokens: τ_c, introduced before, and τ_p. Each close operation acquires τ_c, and each place operation, τ_p. We have $\tau_c \bowtie \tau_p$. Hence, for every pair of close and $\mathsf{place}(b)$ operations, either close is aware of the bid b and takes it into account when computing the winner, or $\mathsf{place}(b)$ is aware that the auction has been closed and, hence, does not place the bid. However, we do not have $\tau_p \bowtie \tau_p$ and, hence, bid placements can be causally independent. In our analogy with a readers-writer lock, bid placements play the role of readers and closing the auction, the role of a writer.

$$\begin{aligned}
\mathsf{State} &= \mathcal{P}(\mathsf{Student}) \times \mathsf{RWset}(\mathsf{Course}) \times \mathcal{P}(\mathsf{Student} \times \mathsf{Course})\\
\sigma_{\mathsf{init}} &= (\emptyset, \emptyset_{\mathsf{RWset}}, \emptyset)\\
I &= \{(S, C, E) \mid E \subseteq \mathcal{P}(S \times \mathsf{contents}(C))\}\\
\mathsf{Token} &= \{\tau_{e(c)}, \tau_{r(c)} \mid c \in \mathsf{Course}\}\\
\bowtie &= \{(\tau_{e(c)}, \tau_{r(c)}), (\tau_{r(c)}, \tau_{e(c)}) \mid c \in \mathsf{Course}\}
\end{aligned}$$

$\mathcal{F}_{\mathsf{register}(s)}((S, C, E)) =$
$(\bot, (\lambda(S', C', E').\, (S' \cup \{s\}, C', E')), \emptyset)$

$\mathcal{F}_{\mathsf{addCourse}(c)}((S, C, E)) =$
$(\bot, (\lambda(S', C', E').\, (S', \mathsf{add}(c, C'), E')), \emptyset)$

$\mathcal{F}_{\mathsf{enrol}(s,c)}((S, C, E)) =$
if $(s \notin S \vee c \notin \mathsf{contents}(C))$ then $(\text{✗}, \mathsf{skip}, \{\tau_{e(c)}\})$
else $(\checkmark, (\lambda(S', C', E').\, (S', C', E' \cup \{(s, c)\})), \{\tau_{e(c)}\})$

$\mathcal{F}_{\mathsf{remCourse}(c)}((S, C, E)) =$
if $(c \notin \mathsf{contents}(C) \vee \exists s.\, (s, c) \in E))$ then $(\text{✗}, \mathsf{skip}, \{\tau_{r(c)}\})$
else $(\checkmark, (\lambda(S', C', E').\, (S', \mathsf{remove}(c, C'), E')), \{\tau_{r(c)}\})$

$\mathcal{F}_{\mathsf{query}}((S, C, E)) = ((S, \mathsf{contents}(C), E), \mathsf{skip}, \emptyset)$

$$\begin{aligned}
\mathsf{RWset}(\mathsf{Course}) &= \mathcal{P}(\mathsf{Course}) \times \mathcal{P}(\mathsf{Course})\\
\emptyset_{\mathsf{RWset}} &= (\emptyset, \emptyset)\\
\mathsf{add}(c, (A, T)) &= (A \cup \{c\}, T)\\
\mathsf{remove}(c, (A, T)) &= (A, T \cup \{c\})\\
\mathsf{contents}((A, T)) &= A - T
\end{aligned}$$

Figure 11. A fragment of a courseware application.

Balegas et al. [9] show how to implement multi-level locks so that a replica can place a bid without any synchronisation; only an operation closing the auction has to synchronise with other replicas to make sure that no bids are placed concurrently. Thus, the most frequent operation of bid placement is the least expensive.

We now use our proof rule to show that the above consistency choice is indeed sufficient to preserve the invariant I. Let

$$\begin{aligned}
G_0 &= \{((B, w), (B, w)) \mid (B, w) \in I\};\\
G(\tau_p) &= \{((B, \bot), (B', \bot)) \mid B \subset B'\};\\
G(\tau_c) &= \{((B, \bot), (B, \max(B))) \mid B \neq \emptyset\}.
\end{aligned}$$

Then the condition S2 in Figure 5 is satisfied. We show how to check the condition S3 in the most interesting case of $o = \mathsf{place}(b)$.

Consider $\sigma = (B, w)$ and $\sigma' = (B', w')$ satisfying the premiss of S3. Then $\sigma \in I$. Also, since

$$(\sigma, \sigma') \in (G_0 \cup G((\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp}))^*$$

and $(\mathcal{F}_o^{\mathsf{tok}}(\sigma))^{\perp} = \{\tau_p\}$, we get

$$w' = w \wedge B \subseteq B' \wedge (w \neq \bot \implies B' = B). \qquad (21)$$

If $w \neq \bot$, then $\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') = \sigma'$ and, by (21), $\sigma = \sigma'$. Since $\sigma \in I$, we have

$$(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) = (\sigma', \sigma') = (\sigma, \sigma) \in G_0.$$

This implies the conclusion of S3.

If $w = \bot$, then $\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') = (B' \cup \{b\}, w')$. In this case (21) implies $w' = w = \bot$. We then get

$$(\sigma', \mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma')) = ((B', w'), (B' \cup \{b\}, w')) \in (G_0 \cup G(\tau_p)),$$

the desired conclusion of S3. Operationally, our proof rule establishes that, if the auction was open at the replica where the bid was placed, then it will be open at any replica the bid is delivered to.

Similarly to our banking application (§4), we can deal with multiple auctions by using a pair of tokens (τ_c, τ_p) for every auction. The above proof generalises straightforwardly to this case.

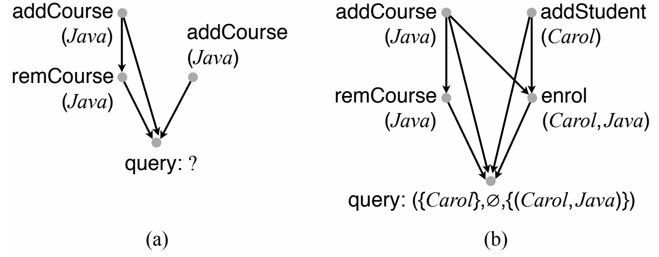


Figure 12. Executions illustrating the need for (a) replicated data types and (b) tokens in the courseware application.

6.2 Courseware

Our next example illustrates a different kind of an integrity invariant and the use of replicated data types [40] to construct commutative operations. Figure 11 shows a fragment of a courseware application. We assume sets of courses Course and students Student. A client can add a course c using $\mathsf{addCourse}(c)$ and register a student s using $\mathsf{register}(s)$. A registered student s can be enrolled into a course c using $\mathsf{enrol}(s, c)$. In the application fragment we consider, student registrations and enrolments cannot be cancelled. However, a course c that has not secured any student enrolment can be removed using $\mathsf{remCourse}(c)$. As usual, we also have a query operation.

A database state (S, C, E) consists of the set of students S, the set of courses C and the enrolment relation E between students and courses. The set of courses is actually not just an ordinary set, but a *replicated remove-wins set* RWset(Course), explained in the following. The effects of operations are mostly as expected, with courses accessed using special functions add, remove and contents on the replicated set. Note that the operation $\mathsf{enrol}(c, s)$ takes effect only if the student s is registered and the course c exists. The operation $\mathsf{remCourse}(c)$ removes the course c only when it exists and has no students enrolled into it.

Using a replicated data type for the set of courses is needed to satisfy (7), because additions to and removals from a usual set do not commute. To illustrate, consider the execution in Figure 12(a). There Alice adds a course on Java and then changes her mind and removes the course; concurrently, Bob adds the same Java course. If we maintained the information about courses using a usual set, then the outcome of the query in the figure would depend on the order in which we evaluate the effects of the causally independent operations $\mathsf{addCourse}(\mathit{Java})$ by Bob and $\mathsf{remCourse}(\mathit{Java})$ by Alice: the query would return $\emptyset$ if the addition was evaluated before removal, and $\{\mathit{Java}\}$ otherwise (see Definition 2). In an actual database, implementing the operations using ordinary sets would violate the replica convergence property (§2.1).

Replicated data types [40] provide implementations of operations on data structures with commutative effects. They differ in the way in which they resolve conflicting updates to the data structure, such as those in Figure 12(a): when using an *add-wins* set, the query in the figure will return $\{\mathit{Java}\}$, and when using a *remove-wins* set, $\emptyset$ [39]. The decision which data type to use ultimately depends on application requirements. To keep presentation manageable, in our example we use one of the simplest set data types, which provides a rudimentary version of the remove-wins semantics.

The data type represents the replicated set of courses using a pair of sets (A, T). The function $\mathsf{add}(c, \cdot)$ puts c into the set of A, and the function $\mathsf{remove}(c, \cdot)$ puts c into the set T, called the *tombstone* set. To get the contents of the replicated set, we just take the difference between A and T. The functions $\mathsf{add}(c, \cdot)$ and $\mathsf{remove}(c, \cdot)$ commute: even if the removal is evaluated first, it will

still cancel the subsequent addition[1]. This ensures that the effects of all operations in Figure 11 commute and thus satisfy (7).

The integrity invariant I we would like to maintain in this application is that the enrolment relation refers to existing courses and students only. This property is an instance of *referential integrity*, which requires an object referenced in one part of the database to exist in another. Without using tokens, the operations in our application can break the invariant. This is illustrated by the execution in Figure 12(b). There a Java course initially has no students enrolled. Then Alice removes the course and concurrently Bob enrols Carol into it, thinking that the course is still available. This results in Carol being enrolled into a non-existent course.

To ensure that such situations do not happen, we use a pair of conflicting tokens for each course $c \in \mathsf{Course}$: $\tau_{e(c)}$ and $\tau_{r(c)}$. The operation $\mathsf{enrol}(s, c)$ acquires $\tau_{e(c)}$, and the operation $\mathsf{remCourse}(c)$ acquires $\tau_{r(c)}$. Then for every pair of operations $\mathsf{enrol}(s, c)$ and $\mathsf{remCourse}(c)$, either the enrolment operation is aware that the course has been removed, or the removal is aware that there are still students enrolled into the course; in either case the corresponding operation takes no effect. However, other pairs of operations can be causally independent and, hence, do not have to synchronise. This includes pairs of operations enrolling students into courses and pairs of operations manipulating courses, such as those in Figure 12(a). The above use of tokens is equivalent to associating every course with a multi-level lock [9] that can be in one of two modes, one of which allows enrolling students into a course ($\tau_{e(c)}$) and the other removing the course ($\tau_{r(c)}$). Unlike in the auction application above, neither of the tokens $\tau_{e(c)}$ or $\tau_{r(c)}$ conflicts with itself, and thus, neither of the above lock modes is exclusive.

Our proof rule can establish that the above consistency choice is sufficient to preserve the integrity invariant. To this end, we use the following guarantees, associating changes with tokens as expected:

$$\begin{aligned} G_0 &= (I \times I) \cap \{((S, C, E), (S', C', E)) \mid \\ &\qquad S \subseteq S' \wedge \mathsf{contents}(C) \subseteq \mathsf{contents}(C')\}; \\ G(\tau_{e(c)}) &= (I \times I) \cap \{((S, C, E), (S, C, E')) \mid \\ &\qquad \exists s.\, E' = E \uplus \{(s, c)\}\}; \\ G(\tau_{r(c)}) &= (I \times I) \cap \{((S, C, E), (S, C', E)) \mid \\ &\qquad \mathsf{contents}(C) = \mathsf{contents}(C') \uplus \{c\}\}. \end{aligned}$$

The actual proof is similar to that of the auction application above and is omitted.

6.3 Parallel Snapshot Isolation

We show that our generic consistency model (§3) can be instantiated to capture *parallel snapshot isolation (PSI)*, a consistency model recently proposed for replicated databases [38, 42], which strengthens causal consistency in a way different from the models we have considered so far. We then give a proof rule specific to PSI.

We assume that the database consists of a finite set Obj of objects, ranged over by x, y. The objects store values from a set Val, so that we let $\mathsf{State} = \mathsf{Obj} \to \mathsf{Val}$. Operations in PSI perform computations that read and write objects, and any two operations writing to the same object acquire conflicting tokens. Thus, by (8) updates to the same object can never be concurrent, and the programmer does not have to merge them explicitly. However, PSI does not provide strong consistency, since it allows updates to *different* objects to be concurrent. For example, PSI allows the outcome in Figure 2(b) when the operations $\mathsf{add}(postA)$ and $\mathsf{add}(postB)$ write to different objects, e.g., representing the feeds of different users.

[1] In fact, once an element was removed, it can never be successfully added again, which may not be a desirable behaviour. There are replicated sets that provide a more sophisticated semantics [39].

$$\frac{\begin{array}{l}\exists G \in \mathcal{P}(\mathsf{State} \times \mathsf{State}) \text{ such that} \\ \quad \text{PSI1. } \sigma_{\mathsf{init}} \in I \\ \quad \text{PSI2. } G(I) \subseteq I \\ \quad \text{PSI3. } \forall o, \sigma, \sigma'.\, (\sigma \in I \wedge (\sigma, \sigma') \in (G \cap (=_{\mathsf{dom}(\mathcal{S}^{\mathsf{updates}}_o(\sigma))}))^*) \\ \qquad\qquad \implies (\sigma', \mathcal{F}^{\mathsf{eff}}_{\mathcal{S},o}(\sigma)(\sigma')) \in G \end{array}}{\mathsf{PSIExec}(\mathcal{S}) \subseteq \mathsf{eval}^{-1}_{\mathcal{F}_\mathcal{S}}(I)}$$

Figure 13. Proof rule for PSI. For a set of objects Ω, we define the relation $(=_\Omega)$ of type $\mathsf{State} \times \mathsf{State}$ as follows: for all states σ, σ', $\sigma =_\Omega \sigma'$ iff $\forall x \in \Omega.\, \sigma(x) = \sigma'(x)$.

To represent PSI in our framework, we consider a token system $\mathcal{T}_{\mathsf{PSI}} = (\mathsf{Token}, \bowtie)$, that associates every object x with a mutual exclusion token τ_x:

$$\mathsf{Token} = \{\tau_x \mid x \in \mathsf{Obj}\}; \quad \tau_x \bowtie \tau_y \iff x = y.$$

To define the semantics of operations $o \in \mathsf{Op}$, we assume a function

$$\mathcal{S} \in \mathsf{Op} \to (\mathsf{State} \to \mathsf{Val} \times (\mathsf{Obj} \rightharpoonup \mathsf{Val}))$$

and let

$$\forall o, \sigma.\, \mathcal{S}_o(\sigma) = (\mathcal{S}^{\mathsf{val}}_o(\sigma), \mathcal{S}^{\mathsf{updates}}_o(\sigma)).$$

Thus, $\mathcal{S}^{\mathsf{val}}_o(\sigma)$ gives the return value of an operation o executed in a state σ, and $\mathcal{S}^{\mathsf{updates}}_o(\sigma)$ gives the values that o writes to objects. We lift $\mathcal{S}$ to a function $\mathcal{F}_\mathcal{S}$ of the type (6) as follows:

$$\begin{aligned} \mathcal{F}_{\mathcal{S},o}(\sigma) = (&\mathcal{S}^{\mathsf{val}}_o(\sigma), \\ &(\lambda\sigma'.\, \lambda x.\, \text{if } x \in \mathsf{dom}(S) \text{ then } S(x) \text{ else } \sigma'(x)), \\ &\{\tau_x \mid x \in \mathsf{dom}(S)\}), \end{aligned}$$

where $S = \mathcal{S}^{\mathsf{updates}}_o(\sigma)$. Thus, the effect of an operation o is limited to writing the values specified by $\mathcal{S}_o$; this is unlike the general case of our consistency model, which allows arbitrary effects. The acquired tokens are those for the objects written according to $\mathcal{S}_o$. Note that the effect specified by $\mathcal{F}_{\mathcal{S},o}(\sigma)$ changes only the values of the objects written to by the operation, but the converse is not true: an operation can write to an object the same value it originally stored. This will still trigger token acquisitions and, hence, create causality relationships with other operations writing to the same object.

PROPOSITION 9. *For any $\mathcal{S}$, the function $\mathcal{F}_\mathcal{S}$ satisfies (7).*

The proof is given in [26, §A]. Note that $\mathcal{F}_\mathcal{S}$ does not always satisfy (4): the flexibility allowed by (7) is crucial to represent PSI as an instance of our generic consistency model.

We write $\mathsf{PSIExec}(\mathcal{S}) = \{X \mid X \models \mathcal{T}_{\mathsf{PSI}}, \mathcal{F}_\mathcal{S}\}$ for the set of all PSI executions with operation semantics given by $\mathcal{S}$. It is easy to show that this definition is equivalent to a recently-proposed declarative definition of PSI [17].

Figure 13 gives a proof rule for checking that operations $\mathcal{S}$ preserve an integrity invariant I when executed on PSI. The rule requires us to specify the changes performed by all operations using a single guarantee G, which has to preserve I (condition PSI2). Condition PSI3 then requires us to check that the effect of an operation o generated in a state $\sigma \in I$ satisfies the guarantee when applied to another state σ'. This state σ' can be assumed to result from σ by a finite number of changes allowed by the guarantee G that *do not modify the objects written by the operation o*. Intuitively, the latter constraint comes from the fact that such operations acquire tokens conflicting with those of o.

THEOREM 10. *The rule in Figure 13 is sound.*

$\exists G_0 \in \mathcal{P}(\mathsf{State} \times \mathsf{State}), G \in \mathsf{Token} \rightarrow \mathcal{P}(\mathsf{State} \times \mathsf{State})$
such that

T1. $\sigma_{\mathsf{init}} \in I$

T2. $G_0(I) \subseteq I \wedge \forall \tau.\ G(\tau)(I) \subseteq I$

T3. $\forall o.\ \exists T.\ \exists P_1, \ldots, P_n, Q_1, \ldots, Q_n \in \mathcal{P}(\mathsf{State})$.

T3a. $T = \bigcap \{\mathcal{F}_o^{\mathsf{tok}}(\sigma) \mid \sigma \in I\} \wedge$

T3b. $I \subseteq \bigcup_{i=1}^{n} P_i \wedge$

T3c. $\forall i = 1..n.\ P_i \subseteq Q_i \wedge$

T3d. $(G_0 \cup G(T^{\perp}))(Q_i) \subseteq Q_i \wedge$

T3e. $(Q_i \times (\mathcal{F}_o^{\mathsf{eff}}(P_i)(Q_i))) \subseteq (G_0 \cup G(T))$

$\mathsf{Exec}(\mathcal{T}, \mathcal{F}) \subseteq \mathsf{eval}_{\mathcal{F}}^{-1}(I)$

Figure 14. Proof rule used by our tool. We assume a token system $\mathcal{T} = (\mathsf{Token}, \bowtie)$ and use the same notation as in Figure 5. We let $\mathcal{F}_o^{\mathsf{eff}}(P_i)(Q_i) = \{\mathcal{F}_o^{\mathsf{eff}}(\sigma)(\sigma') \mid \sigma \in P_i \wedge \sigma' \in Q_i\}$.

The proof, given in [26, §A], derives the rule for PSI directly from the event-based rule in Figure 8. We could derive the rule for PSI from a generalisation of the state-based rule in Figure 5 that associates guarantees with *sets* of tokens. However, to simplify presentation we opted for the simpler version of the state-based rule at the expense of a more complex derivation of the rule for PSI.

6.4 Automation

Our tool uses the proof rule in Figure 14, which is derived from the one in Figure 5 and is more amenable to automation. The premisses T1 and T2 are identical to S1 and S2; T3 changes S3 in two ways. A minor change is motivated by the fact that our tool currently handles only operations that acquire the same set of tokens regardless of the state they are executed in. Hence, T3 precomputes the set of tokens T acquired by an operation o (T3a). The key way in which T3 changes S3 is that it eliminates the transitive closure of the guarantees, which is hard to automate. Whereas S3 quantifies over states σ where the effect of an operation o is generated and σ' where it is applied, T3 considers properties of these states, respectively denoted by predicates P_i and Q_i, $i = 1..n$. T3b requires the predicates P_i to cover all possible states in which o can be executed. T3c requires Q_i to cover P_i, reflecting the fact that the effect of o can be applied in a state different from the one where it was generated. T3d requires Q_i to be *stable* under the changes allowed by the guarantees [27]. Finally, T3e checks that, if the effect of o is generated in a state satisfying P_i, then applying this effect to a state satisfying Q_i is consistent with the guarantees. Note that the constraints T3c and T3d have the same effect as relating the states σ and σ' in S3 by a transitive closure of guarantees.

For example, consider the operation $o = \mathsf{withdraw}(a)$ from the banking application in Figure 4. We let $T = \{\tau\}$ and use the guarantees (12). We use two predicates:

$$P_1 = \{\sigma \mid \sigma \geq a\}; \quad P_2 = \{\sigma \mid 0 \leq \sigma < a\}.$$

These are motivated by the condition of the if-then-else in $\mathcal{F}_o^{\mathsf{eff}}$, as well as the invariant I. We then let $Q_1 = P_1$ and $Q_2 = I$. It is easy to check that the obligations in T3 are fulfilled.

Our tool accepts as input a token system $\mathcal{T}$, the semantics of operations $\mathcal{F}$ and an integrity invariant I, the latter two in the SMT-LIB format (we leave a programming language for writing operations as future work). The tool generates predicates P_i from preconditions of branches in $\mathcal{F}$. As Q_i, the tool takes either P_i or the invariant I. Finally, the tool generates guarantees G_0 and G by intersecting the semantics of operations $\mathcal{F}$ with the invariant I. The required obligations are then discharged using the Z3 SMT solver [1]. Currently our tool assumes that the condition (7) of operation commutativity is fulfilled: checking commutativity automatically is nontrivial [28].

Applications verified. The applications verified using our tool (Figure 9) are more realistic versions of the examples we discussed before (Figures 4, 10 and 11).

The banking application extends the one in Figure 4 by considering multiple accounts and allowing clients to transfer money between accounts. We preserve the non-negativity of all balances by associating a mutual exclusion token with each account, as described in §4.

The auction application extends the one in Figure 10 by additionally maintaining information about buyers, sellers and products, and by allowing clients to sell multiple product items in a single auction. Buyers and sellers can register and unregister. Registered buyers can bid in open auctions, and registered sellers can add products, create auctions consisting of these and close auctions. The complex data model of this application requires multiple integrity invariants, including referential integrity constraints spanning multiple parts of the database. This makes it nontrivial to see if enough synchronisation has been added to the application to preserve these invariants, and our tool copes with this task.

The courseware application extends the one in Figure 11 by allowing clients to cancel student registrations and enrolments. It also imposes an additional integrity invariant limiting the number of students that can register for a course; maintaining this invariant requires extra synchronisation.

The above case studies demonstrate the feasibility of applying our proof rule to realistic applications.

7. Related Work

Reasoning in strongly consistent shared memory. Our state-based proof rule interprets tokens as permissions to perform certain state changes. Such interpretations have been used in various logics for strongly consistent shared memory [20, 21, 36]. For example, such a logic could allow threads to modify the memory in a particular way only when holding a mutual exclusion lock, similar to our use of a token in the banking application (§4).

This similarity suggests that existing work in shared memory may be helpful in exploring the novel area of replicated databases. However, the distributed and weakly consistent setting in which our proof rule is applied makes the reasons for its soundness subtle. In this setting, we do not have an illusion of a single copy of the database state and a global notion of time this copy would evolve with: as Figure 1(b) illustrates, different processes can see events as occurring in different orders. The usual justification for the soundness of the proof rules for strong consistency relies on the concepts of global time and state: when considering a thread holding a mutex lock, such proof rules reason that no other thread can hold the lock *at the same time* and, hence, modify *the* memory state in the way associated with the lock. In this setting, locks constrain the global order on events. In contrast, tokens in our consistency model provide a more subtle guarantee (8), only constraining the partial happens-before relation.

Reasoning about consistency in distributed systems and databases. Several papers have considered reasoning about correctness properties on weak consistency models of replicated and centralised databases.

Bailis et al. [7] have proposed a criterion for checking when an integrity invariant is preserved by running operations without using any synchronisation at all. But they do not provide guidelines on how to introduce synchronisation if the invariant is violated.

Li et al. [31, 32] have proposed a static analysis that uses the proof rule (11) to check if executing operations on causal consistency preserves a given integrity invariant. In case when (11) fails for some operation o, the analysis suggests to execute o under strong consistency in the RedBlue consistency model (§3). However, the analysis does not check that the result will indeed validate the invariant, and our proof rule fills this gap.

Sivaramakrishnan et al. [41] have proposed a static analysis that automatically chooses consistency levels in a replicated database given programmer-supplied contracts. However, these contracts are more low-level than our invariants, since they typically constrain the happens-before relation. For example, in the banking application (Figure 4) their contract requires happens-before to totally order all withdrawal operations. The static analysis then ensures that the contract is followed, but not that it ensures the integrity invariant (5).

Lu et al. [34] proposed proof rules for establishing correctness properties of transactions running on non-hybrid weak consistency models of classical relational databases, such as snapshot isolation [12]. In contrast, we concentrate on hybrid consistency models of modern replicated databases, which are more sophisticated.

Fekete [22] considered a hybrid consistency model for relational databases where some transactions execute under snapshot isolation [12] and some under serialisability, a form of strong consistency. He proposed conditions determining which transactions in an application need to execute under serialisability for the whole application to be *robust*, i.e., produce only behaviours that would be obtained by executing *all* transactions under serialisability. In contrast, our proof rule only checks integrity invariants while allowing the application to produce weakly consistent behaviours and, hence, benefit from the resulting performance gains.

Weak memory models. Strong consistency is forgone not only by modern databases, but also by shared-memory multiprocessors and programming languages, which provide *weak memory models*. All such models used in practice are hybrid, in that they allow the programmer to strengthen consistency on demand, e.g., using memory fences. However, weak memory models usually provide only a limited number of operations on data, such as reads, writes and compare-and-swaps on single memory cells. Concurrent writes to the same memory cell result in one value being overwritten by the other. In contrast, we deal with arbitrary operations (6) that merge concurrent updates in a user-defined way.

That said, in the future there may be a fruitful exchange of ideas between program logics for applications using weakly consistent databases and those running on weak memory models. In particular, there have been recent proposals of program logics for the "release-acquire" fragment of the C/C++ memory model [45, 46]. This fragment is analogous to causal consistency, with the above caveats about the operations allowed. However, the published logics do not meaningfully handle operations requesting the stronger "sequentially consistent" level of C/C++. Reasoning about on-demand requests of stronger-than-causal consistency is precisely the goal of the present paper.

Several papers [3–5, 13, 18, 19] have verified application correctness on weak memory models using model checkers and abstract interpreters. These papers thus explore verification approaches different from the one considered in this paper. Additionally, most of the papers have focused on models similar to TSO [3, 13, 18, 19], which is stronger than the causal consistency model we consider as a baseline. As the target correctness property, papers on weak memory models have often considered robustness (see above), which is too strong a requirement for our setting. On the other hand, some of the papers [3, 4, 13, 19] automatically inferred fences required to satisfy a correctness property. We do not address the inference of consistency choices, although in the future our state-based proof rule can serve as a basis for this. In particular, the proof in Figure 6 can be used to refine a given conflict relation: when the stability check T3d fails, the relation can be extended to so as to shrink the set $G(T^{\perp})$ and make the check pass.

Consistency models. Our conflict relations are similar to those used by Pedone and Schiper [37] to specify constraints on message delivery in a broadcast algorithm. We use the conflict relations to define a high-level consistency model, which abstracts from a message-based database implementation.

In a position paper, Li et al. [30] independently proposed an idea of a hybrid consistency model similar to ours. Their model does not have a formal semantics and is less flexible than ours, since their analogue of the conflict relation is defined directly on operations, instead of indirectly using tokens. This does not allow the synchronisation mandated for an operation to depend on the state it is executed in and, hence, does not allow expressing parallel snapshot isolation (§6).

Specifying consistency models. The formal specification of our consistency model (§3) builds on a framework previously proposed to specify forms of eventual consistency [15]. Despite this similarity, we take a somewhat different approach to specifying the semantics of operations. Previous work [15] specified the return value of an event by an arbitrary function of its context in the execution (§3). In contrast, our Definition 2 uses a particular function $\mathsf{eval}^{\dagger}_{\mathcal{F}}$, itself constructed from more primitive functions $\mathcal{F}^{\mathsf{eff}}_o$, operating on states. This choice allows us to define the semantics of operations in terms of states, as opposed to events, which can then be used in our state-based proof rule. The use of states also allows to use off-the-shelf SMT solvers to discharge the required verification conditions. However, it is likely that our event-based rule may be adapted to the operation specifications used in [15].

8. Conclusion and Future Work

We presented the first proof rule establishing that a given consistency choice in a replicated database is sufficient to preserve a given integrity invariant. Our proof rule is modular and simple to use. We demonstrated this by small but nontrivial examples, and by reducing the verification conditions of the proof rule to SMT checks. Despite this simplicity, the soundness of our proof rule is nontrivial: the rule fully exploits the guarantees provided by our consistency model while correctly accounting for anomalies it allows.

Our results represent only an initial step in building an infrastructure of reasoning methods for applications using modern replicated databases. They open several avenues for future work. First, our generic consistency model is not implemented by any database in its full generality; we use it only as a means to compactly represent a selection of more specific models in existing implementations. However, in the future the generic model can serve as a basis for exploring the space of possible hybrid consistency models. One could also consider a database that implements our model in its general form.

Second, the soundness of our proof rule relies on the fact that our consistency model guarantees at least causal consistency (§4). Even though causal consistency can be implemented without any synchronisation between replicas, this model has its cost [14]. In the future, we plan to propose proof rules for weaker models where causality preservation is not guaranteed for all operations. We also hope to generalise our methods to more expressive correctness properties than integrity invariants.

Finally, in this paper we used the event-based proof rule just to structure the proof of soundness of the state-based one. However, the event-based rule is also interesting in its own right. In the future it can be used in cases when, to prove a correctness property,

we need to maintain information about the computation history. For example, this is often necessary when reasoning about shared-memory concurrency [23, 25].

Acknowledgements. We thank Valter Balegas for helpful discussions and Vincent Gramoli for comments on an earlier draft of this paper. Gotsman was supported by an EU FET project ADVENT. Ferreira, Najafzadeh, and Shapiro were supported in part by an EU project SyncFree (FP7 609 551, 2013–2016). Yang was supported by EPSRC and an Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIP, No. R0190-15-2011).

References

[1] https://github.com/Z3Prover/z3.

[2] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.

[3] P. A. Abdulla, M. F. Atig, and N. T. Phong. The best of both worlds: Trading efficiency and optimality in fence insertion for TSO. In *ESOP*, 2015.

[4] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. In *CAV*, 2014.

[5] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, 2013.

[6] Amazon. Supported operations in DynamoDB. http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html, 2015.

[7] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *PVLDB*, 2015.

[8] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP transactions. In *SIGMOD*, 2014.

[9] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting the consistency back into eventual consistency. In *EuroSys*, 2015.

[10] Basho Inc. Using strong consistency in Riak. http://docs.basho.com/riak/latest/dev/advanced/strong-consistency/, 2015.

[11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *POPL*, 2011.

[12] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD*, 1995.

[13] A. Bouajjani, E. Derevenetc, and R. Meyer. Checking and enforcing robustness against TSO. In *ESOP*, 2013.

[14] M. Bravo, N. Diegues, J. Zeng, P. Romano, and L. E. T. Rodrigues. On the use of clocks to enforce consistency in the cloud. *IEEE Data Eng. Bull.*, 38(1), 2015.

[15] S. Burckhardt, A. Gotsman, H. Yang, and M. Zawirski. Replicated data types: specification, verification, optimality. In *POPL*, 2014.

[16] S. Burckhardt, D. Leijen, M. Fähndrich, and M. Sagiv. Eventually consistent transactions. In *ESOP*, 2012.

[17] A. Cerone, G. Bernardi, and A. Gotsman. A framework for transactional consistency models with atomic visibility. In *CONCUR*, 2015.

[18] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Predicate abstraction for relaxed memory models. In *SAS*, 2013.

[19] A. M. Dan, Y. Meshman, M. T. Vechev, and E. Yahav. Effective abstractions for verification under relaxed memory models. In *VMCAI*, 2015.

[20] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, 2010.

[21] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *ESOP*, 2009.

[22] A. Fekete. Allocating isolation levels to transactions. In *PODS*, 2005.

[23] M. Fu, Y. Li, X. Feng, Z. Shao, and Y. Zhang. Reasoning about optimistic concurrency using a program logic for history. In *CONCUR*, 2010.

[24] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2), 2002.

[25] A. Gotsman, N. Rinetzky, and H. Yang. Verifying concurrent memory reclamation algorithms with grace. In *ESOP*, 2013.

[26] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems (extended version). Available from http://software.imdea.org/~gotsman/.

[27] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*. North-Holland, 1983.

[28] D. Kim and M. C. Rinard. Verification of semantic commutativity conditions and inverse operations on linked data structures. In *PLDI*, 2011.

[29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9), 1979.

[30] C. Li, J. Leitão, A. Clement, N. Preguiça, and R. Rodrigues. Minimizing coordination in replicated systems. In *Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC)*, 2015.

[31] C. Li, J. Leitão, A. Clement, N. M. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *USENIX ATC*, 2014.

[32] C. Li, D. Porto, A. Clement, R. Rodrigues, N. Preguiça, and J. Gehrke. Making geo-replicated systems fast if possible, consistent when necessary. In *OSDI*, 2012.

[33] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *SOSP*, 2011.

[34] S. Lu, A. J. Bernstein, and P. M. Lewis. Correct execution of transactions at different isolation levels. *IEEE Trans. Knowl. Data Eng.*, 16(9), 2004.

[35] Microsoft. Consistency levels in DocumentDB. http://azure.microsoft.com/en-us/documentation/articles/documentdb-consistency-levels/, 2015.

[36] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theor. Comput. Sci.*, 375(1-3), 2007.

[37] F. Pedone and A. Schiper. Generic broadcast. In *DISC*, 1999.

[38] M. Saeida Ardekani, P. Sutra, and M. Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, 2013.

[39] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Technical Report 7506, INRIA, 2011.

[40] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *SSS*, 2011.

[41] K. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. In *PLDI*, 2015.

[42] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional storage for geo-replicated systems. In *SOSP*, 2011.

[43] D. Terry. Replicated data consistency explained through baseball. *Commun. ACM*, 56(12), 2013.

[44] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[45] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating weak memory with ghosts, protocols, and separation. In *OOPSLA*, 2014.

[46] V. Vafeiadis and C. Narayan. Relaxed separation logic: A program logic for C11 concurrency. In *OOPSLA*, 2013.

[47] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.

6.2 Iwan Briquemont, Manuel Bravo, Zhongmiao Li, and Peter Van Roy. Conflict- free partially replicated data types. In Proceedings of the 7th IEEE Interna- tional Conference on Cloud Computing Technology and Science (CloudCom 2015). IEEE, Nov 2015.

Conflict-free Partially Replicated Data Types

Iwan Briquemont[*], Manuel Bravo[*†], Zhongmiao Li[*†] and Peter Van Roy[*]
[*]Université catholique de Louvain, Belgium
[†]Instituto Superior Técnico, Universidade de Lisboa, Portugal

***Abstract*—Designers of large user-oriented distributed applications, such as social networks and mobile applications, have adopted measures to improve the responsiveness of their applications. Latency is a major concern as people are very sensitive to it. Geo-replication is a commonly used mechanism to bring the data closer to clients. Nevertheless, reaching the closest datacenter can still be considerably slow. Thus, in order to further reduce the access latency, mobile and web applications may be forced to replicate data at the client-side. Unfortunately, fully replicating large data structures may still be a waste of resources, specially for thin-clients.**

We propose a replication mechanism built upon conflict-free replicated data types (CRDT) to seamlessly replicate parts of large data structures. The mechanism is transparent to developers and gives improvements without increasing application complexity. We define partial replication and give an approach to keep the strong eventual consistency properties of CRDTs with partial replicas. We integrate our mechanism into SwiftCloud, a transactional system that brings geo-replication to clients. We evaluate the solution with a content-sharing application. Our results show improvements in bandwidth, memory, and latency over both classical geo-replication and the existing SwiftCloud solution.

***Keywords*—CRDTs, partial replication, caching**

I. Introduction

Globally accessible web applications, such as social networks, aim to provide low-latency access to their services. Thus, data locality is a fundamental property of their systems. Geo-replication is a common solution where data is replicated in multiple datacenters [1]–[3]. In this scenario, user requests are forwarded to the closest datacenter. Therefore, the latency is reduced. Unfortunately, the latency, even when accessing the closest datacenter, may still be considerable. [4], [5] argue that clients are sensitive to even small increases of latency.

Systems such as [6], [7] use caching techniques to yet reduce latency even more. However, this can be challenging and expensive. For instance, one could simply use client caches for reading purposes. Nevertheless, in order to keep some consistency guarantees and freshness of data, mechanisms, such as cache invalidation, need to be used. Scaling these kinds of techniques is difficult and directly affects the performance. Moreover, one could let clients apply write operations locally and eventually propagate them. However, this can cause conflicts between replicas and potential rollbacks.

The recently formalized CRDTs [8], [9] can serve to diminish the impact of some of the previously mentioned problems. These data types are conflict-free by default; therefore, no conflict resolution mechanisms need to be written by application developers. SwiftCloud [10], a geo-replicated storage system that ensures causal consistency, benefits from CRDT semantics. It replicates CRDTs not only across datacenters, but it also replicates them in clients. It allows read and write operations to be directly executed in client caches. In consequence, SwiftCloud reduces latency, and enables off-line mode during disconnection periods.

The current specifications of CRDTs do not allow partitioning. Thus, a CRDT replica is assumed to contain the full data structure. We believe partitioned CRDTs may pose several benefits for current applications. First, CRDTs can easily become heavy data structures, such as a set CRDT that contains the posts of a user wall in a Facebook-like application. In many cases, users are simply interested in the most relevant posts, according to some criterium. For instance, one may be interested in reading the top-ten most voted posts of a Reddit-like application. Thus, replicating the whole CRDT is a waste of resources, of both storage and bandwidth. The former can be critical when thin devices, such as smartphones, are considered as clients. These types of clients have limited memory resources; therefore, it is convenient to avoid storing unnecessary data. On the other hand, bandwidth is one of the most costly resources offered by cloud providers such as Amazon S3 [11], Google Cloud Storage (GCS) [12], and Microsoft Azure [13]; therefore, it is beneficial to use it efficiently. Second, the full replication of CRDTs in clients may arise security concerns. By partitioning CRDTs, applications could precisely decide which data each client stores. This could keep malicious clients from storing sensitive data. Finally, they can also be used to provide multiple fidelity requirements for data accommodated in resource-limited devices, while keeping consistency between fidelity levels [14]. This could be achieved by not replicating less important information on mobile devices.

In this paper, we propose a new kind of CRDTs that allows partitioning. We call them "Conflict-free Partially Replicated Data Types" (hereafter CPRDTs). We study how partitions of the same CRDT can interact among each other and still maintain its consistency guarantees. Furthermore, we revise previously defined CRDT specifications and propose new specifications that consider partitioning. One could claim that developers could simply re-format their data structures to obtain similar benefits; nevertheless, this adds complexity to application development and, in some cases, optimal results can be difficult to achieve. We propose to better integrate CPRDTs into the system. Thus, developers will benefit from them transparently, without being aware of their existence.

The major contributions of this paper are the following: (i) definition of the new CPRDTs, which includes revisiting the specifications of previously defined CRDTs; (ii) extension of SwiftCloud to integrate CPRDTs; and (iii) extensive evaluation of the performance improvements of CPRDTs in SwiftCloud. The latter includes the implementation of a Reddit-like [15], [16] application, called SwiftLinks, on top of SwiftCloud.

The remainder of the paper is organized as follows: Section II presents a formal definition of the partitioned CRDTs; Section III discusses some practical issues of CPRDTs; Section IV presents an extensive evaluation of the SwiftCloud extension that includes CPRDTs; Section V briefly describes preceding related work; finally, Section VI concludes the paper.

II. Conflict-free Partially Replicated Data Types

Allowing partitioning poses new challenges: all operations are not enabled on partial replicas, which means new preconditions must be added to ensure correctness. However, these preconditions must not compromise the convergence of replicas. Plus, a partial replica could vary the parts it keeps, by choosing to replicate more, or less, parts. This has to be done without losing data and still achieving convergence.

A. Example of use

Let us use an example to illustrate the advantages of CPRDTs: the user wall of a social network. We can model a user's wall as a set. In this example, there are four users that interact: Alice, Bob, Charlie and an anonymous user. Bob is a friend of Alice, while Charlie is a friend of Bob, but not of Alice. Participating users may want to read or post something in Alice's wall. We make two assumptions: (i) users maintain a full replica of their wall; and (ii) a user X that reads or posts in user Y's wall replicates user Y's wall locally.

Each post contains a date, an author, and a message. Each user is allowed to read a subset of other users' walls, depending on their friendship and posts visibility (private or public). For instance, Bob can read all the posts of Alice's wall because of their friendship. Nevertheless, Charlie can only read public and Bob's posts (friends of friends). Finally, any other user can only read public posts.

We can assume that Alice has been using the social network for a few years and there are a considerable number of posts on her wall. It seems natural that a user should not have to replicate the whole wall to simply read the latest posts. Nevertheless, this is what presumably may occur in a fully-replicated scenario (CRDT-like), where the data structures cannot be partitioned and we still want to replicate data in clients-side. One solution is to manually split the data structure according to some criteria (e.g. by date, author or privacy setting). However, developers then need to anticipate how users will use the application. While possible in some cases, it seems to make the application cumbersome to write.

In this scenario, CPRDTs have two applications. On the one hand, CPRDTs abstract the partitioning from the application. Thus, from the point of view of programmers, there will only be one logical data structure per wall. This simplifies the developer's task. Moreover, this allows a more efficient and fine-grained partitioning adapted to the needs of a particular client in a specific point of time, which may be impossible if the partitioning is done manually by developers. The second application of CPRDTs is related to the enforcement of security policies. We may want users to only replicate posts that they are allowed to see. This could keep malicious users from storing sensitive data locally.

B. Definitions

Before defining CPRDTs, we have to clarify some concepts that we will use throughout the paper. An *object* is a named instance of a CRDT or CPRDT in our case. Each participating process replicates a set of objects. A process that replicates an object is called *replica* of the CRDT (or CPRDT) instantiated by the object. Objects can be read using *query* operations and modified by *update* operations. Query operations return the abstract state of the object, that we call the *data* of the object. Nevertheless, additional data, which we refer as *metadata*, is kept internally to ensure convergence.

An update operation can have preconditions that capture its safety requirements. In consequence, an operation is said to be *enabled* at a replica, if it satisfies its preconditions. For instance, the remove operation of a set is enabled only if the element to be removed is present in the set.

Previous definitions fit into both CRDTs and CPRDTs. Nevertheless, for CPRDTs, we further consider that a process might replicate an object partially: it only has access to a part of data, thus the process only keeps the metadata required for that given part. Intuitively, this means that only part of the data structure is replicated: some elements of a set, a subgraph of a graph, or a slice of a sequence. CRDTs that only have one element, such as counters and registers, cannot be partitioned and therefore do not need to be specified as CPRDTs.

particle We define a *particle* as an element of a collection. For instance, a particle in a set would be any element that can be added to the set.

Apart from the definition of particle, we introduce three new concepts: *shard set*, *required*, and *affected*.

shard set Each replica x_i of a CPRDT has associated a set of particles, namely *shard set* in analogy to the databases concept. Respectively, $\mathrm{shard}(x_i)$ is a function that returns the *shard set* of x_i. A replica x_i only knows the state of the particles in $\mathrm{shard}(x_i)$; therefore, it can only enable query operations that require those particles. Furthermore, a replica x_i only needs to receive update operations that affect the particles in $\mathrm{shard}(x_i)$.

There are two special cases: a *full* replica and a *hollow* replica. We use π to represent the full set of possible particles a CPRDT may be interested in. The set π may be infinite. Thus, we say that a *full* replica's *shard set* is equal to π. Notice that a *full* replica CPRDT is equivalent to a normal CRDT. On the other hand, when $\mathrm{shard}(x_i) = \varnothing$, then x_i is a *hollow* replica (as named in [17]). A *hollow* replica does not maintain any state. Nevertheless, it can still handle updates (section II-C2).

required For an operation op with its arguments, $\mathrm{required}(op)$ is the set of particles needed by op to be properly executed. This means that, for replica x_i, an operation op is enabled only if $\mathrm{required}(op) \subseteq \mathrm{shard}(x_i)$. For instance, for the lookup operation of a set, $\mathrm{required}(lookup(e)) = \{e\}$ where e is an element of the set. In case $e \notin shard(x_i)$, the replica x_i does not know whether e is in the set because it has not kept a state for it. This implies that updates affecting e have not been necessarily seen by x_i.

affected The function $\mathrm{affected}(op)$ returns a particle that may have its state affected after executing an update operation. We assume that an update can only affect one particle. This may

not be true for complex data structures, however it is always possible to split an operation into several ones that each only affects one particle. For example, for a graph, an operation for removing a vertex will remove the vertex as well as all its edges. It can be split into several sub-operations that firstly remove all edges of the vertex and then remove the vertex.

C. Replication

As for the original CRDTs, we consider two equivalent replication techniques: state- and operation-based. Allowing partitioning introduces changes in the way these replication techniques work. Furthermore, concepts such as causal history and convergence have to be revisited. The following definitions are based on the ones in [8] for fully-replicated CRDTs.

To simplify our definitions, we assume that the *shard set* of a CPRDT is fixed. However, in practice, it can be necessary to dynamically change it. Nevertheless, definitions apply if we consider that changing the *shard set* is equivalent to the creation of a new CPRDT replica.

Since the abstract state of a CPRDT may change after applying an update, we denote the abstract states of a CPRDT replica (x_i) by an increasing numbered sequence as $s_k(x_i)$, such as $s_0(x_i), s_1(x_i)$... $s_k(x_i)$...

Now we define when two replicas are equivalent.

Definition 1 (Equivalence)**.** *x_i and x_j have equivalent abstract states if all `query` operations q, for which* $\mathrm{required}(q) \subseteq (\mathrm{shard}(x_i) \cap \mathrm{shard}(x_j))$*, return the same values.*

Different replicas of the same CPRDT might have different *shard sets*. Thus, we define intersecting abstract state as the abstract state for the particles in the intersection of *shard sets*.

Definition 2 (Intersecting abstract state)**.** *For a replica x_i with its current state $s_k(x_i)$, $s_k(x_i|x_j)$ denotes the s_k state for $particles \in \mathrm{shard}(x_i) \cap \mathrm{shard}(x_j)$.*

The requirement for replicas to converge is that they apply, directly or indirectly, the same update operations. We can informally define the causal history of a replica, denoted by $C_k(x_i)$, as a set containing the applied update operations. As x_i applies each operation, its causal history goes through a sequence of states $C_0(x_i), C_1(x_i), ..., C_k(x_i), ...$. We also define the intersecting causal history as $C_k(x_i|x_j) = \{f \in C_k(x_i) | \,\mathrm{affected}(f) \in (shard(x_i) \cap shard(x_j))\}$. Intuitively, it includes updates from $C_k(x_i)$ that affect the particles of x_j.

Now, we are ready to formally define convergence in the context of CPRDTs:

Definition 3 (Eventual Convergence of Partial Replicas)**.** *Two partial replicas x_i and x_j of an object x converge eventually if the following conditions are met:*

- *Safety: $\forall i, j : \forall k, k'$, if $C_k(x_i|x_j) = C_{k'}(x_j|x_i)$, then $s_k(x_i|x_j) = s_{k'}(x_j|x_i)$.*
- *Liveness: $\forall i, j : \forall k$, if $f \in C_k(x_i)$ and* $\mathrm{affected}(f) \in \mathrm{shard}(x_j)$*, then $\exists k'$ that $f \in C_{k'}(x_j)$.*

1) State-based partial replication: In this replication technique, a replica ships its whole internal state to the rest. Upon arrival, replicas merge both the local and the received states. The merge method is an idempotent, commutative and associative operation that has two replicas internal states as arguments. In the CPRDTs context, the *merge* method used by a replica must only merge the state of the particles belonging to the intersection between local and remote replicas *shard sets*, and ignore the rest.

State-based replication is an interesting propagation mechanism since it poses almost no communication requirements. Nevertheless, it may be expensive to always ship the full internal state. CPRDTs can optimize this technique since only parts of the state need to be sent and received. We define the causal history of a replica for state-based replication as follows:

Definition 4 (Causal History on Partial Replicas - state-based)**.** *For any replica x_i of x:*

- *Initially, $C_0(x_i) = \varnothing$.*
- *Before executing `update` operation f,*
  if $\mathrm{affected}(f) \in \mathrm{shard}(x_i)$ *then execute f and* $C_{k+1}(x_i) = C_k(x_i) \cup \{f\}$,
  otherwise $C_{k+1}(x_i) = C_k(x_i)$.
- *After executing merge against states x_i, x_j,* $C_{k+1}(x_i) = C_k(x_i) \cup \{f \in C_{k'}(x_j) | \,\mathrm{affected}(f) \in \mathrm{shard}(x_i)\}$

To achieve convergence with state-based replication on partial replicas, updates operations cannot be applied if it affects a particle that is not in that replica's *shard set*. This would violate the liveness property of convergence as that update might not be added to the causal history of another replica when merging. Thus, an operation f is disabled if $\mathrm{affected}(f) \notin \mathrm{shard}(x_j)$. On the other hand, since the replicas only converge on their common parts, a replica x_i just needs to send to another, x_j, the state of the intersection of their shards $(\mathrm{shard}(x_i) \cap \mathrm{shard}(x_j))$.

2) Operation-based partial replication: As with classical CRDTs, the update operations are divided into two phases: *prepare* and *downstream* phase. The former is done at the source replica and does not have any side-effect. The latter is applied at all replicas and it affects the state of replicas. We define the causal history of a replica for operation-based replication as follows:

Definition 5 (Causal History on Partial Replicas - op-based)**.** *For any replica x_i of x:*

- *Initially, $C_0(x_i) = \varnothing$.*
- *After executing the `downstream` phase of operation f at replica x_i,*
  if $\mathrm{affected}(f) \in \mathrm{shard}(x_i)$ *then* $C_{k+1}(x_i) = C_k(x_i) \cup \{f\}$,
  otherwise $C_{k+1}(x_i) = C_k(x_i)$.

In contrast to CRDTs, CPRDTs only have to broadcast updates to the replicas interested in the particles affected by the update. Therefore, an update u is broadcasted to x_i if $\mathrm{affected}(u) \in \mathrm{shard}(x_i)$. This poses an interesting situation.

A CPRDT replica can sometimes complete the first phase of the update operation without necessarily completing the second phase. For instance, a replica x_i, whose $\mathrm{shard}(x_i)$ are particles a and b, receives an update operation that affects particle c. In this situation x_i may complete the prepare phase, broadcast the downstream operation to the interested replicas, and discard it locally. We name this scenario *blind updates*. This can only happen in operation-based replication. Hollow replicas, whose shard is empty, can only do blind updates.

D. Specification of CPRDTs

In this section, we present the specifications of an operation-based observed-remove set (OR-set) CPRDT. We resort into this example in order to better illustrate how to integrate the newly defined concepts into a CRDT (original specifications in [8]); and thus, transform it into a CPRDT. More examples of CPRDTs and generic specification templates, for both operation- and state-based, are found in [18].

An OR-set works as follows: (i) elements are uniquely tagged by the source replica when added to the set. The source replica is the one receiving the client operation. (ii) concurrent additions of the same element are all reflected in the set internal state by storing them with different tags. (iii) a remove operation is transformed into the list of unique tags related to the element to be removed that are present in the source replica. Since causal delivery is assumed, this ensures convergence of replicas even in the presence of concurrent adds and removes of the same element.

The specifications incorporate (i) the particle definition (line 1); (ii) the *required* and *affected* preconditions (lines 11, 15 and 19); and (iii) a new function called *fraction*. The *fraction* operation allows us to create new partial replicas from a subset of a given replica. The subset we want to copy in the new replica is defined by a set of particles. More formally, *fraction* can be defined as follows:
$x_j = \mathrm{fraction}(x_i, Z)$, where Z is the set of particles to replicate. The operations ensures that $\mathrm{shard}(x_j) = \mathrm{shard}(x_i) \cap Z$.

Specification 1 Op-based OR-set with Partial Replication

1: **particle definition** A possible element of the set.
2: **payload** set S
3: **initial** $\varnothing$
4: **query** lookup(element e) : boolean b
5: **required particles** $\{e\}$
6: **let** $b = \exists u : (e, u) \in S$
7: **update** add(element e)
8: **prepare** (e) : α
9: **let** $\alpha = unique()$
10: **effect** (e, α)
11: **affected particles** $\{e\}$
12: $S := S \cup \{e, \alpha\}$
13: **update** remove(element e)
14: **prepare** (e) : R
15: **required particles** $\{e\}$
16: **pre** lookup(e)
17: **let** $R = \{(e, u) | \exists u : (e, u) \in S\}$
18: **effect** (R)
19: **affected particles** $\{e\}$
20: **pre** $\forall (e, u) \in R : add(e, u)$ has been delivered
21: $S := S \setminus R$
22: **fraction** (particles Z) : payload D
23: **let** $D.S = \{(e, u) \in S | e \in Z\}$

III. Practical issues

In this section, we discuss (i) *shard queries*, and (ii) the implications of allowing dynamic shard sets. Both issues are relevant for making CPRDTs practical.

A. Shard queries

The operation *fraction*, introduced in II-D, is the canonical form to define the *shard set* of a replica. Nevertheless, *fraction* is not practical. In practice, applications will transform their semantics into a high-level query language. For instance, an application could issue a query in the form of "give me the first 10 elements of your sorted set". We name this type of queries *shard queries*. They bridge the gap between the application semantics and the function *fraction* adding expressiveness to the usage of CPRDTs.

There are two types of *shard queries*: version-independent and version-dependent. The former only depends on the properties of the particles, and not in the version of the CPRDT. In contrast, the latter depends on the current version of the CPRDT. Let us use a CPRDT set whose domain is the set of integers as example. A version-independent query could be "integers greater than 0". This *shard query* will always return the same *shard set* ($(0, +\infty)$) independently of the queried CPRDT version. On the other hand, a version-dependent query, such as "the 10 highest integers in the set", will return a different *shard set* depending on which elements have been already added, and removed, on the version being queried.

Version-independent queries are easier to work with: they are comparable. One could determine which query is more specific without knowing the version of the CPDRT they apply to. While with version-dependent queries, one can only compare queries if they apply to the same version. Nevertheless, both types are needed in order to make CPRDTs practical.

B. Dynamic shard set

Dynamic *shard set* refers to the capability of a partial replica to modify, either shrink or expand, its *shard set*. We believe this capability is useful in practice, e.g. a client may become interested in new parts. Having dynamic *shard set*, a replica does not need to be re-created, only the missing state needs to be grabbed. Nevertheless, maintaining convergence in some scenarios can become challenging.

On the one hand, a partial replica can easily shrink its *shard set* without compromising convergence in the operation-based scenario. A replica only needs to take two things into consideration: (i) updates prepared locally have been already broadcasted, and (ii) the data to be dropped is replicated by some other replica; therefore, data is not lost. On the other hand, expanding a partial replica is more tricky. For instance, in an operation-based scenario, the following situation can easily occur: (i) a replica's (x_i) *shard set* is a, c; therefore, x_i does not receive updates that affect b; (ii) suddenly, x_i becomes interested in b and starts accepting updates on b; (iii) unfortunately, x_i will not converge since updates have been missed. In order to ensure convergence, extra communication between replicas would be needed to recover dropped updates. This would add complexity to the underlying protocols.

In state-based replication, shrinking or expanding the *shard set* is simpler. On the one hand, a replica only needs to broadcast its state before shrinking its *shard set*. On the other hand, a replica that wants to expand its *shard set* only needs to merge its current state with the state of a replica that contains new particles.

IV. Evaluation

In this section, we report the results of our experimental evaluation. This study aims at evaluating the benefits of CPRDTs in terms of memory, bandwidth and latency.

SwiftLinks In order to evaluate our solution, we implemented an application, namely SwiftLinks, on top of SwiftCloud. SwiftLinks is a vote-based content-sharing application based on Reddit. In short, the application allows users to create forums where they can publish posts. Then, users can vote posts positively or negatively. As a consequence, posts get ranked. In addition, users can comment posts and other comments. Users can also vote comments, and consequently, comments get ranked (more information [15], [16]).

SwiftLinks is modeled with three types of data structures: (i) OR-Set for each forum, (ii) a novel Remove-once Tree for each tree of comments, and (iii) Last-Writer-Wins Registers for each vote associated to a post/comment. The application uses both types of queries: version-independent and version-dependent. The former is mostly used for reading single comments or posts. The latter is used for reading ranking of posts and comments.

Warm-up We used Reddit's API to fetch data to warm up our system. For each benchmark, we create 10000 posts over 20 forums (so an average of 500 posts per forum). Each post has 20 comments on average. Moreover, each post has an average of 170 votes, while comments an average of 13 votes.

Workload Our workloads are composed by read and update operations. Read operations are executed over posts and comments. On the other hand, there are three types of update operations: (i) new post, (ii) new comment, and (iii) new vote.

For most of the experiments, 20% of the operations are updates and 80% are reads. Furthermore, 90% of the operations are biased to previously accessed objects. This means that they are likely to hit the cache. The rest (10%) is done on randomly selected posts and comments.

A. Integration of CPRDTs into a real system

We chose SwiftCloud [10] to integrate CPRDTs. SwiftCloud is a geo-replicated cloud storage system written in Java that stores CRDTs and caches data at clients. It consists of several datacenters that fully replicate the key-space. Clients indirectly communicate through the datacenters. In absence of failures, a client always interacts with its closest datacenter and caches accessed data in its local cache. SwiftCloud provides transactional causal+ consistency. Transactions are first executed and committed on the client side, then propagated to the datacenters. For fault tolerance purposes, committed transactions are only visible after they have been seen by K datacenters.

In our version of SwiftCloud, datacenters store full replicas as in the original implementation. Nevertheless, clients only cache partial replicas. Having full replicas coexisting with the partial replicas considerably simplifies the management of the latter. This poses several advantages in comparison to an ad-hoc architecture where no full replicas, namely authorities, are assumed. Firstly, clients can discard their (partial) replicas at will as long as their updates have been reliably sent to an authority. Secondly, a client can request any fraction from an authority in order to either get a new partial replica, or to expand its own *shard set*. Notice that having an authority also simplifies the integration of state-dependent *shard queries* in the system, very difficult and costly otherwise. Finally, the authority could store which particles each partial replica has in his *shard set*. Thus, it could only propagate operations to the interested replicas, saving bandwidth.

B. Experimental setup

SwiftLinks was evaluated using three Amazon EC2 servers as datacenters: one in Ireland and two in the USA (east and west coast). The EC2 instances are equivalent to a single core 64-bit 2.8 GHz Intel Xeon virtual processor (4 ECUs) with 7.5 GB of RAM. The clients run in 15 PlanetLab nodes located near the DCs. These nodes have heterogeneous configurations with varying processing power and RAM. We set up five SwiftLinks users running concurrently per node, a total of 75. Each client performs an operation per second.

Throughout the evaluation, we use three different modes:

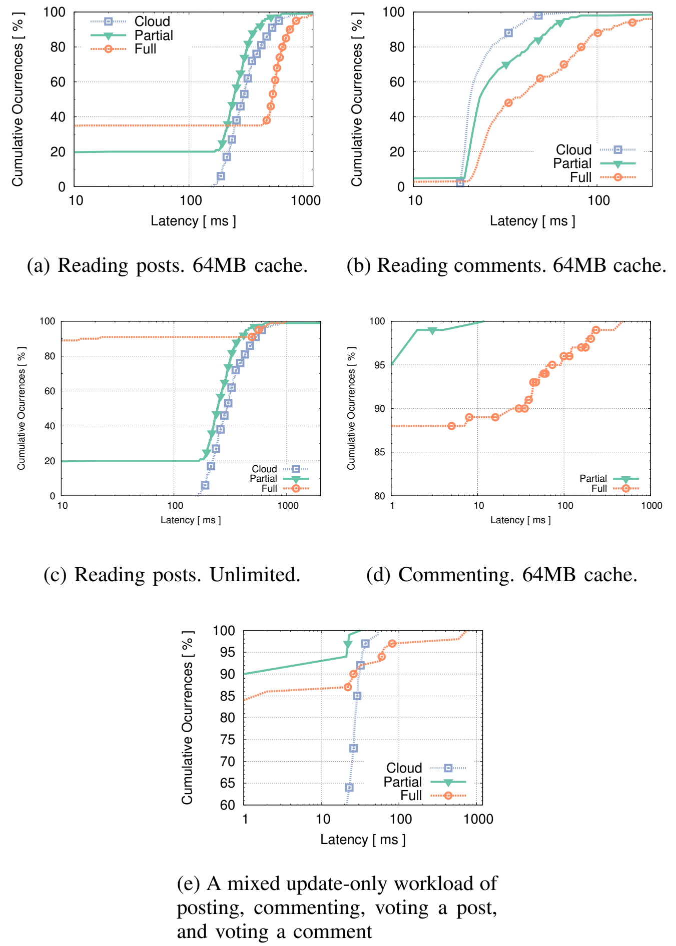
- *Cloud*: This mode simulates a typical geo-replicated system. Clients do not cache any data. Operations are applied synchronously at one datacenter and asynchronously replicated to the rest of datacenters.
- *Partial*. This is the mode that integrates the CPRDTs. Thus, clients only fetch and cache parts of the data structures (CRDTs) as needed.
- *Full*. This is the SwiftCloud approach. Clients cache whole CRDTs even when only part of it is needed.

We limit the capacity of the cache in our experiments to simulate memory restrictions on thin clients, such as a mobile phone. Nowadays, a mobile phone can have up to several gigabytes of memory, but it can easily have tens of applications running simultaneously. An application needs to cohabit with many other applications with limited memory. Therefore, we use 64MB as the default size for cache. If the cache size exceeds this limit, the least recently used object is dropped. In this configuration, *full* and *partial*, if the cache contains the required data, the operations are run locally, and asynchronously propagated to the closest datacenter.

The difference between *full* and *partial* is that the latter benefits from the partial replication mechanism described in the paper. This means that objects are fetched in parts as needed, so the cache can hold parts of an object. For instance, a query for the top ten posts of a forum would only replicate those ten posts in clients cache. On the other hand, for the *full* mode, the objects are only fully replicated in clients side, as in SwiftCloud. Therefore, the same top ten posts query would replicate the whole forum.

C. Latency

We evaluated the perceived latency for various operations with and without partial object replication. Figure 1 shows the cumulative distribution functions of different operations' latency. These results are obtained after a warm-up phase for the cache. This means that the cache is pre-filled with objects that will be used by the operations present in the workload. For the *full* and *partial* mode, there are always a percentage of operations with a very low latency. We can conclude that it is the percentage of operations that hit the cache.



(a) Reading posts. 64MB cache.

(b) Reading comments. 64MB cache.

(c) Reading posts. Unlimited.

(d) Commenting. 64MB cache.

(e) A mixed update-only workload of posting, commenting, voting a post, and voting a comment

Fig. 1: CDF of SwiftLinks operation latencies

Read operations Figure 1a shows that the *full* mode has greater cache hit rate (35%) than the *partial* mode. Nevertheless, the hit rate is not optimal due to the limit in the cache size: the cache cannot hold full replicas of all the forums and thus sometimes need to fetch them again. Figure 1c shows the results of a similar experiment but without any cache size limit. In that case, the cache hit rate, for the *full* mode, is 90%, which corresponds to our ratio of biased operations, and it confirms the previous results with a social network application of the SwiftCloud paper [10]. On the other hand, in *partial* mode, the cache hit rate is lower, with only 20% in both experiments (figures 1a and 1c), because the cache only holds partial replicas which gives it less chance of having all the parts needed for hitting the cache in subsequent operations. However, it has the advantage of a lower maximum latency: if an operation does not hit the cache, it only needs to fetch some parts, instead of the full object. In that scenario, it induces a delay similar to the cloud solution, around 200 to 300 ms, while without partial replication, the delay is increased to around 500 to 700 ms by having to replicate a full object. This poses a trade-off between the cache hit rate and the maximum latency. While fully replicating an object will provide more cache hits, a cache miss is more costly.

For the latency of reading comments of a post, shown in Figure 1b, the situation is a bit different. Clients are less likely to read the same comment tree multiple times; therefore, this affects the cache hit ratio. As the figure shows, the hit ratio is less than 5% in both *partial* and *full* replication. But again, *partial* replication has the advantage of reducing the impact of a cache miss as it only replicates the comments required by the operation instead of the full comment tree. In consequence, the *partial* approach has a slightly better latency, close to the *cloud* mode. The *cloud* mode performs better because it never needs to fetch any extra metadata, which means that the returned messages are considerable smaller. Notice that the difference between *full* and *partial* mode has been reduced in this experiment because the involved objects are smaller.

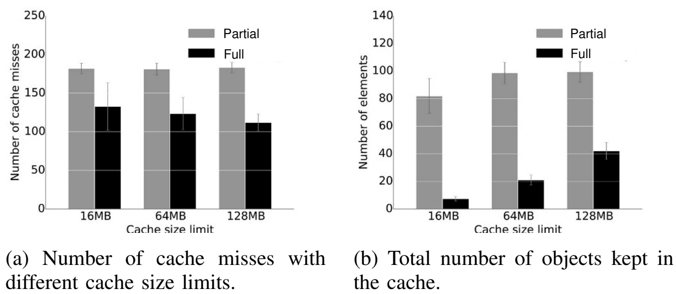
Update operations Caching modes (*full* and *partial*) are more beneficial with update operations. The reason is that update operations are typically applied on objects, or parts of objects, that have already been read by the client. In addition, the update operations only use version-independent queries to fetch their missing parts, which substantially simplifies the comparison of partial objects in the cache. Figure 1e proves experimentally our reasoning. While the cloud mode has an almost constant latency for all operations of a round-trip time, with caching modes, most of the operations (almost 90%) have no latency. Again, the *partial* mode has the advantage of reducing the latency when the cache is not hit, as it only needs to fetch the parts of the object that need to be updated, instead of the full object. Moreover, some updates can be done blindly, therefore, they are completed locally.

In particular, Figure 1d shows the benefit of updates when posting comments, which almost always only requires particles already present in the cache. One can see that with partial replication, all the operations have almost no latency, as they can be done completely asynchronously. In contrast, in *full* mode, there can be a large delay when the tree of comments is not in the cache, as it needs to be fetched from the store. As in previous scenarios, even if an operation cannot be done completely locally in *partial* mode, the client only has to fetch part of the tree to complete the update.

D. Impact of cache size limit

In this section we look at how the application performance changes with various cache size limits (16, 64, and 128MB).

1) Impact on latency: We have empirically demonstrated that the *partial* mode performs better without cache limit when reading links. We run the same experiments showed in figures 1a, 1b and 1e setting the cache size limit to 16MB and 128MB. The experiments show that a smaller cache (16MB) size limit has a big latency impact on reading links and updates in *full* mode. Nevertheless, its impact is considerable smaller in



(a) Number of cache misses with different cache size limits.

(b) Total number of objects kept in the cache.

Fig. 2: Impact of cache size limit in partial and full modes

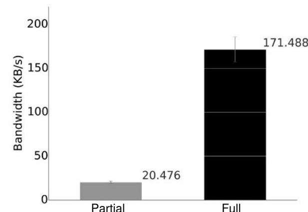
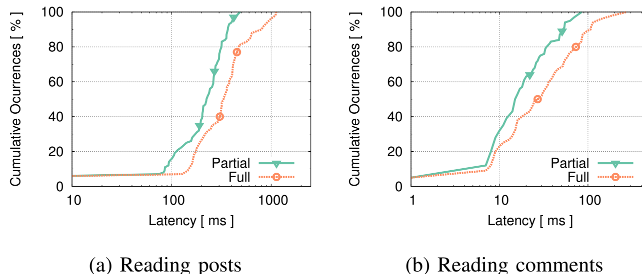


Fig. 3: Average bandwidth usage to fetch objects with a 128MB cache limit, with the cache already warmed up.



(a) Reading posts

(b) Reading comments

Fig. 4: Perceived latency of SwiftLinks during cache warm up.

partial mode. With a small cache, the cache hit rate of *full* mode of reading links becomes worse than in *partial* mode. This is because only a few objects can fit in the cache at a given time; therefore, clients need to fetch objects more frequently. This results in a lower fraction of operations having no latency, about 5% against the 35% obtained with a 64MB cache. There is also an impact for the *partial* mode, but it is considerable lower: it only drops to 13% from 20%. The same applies for update operations. Nevertheless, reads of comments are almost not impacted by the cache size limit: the operations have a low cache locality, so most operations need to fetch an object from the datacenter.

With a 128MB cache size limit, the *full* mode has a large portion of zero latency operations when reading posts, as more are kept in the cache. It however still performs worse than *partial* fetching for operations that do not hit the cache. The latency of updates also improves for the *full* mode with larger cache size, but the *partial* mode still outperforms it.

2) Impact on cache miss rate: The size limit imposed on the cache also has an impact on the cache hit rate. Figure 2a shows that the *partial* mode is less impacted by the cache size limit than the *full* mode. With the three cache limits, the *partial* mode shows a rather stable number of cache misses, about 180. Nevertheless, this does not apply to the *full* mode, where the number of caches misses increases as the cache size is reduced. As in previous experiments, the cache miss rate is greater in the *partial* mode. Nevertheless, we have shown that latency in *partial* mode, is always smaller in average.

3) Impact on number of objects in the cache: The cache size also impacts the number of objects that can be kept in the cache. Notice that for partial replication, only one object is counted even if multiple parts of it have been fetched over time. Figure 2b shows the difference between both modes: *partial* and *full*. In the *partial* mode, many more objects can fit in the cache at any moment, since only parts are kept. 64MB is enough to keep all the objects needed by the application, while in the *full* mode, even 128MB is not enough. This, depending on the workload, may increase the cache hit rate.

E. Bandwidth usage

In *partial mode*, when a client accesses an object, only the needed part of that object is fetched. This can result in saving bandwidth usage compared to *full* mode. In this experiment, we compare the bandwidth usage of *partial* mode and *full* mode. We measure the average bandwidth usage of one client for both over one minute, with the cache already warmed up. Figure 3 shows that the *partial* mode uses only about 12% of bandwidth compared to the *full* mode.

F. Cache warm up

The following experiments compare both *partial* and *full* modes latencies when the cache is still cold, i.e. no objects are stored in the client side. Figure 4 shows the latency of operations during the first 10 seconds of running the application, with a cold cache. In this case, the *partial* mode produces lower latencies as it does not need to replicate the full object. The difference is more noticeable for post reading operations, as shown in Figure 4a, as the set of posts (forums) are large objects. But even for smaller objects, such as comment trees, the *partial* mode outperforms the *full* one (Figure 4b). Notice that the cache size limit does not impact these experiments, since after 10 seconds, the cache does not get full.

G. Discussion

We have seen that partial replication has advantages over full replication of objects. First, it sets an upper bound on the latency of operations by limiting the amount of data that is fetched from the store. Plus, blind update operations gain the additional benefit of being applied locally even if the object is not cached. Second, the cache is more efficiently used, which allows more objects to be kept locally even with a small cache size limit. This is useful for memory-thin devices, and to work on very large data structures with a low memory usage. Third, partial replication also reduces the bandwidth usage of the application by a factor of 8, which is especially valuable on mobile wireless connections, such as EDGE and 3G. Finally, the last advantage is a lower cost of filling the cache when

starting the application. When the cache is empty all operations induce a cache miss, which is especially costly if a large object has to be fetched. Partial replication limits this issue by only replicating the parts of the object that are actually needed.

Unfortunately, *partial* mode limits the cache hit rate, as objects are not fully replicated right away, and non-replicated parts may be needed by subsequent operations. Thus, its use may depend on the workload and the cost of a cache miss. However, a tradeoff is possible between the two: instead of only fetching the parts needed by the operations, one could fetch extra parts of the object. This would however increase bandwidth and cache size utilisation. Latency could be kept low by asynchronously fetching the additional parts.

V. Related work

PRACTI [19] allows clients to select a subset of objects to replicate. Clients only receive updates on objects of their selected subset. However, clients are forced to keep some metadata about objects that they are not interested. Polyjuz [20] stores objects consisting of a set of fields. Clients can decide which fields of each object to replicate. Each subset of fields is denoted as fidelity level. Clients can select different fidelity levels according to the space or network limitations of the device where the objects are replicated. Polyjuz transparently handles the replication of an object in different fidelity levels. In Cimbiosys [21], objects are grouped into collections. Users can use filter expressions to only replicate objects that satisfy some criteria. For example, a user can group his emails in a collection and choose only to replicate emails from his university in his phone. While in the first two systems, users choose the object or fields to replicate based on their name or type, in Cimbiosys user can define replication criteria based on the value of some properties of objects.

VI. Conclusion and future work

We have introduced and formalized a new set of CRDTs called Conflict-free Partially Replicated Data Types, an extension of CRDTs which allows replicas to hold parts of data structures. Our extensive evaluation has shown that CPRDTs can improve the bandwidth and memory usage of replicas by only replicating parts needed by clients, specially in the presence of large data structures under limited cache sizes. Although cache sizes may be larger in the future, we believe that our reasoning will still apply and future applications will still benefit from the CPRDTs approach. The experimental study has also shown that CPRDTs reduce latency in average in comparison to the full mode. However, CPRDTs have a negative impact on the cache hit rate, which has to be weighted against the upper bound on the latency provided.

We plan to extend this work in several directions. First, partial replication can be used as a security mechanism to avoid replicating sensitive data by restricting access with finely grained rules. We believe it is an interesting way of exploiting CPRDTs. Second, we want to study how predictive caching techniques could still improve bandwidth usage and consequently reduce latency even more.

Acknowledgments: We thank Marek Zawirski for his help integrating CPRDTs into SwiftCloud. This work was partially funded by the SyncFree project in the European Seventh Framework Programme (FP7/2007-2013) under Grant Agreement n° 609551 and by the Erasmus Mundus Joint Doctorate Programme under Grant Agreement 2012-0030.

References

[1] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, Aug. 2013.

[2] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with cops," in *SOSP'11*. New York, NY, USA: ACM, 2011, pp. 401–416.

[3] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[4] E. Schurman and J. Brutlag, "The user and business impact of server delays, additional bytes, and http chunking in web search," in *Velocity Web Performance and Operations Conference*, June 2009.

[5] C. Jay, M. Glencross, and R. Hubbold, "Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment," *ACM Trans. Comput.-Hum. Interact.*, vol. 14, no. 2, Aug. 2007.

[6] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: a highly available file system for a distributed workstation environment," *IEEE Transactions on Computers*, vol. 39, no. 4, p. 447459, Apr 1990.

[7] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system*. ACM, 1995, vol. 29.

[8] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," INRIA, Rapport de recherche RR-7506, Jan. 2011.

[9] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Stabilization, Safety, and Security of Distributed Systems*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, vol. 6976, pp. 386–400.

[10] M. Zawirski, A. Bieniusa, V. Balegas, S. Duarte, C. Baquero, M. Shapiro, and N. M. Preguiça, "Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine," *CoRR*, vol. abs/1310.3107, 2013.

[11] "Amazon S3," http://aws.amazon.com/s3.

[12] "Google cloud storage," http://cloud.google.com/storage.

[13] "Windows Azure," http://www.microsoft.com/windowsazure.

[14] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *MobiSys'09*. Association for Computing Machinery, Inc., June 2009.

[15] "About reddit," http://www.reddit.com/about/, accessed: 2014-06-02.

[16] "Reddit source code," https://github.com/reddit/reddit, accessed: 2014-04-08.

[17] D. Navalho, S. Duarte, N. Preguiça, and M. Shapiro, "Incremental stream processing using computational conflict-free replicated data types," in *CloudDP'13*. New York, NY, USA: ACM, 2013, pp. 31–36.

[18] I. Briquemont, "Optimising client-side geo-replication with partially replicated data structures," Master's thesis, ICTEAM Institute, Universit catholique de Louvain, Sep. 2014.

[19] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng, "Practi replication," in *NSDI'06*. Berkeley, CA, USA: USENIX Association, 2006, pp. 5–5.

[20] K. Veeraraghavan, V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, and T. Wobber, "Fidelity-aware replication for mobile devices," in *MobiSys '09*. New York, NY, USA: ACM, 2009, pp. 83–94.

[21] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat, "Cimbiosys: A platform for content-based partial replication," in *NSDI'09*. Berkeley, CA, USA: USENIX Association, 2009, pp. 261–276.

6.3 Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Queue, January 2016. ACM.

Why Logical Clocks are Easy

SOMETIMES ALL YOU NEED IS THE RIGHT LANGUAGE

CARLOS BAQUERO AND NUNO PREGUIÇA

Any computing system can be described as executing sequences of actions, with an *action* being any relevant change in the state of the system. For example, reading a file to memory, modifying the contents of the file in memory, or writing the new contents to the file are relevant actions for a text editor. In a distributed system, actions execute in multiple locations; in this context, actions are often called events. Examples of events in distributed systems include sending or receiving messages, or changing some state in a node. Not all events are related, but some events can cause and influence how other, later events occur. For example, a reply to a received mail message is influenced by that message, and maybe by prior messages received.

Events in a distributed system can occur in a close location, with different processes running in the same machine, for example; or at nodes inside a data center; or geographically spread across the globe; or even at a larger scale in the near future. The relations of potential cause and effect between events are fundamental to the design of distributed algorithms. These days hardly any service can claim not to have some form of distributed algorithm at its core.

To make sense of these cause-and-effect relations, it is necessary to limit their scope to what can be perceived

A distributed system has no way of knowing that the first reservation has actually caused the second one

inside the distributed system itself—*internal causality*. Naturally, a distributed system interacts with the rest of the physical world outside of it, and there are also cause-and-effect relations in that world at large. For example, consider a couple planning a night out using a system that manages reservations for dinner and a movie. One person makes a reservation for dinner and lets the other person know with a phone call. After receiving the phone call, the second person goes to the system and reserves a movie. A distributed system has no way of knowing that the first reservation has actually caused the second one.

This *external causality* cannot be detected by the system and can only be approximated by *physical time*. (Time, however, totally orders all events, even those unrelated—thus, it is no substitute to causality—and wall clocks are never perfectly synchronized.[11,16]) This article focuses instead on internal causality—the type that can be tracked by the system.

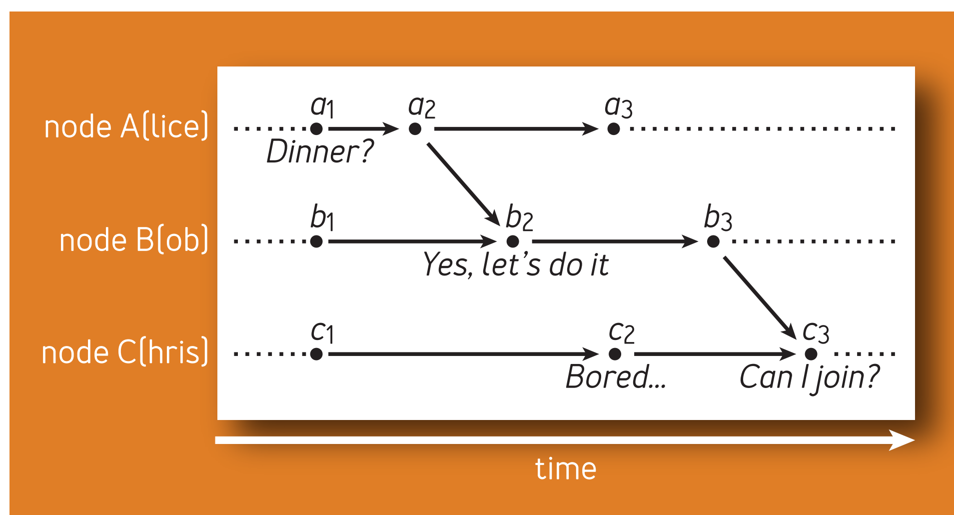
HAPPENED-BEFORE RELATION

In 1978 Leslie Lamport defined a partial order, referred to as *happened before*, that connects events of a distributed system that are potentially causally linked.[8] An event *c* can be the cause of an event *e*, or *c* happened before *e*, iff (if and only if) both occur in the same node and *c* executed first, or, being at different nodes, if *e* could know about the occurrence of *c* thanks to some message received from some node that knows about *c*. If neither event can know about the other, then they are said to be concurrent.

Using the example of dinner and movie reservations, figure 1 shows a distributed system with three nodes. An arrow between nodes represents a message sent and delivered. Both Bob's positive answer to the dinner suggestion by Alice and Chris's later request to join the party are influenced by Alice's initial question about plans for dinner.

In this distributed computation, a simple way to check if an event *c* could have caused another event *e* (*c* happened before *e*) is to find at least one directed path linking *c* to *e*. If such a connection is found, this partial order relation is marked $c \rightarrow e$ to denote the happened-before relation or potential causality. Figure 1 has $a_1 \rightarrow b_2$ and $b_3 \rightarrow c_3$ (and, yes, also $a_1 \rightarrow c_3$, since causality is transitive). Events a_1 and c_2 are concurrent (denoted $a_1 \parallel c_2$), because there are no causal paths in either direction. Note $x \parallel y$ if and only if $x \nrightarrow y$ and $y \nrightarrow x$. The fact that Chris was bored neither influenced Alice's question about dinner, nor the other way around.

FIGURE 1: **HAPPENED-BEFORE RELATION**



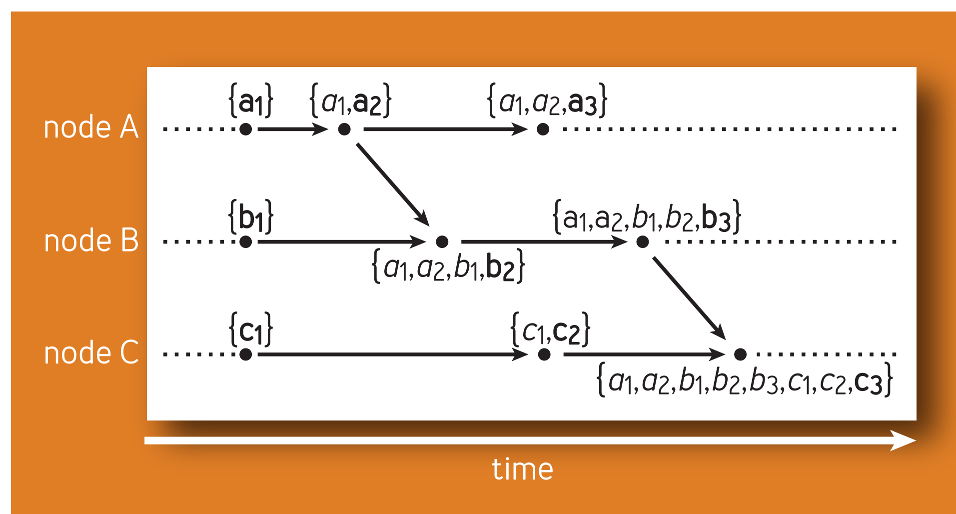
Thus, the three possible relations between two events *x* and *y* are: (a) *x* might have influenced *y*, if $x \rightarrow y$; (b) *y* might have influenced *x*, if $y \rightarrow x$; (c) there is no known influence between *x* and *y*, as they occurred concurrently $x \parallel y$.

CAUSAL HISTORIES

Causality can be tracked in a very simple way by using *causal histories*.[3,14] The system can locally assign unique names to each event (e.g., node name and local increasing counter) and collect and transmit sets of events to capture the known past.

For a new event, the system creates a new unique name, and the causal history consists of the union of this name and the causal history of the previous event in the node. For example, the second event in node *C* is assigned the name c_2, and its causal history is $H_c = \{c_1, \mathbf{c_2}\}$ (shown in figure 2). When a node sends a message, the causal history of the send event is sent with the message. When the message is received, the

FIGURE 2: **CAUSAL HISTORIES**



remote causal history is merged (by set union) with the local history. For example, the delivery of the first message from node A to B merges the remote causal history $\{a_1, a_2\}$ with the local history $\{b_1\}$ and the new unique name b_2, leading to $\{a_1, a_2, b_1, \mathbf{b_2}\}$.

Checking causality between two events x and y can be tested simply by set inclusion: $x \rightarrow y$ iff $H_x \subsetneq H_y$. This follows from the definition of causal histories, where the causal history of an event will be included in the causal history of the following event. Even better, marking the last local event added to the history (distinguished in bold in the figure) allows the use of a simpler test: $x \rightarrow y$ iff $x \in H_y$ (e.g., $a_1 \rightarrow b_2$, since $a_1 \in \{a_1, a_2, b_1, b_2\}$). This follows from the fact that a causal history includes all events that (causally) precede a given event.

VECTOR CLOCKS

It should be obvious by now that causal histories work but are not very compact. This problem can be addressed by relying on the following observation: the mechanism of building the causal history implies that if an event b_3 is present in H_y, then all preceding events from that same node, b_1 and b_2, are also present in H_y. Thus, it suffices to store the most recent event from each node. Causal history $\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\}$ is compacted to $\{a \longmapsto 2, b \longmapsto 3, c \longmapsto 3\}$ or simply a vector [2, 3, 3].

Now the rules used with causal histories can be translated to the new compact vector representation.

Verifying that $x \rightarrow y$ requires checking if $H_x \subsetneq H_y$. This

can be done, verifying for each node, if the unique names contained in H_x are also contained in H_y and there is at least one unique name in H_y that is not contained in H_x. This is immediately translated to checking if each entry in the vector of x is smaller or equal to the corresponding entry in the vector of y and one is strictly smaller (i.e., $\forall i : V_x[i] \leq V_y[i]$ and $\exists j : V_x[j] < V_y[j]$). This can be stated more compactly as $x \rightarrow y$ iff $V_x < V_y$.

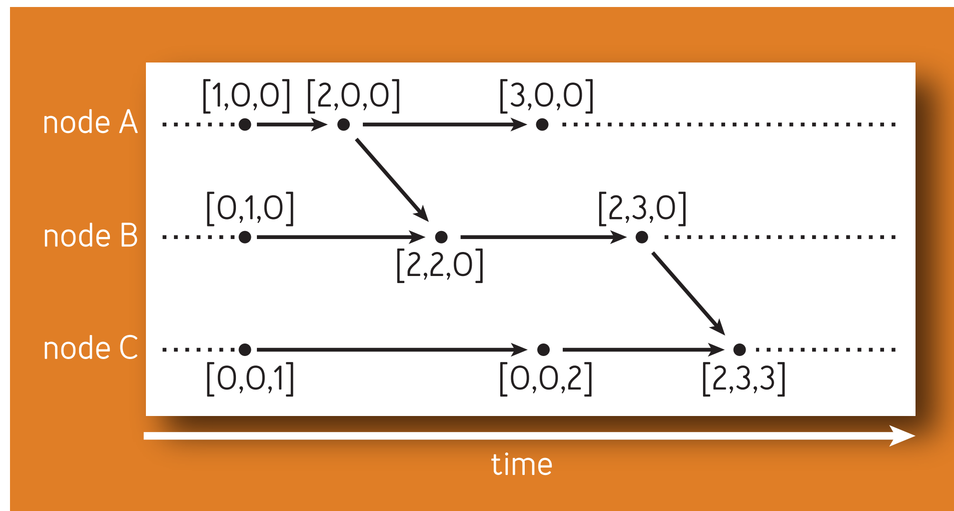
For a new event the creation of a new unique name is equivalent to incrementing the entry in the vector for the node where the event is created. For example, the second event in node C has vector [0, 0, 2], which corresponds to the creation of event c_2 of the causal history.

Finally, creating the union of the two causal histories H_x and H_y is equivalent to taking the pointwise maximum of the corresponding two vectors V_x and V_y (i.e., $\forall i : V[i] = \max(V_x[i], V_y[i])$). Logic tells us that, for the unique names generated in each node, only the one with the largest counter needs to be kept.

When a message is received, in addition to merging the causal histories, a new event is created. The vector representation of these steps can be seen, for example, when the first message from *a* is received in *b*, where taking the pointwise maximum leads to [2, 1, 0] and the new unique name finally leads to [2, 2, 0], as shown in figure 3.

This compact representation, known as a *vector clock,* was introduced around 1988.[5,10] Vector comparison is an immediate translation of set inclusion of causal histories. This equivalence is often forgotten in modern descriptions of

FIGURE 3: **VECTOR CLOCKS**



vector clocks and can turn what is a simple encoding problem into an unnecessarily complex and arcane set of rules, going against logic.

DOTTED VECTOR CLOCKS

As shown thus far, when using causal histories, knowing the last event could simplify comparison by simply checking if the last event is included in the causal history. This can still be done with vectors, if you keep track of the node in which the last event has been created. For example, when questioning if $x = [\mathbf{2}, 0, 0] \rightarrow y = [2, \mathbf{3}, 0]$, with boldface indicating the last event in each vector, you can simply test if $x[0] \leq y[0]$ ($\mathbf{2} \leq 2$) since you have marked that the last event in x was created in node A (i.e., it corresponds to the first entry of the vector). Since marking numbers in bold is not a practical implementation, however, the last event is usually stored outside the vector (and is sometimes called a *dot*): for example, $[2, \mathbf{2}, 0]$ can be represented as $[2, 1, 0]b_2$. Notice that

now the vector represents the causal past of b_2, excluding the event itself.

VERSION VECTORS

In an important class of applications there is no need to register causality for all the events in a distributed computation. For example, to modify replicas of data, it often suffices to register only those events that change replicas. In this case, when thinking about causal histories, you need only to assign a new unique name to these relevant events. Still, you need to propagate the causal histories when messages are propagated from one site to another, and the remaining rules for comparing causal histories remain unchanged.

Figure 4 presents the same example as before, but now with events that are not registered for causality tracking denoted with ∘. If the run represents the updates to replicas of a data object, then after nodes *A* and *B* are concurrently modified, the state of replica *a* is sent to replica *b* (in a

FIGURE 4: **CAUSAL HISTORIES WITH ONLY SOME RELEVANT EVENTS**

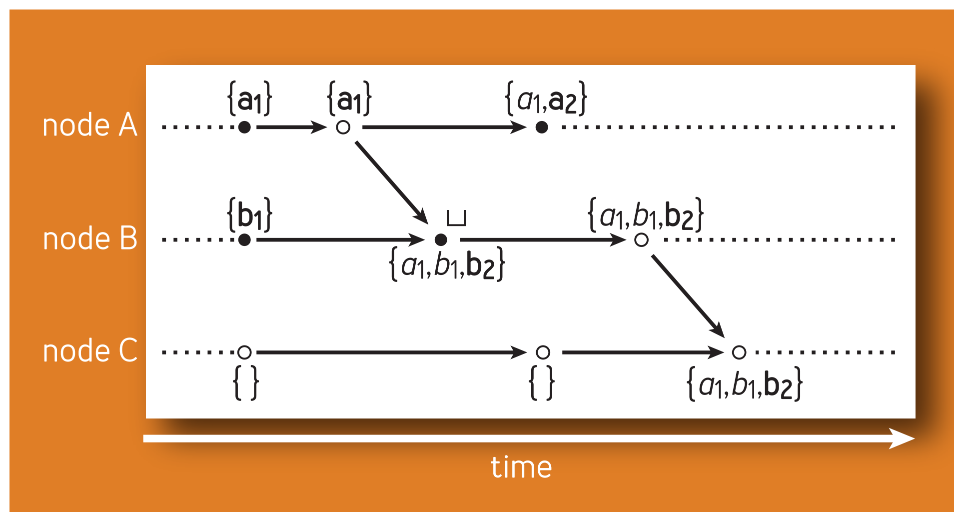
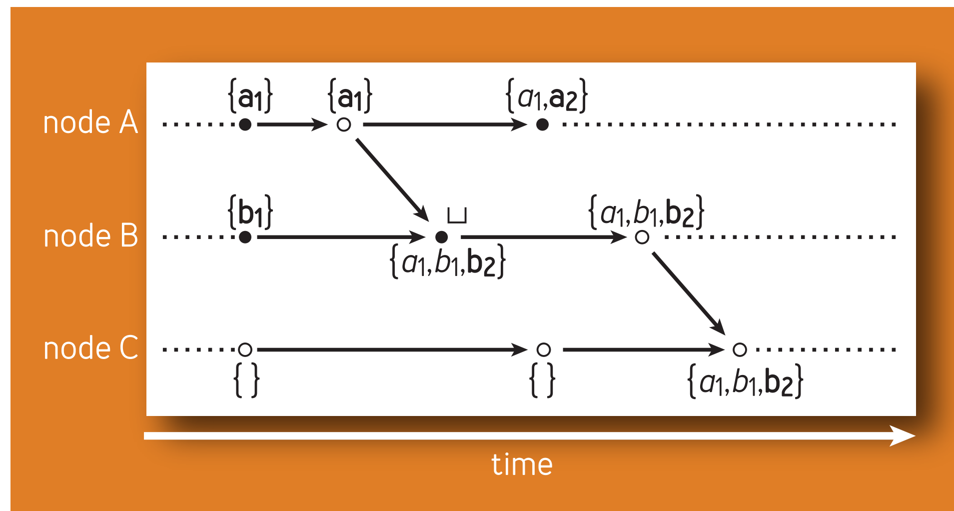


FIGURE 4: **CAUSAL HISTORIES WITH ONLY SOME RELEVANT EVENTS**



message). When the message is received in node *B*, it is detected that two concurrent updates have occurred, with histories $\{a_1\}$ and $\{b_1\}$, as neither $a_1 \rightarrow b_1$ nor $b_1 \rightarrow a_1$. In this case, a new version that merges the two updates is created (merge is denoted by the join symbol $\sqcup$), which requires creating a new unique name, leading to $\{a_1, b_1, b_2\}$. When the state of replica *b* is later propagated to replica *c*, as no concurrent update exists in replica *c*, no new version is created.

Again, vectors can compact the representation. The result, known as a *version vector,* was created in 1983,[12] five years before vector clocks. Figure 5 presents the same example as before, represented with version vectors.

In some cases when the state of one replica is propagated to another replica, the two versions are kept by the system as conflicting versions. For example, in figure 6, when the message from node *A* is received in node *B*, the system keeps each causal history $\{a_1\}$ and $\{b_1\}$ associated with the

FIGURE 5: **VERSION VECTORS WITH ONLY SOME RELEVANT EVENTS**

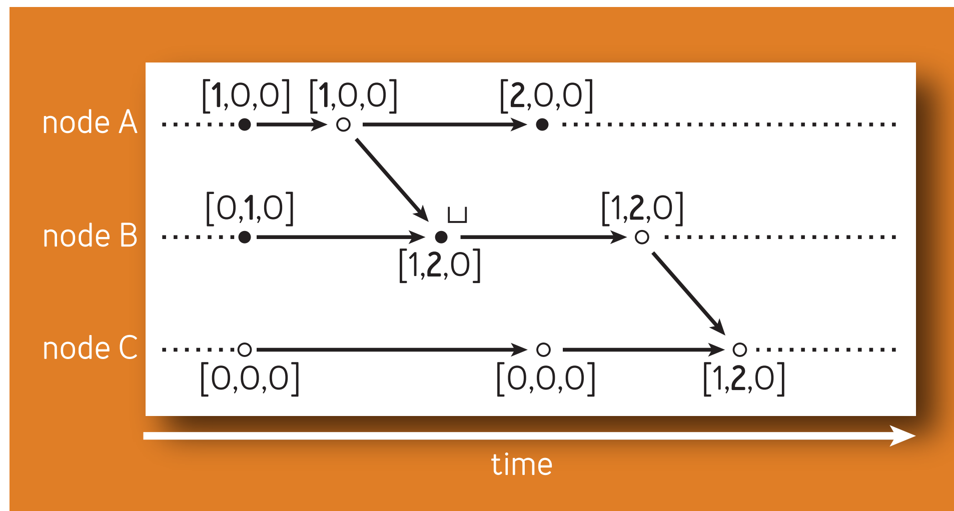
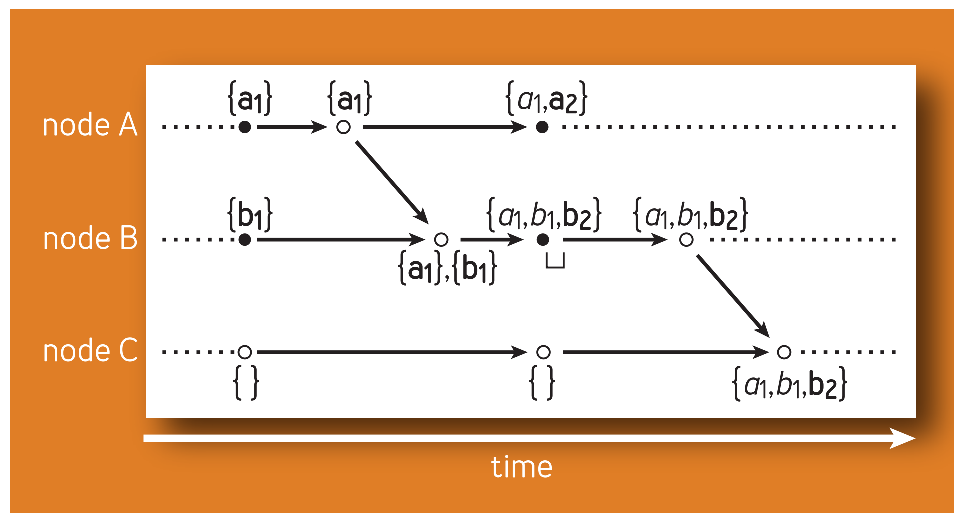


FIGURE 6: **CAUSAL HISTORIES WITH VERSIONS NOT IMMEDIATELY MERGED**



respective version. The causal history associated with the node containing both versions is $\{a_1, b_1\}$, the union of the causal history of all versions. This approach allows later checking for causality relations between each version and other versions when merging the states of additional nodes. The conflicting versions could also be merged, creating a new unique name, as in the example.

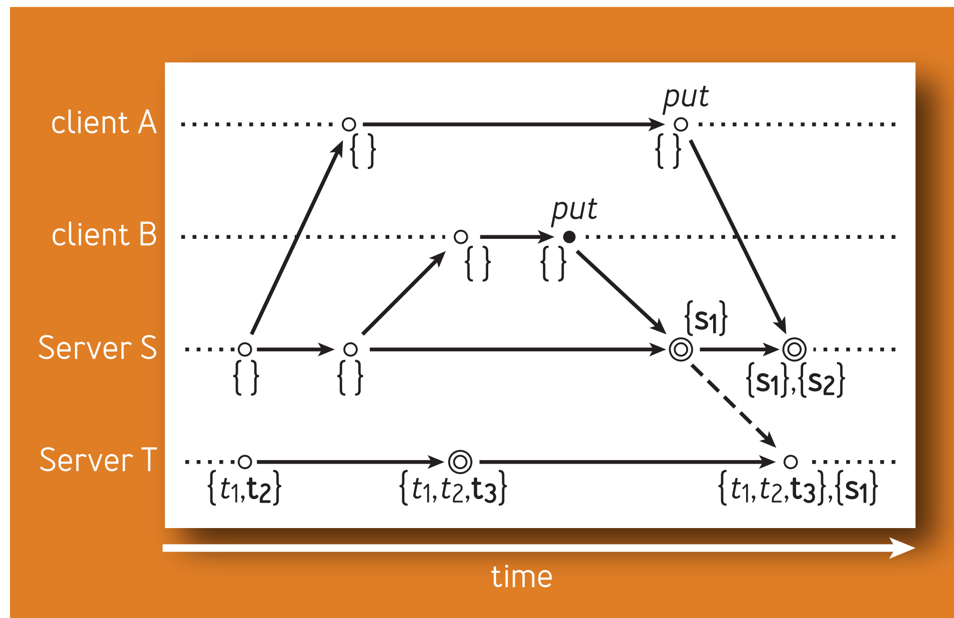
DOTTED VERSION VECTORS

One limitation of causality tracking by vectors is that one entry is needed for each source of concurrency.[4] You can expect a difference of several orders of magnitude between the number of nodes in a data center and the number of clients they handle. Vectors with one entry per client don't scale well when millions of clients are accessing the service.[7] Again, a look at the foundation of causal histories shows how to overcome this limitation.

The basic requirement in causal histories is that each event be assigned a unique identifier. There is no requirement that this unique identifier be created locally or immediately. Thus, in systems where nodes can be divided into clients and servers and where clients communicate only with servers, it is possible both to delay the creation of a new unique name until the client communicates with the server and to use a unique name generated in the server. The causal history associated with the new version is the union of the causal history of the client and the newly assigned unique name.

Figure 7 shows an example where clients A and B concurrently update server S. When client B first writes its version, a new unique name, s_1, is created (in the figure this action is denoted by the symbol ⊚) and merged with the causal history read by the client {}, leading to the causal history $\{s_1\}$. When client A later writes its version, the causal history assigned to this version is the causal history at the client, {}, merged with the new unique name s_2, leading to $\{s_2\}$. Using the normal rules for checking for concurrent updates, these two versions are concurrent. In the example, the

FIGURE 7: **CAUSAL HISTORIES IN A DISTRIBUTED STORAGE SYSTEM**



system keeps both concurrent updates. For simplicity, the interactions of server T with its own clients were omitted, but as shown in the figure, before receiving data from server S, server T had a single version that depicted three updates it managed—causal history $\{t_1, t_2, \mathbf{t_3}\}$—and after that it holds two concurrent versions.

One important observation is that in each node, the union of the causal histories of all versions includes all generated unique names until the last known one: for example, in server S, after both clients send their new versions, all unique names generated in S are known. Thus, the causal past of any update can always be represented using a compact vector representation, as it is the union of all versions known at some server when the client read the object. The combination of the causal past represented as a vector and

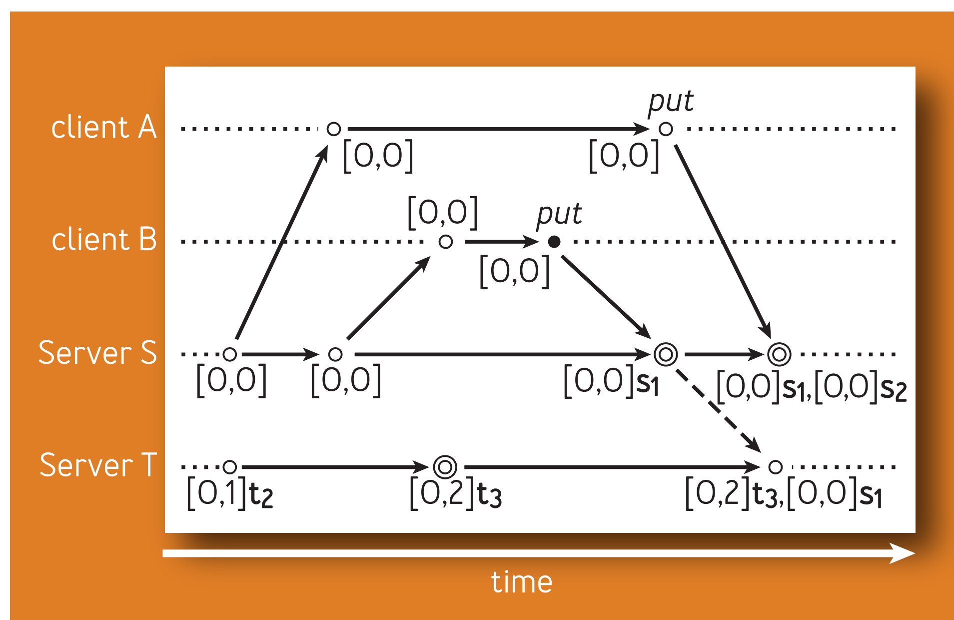
the last event, kept outside the vector, is known as a *dotted version vector*.[2,13] Figure 8 shows the previous example using this representation, which, as the system keeps running, eventually becomes much more compact than causal histories.

In the condition expressed before (clients communicate only with servers and a new update overwrites all versions previously read), which is common in key-value stores where multiple clients interact with storage nodes via a *get/put* interface, the dotted version vectors allow causality to be tracked between the written versions with vectors of the size of the number of servers.

FINAL REMARKS

Tracking causality should not be ignored. It is important in the design of many distributed algorithms. And not

FIGURE 8: **DOTTED VERSION VECTORS IN DISTRIBUTED STORAGE SYSTEM**



respecting causality can lead to strange behaviors for users, as reported by multiple authors.[1,9]

The mechanisms for tracking causality and the rules used in these mechanisms are often seen as complex,[6,15] and their presentation is not always intuitive. The most commonly used mechanisms for tracking causality—vector clocks and version vectors—are simply optimized representations of causal histories, which are easy to understand.

By building on the notion of causal histories, you can begin to see the logic behind these mechanisms, to identify how they differ, and even consider possible optimizations. When confronted with an unfamiliar causality-tracking mechanism, or when trying to design a new system that requires it, readers should ask two simple questions: (a) Which events need tracking? (b) How does the mechanism translate back to a simple causal history?

Without a simple mental image for guidance, errors and misconceptions become more common. Sometimes, all you need is the right language.

Acknowledgments
We would like to thank Rodrigo Rodrigues, Marc Shapiro, Russell Brown, Sean Cribbs, and Justin Sheehy for their feedback. This work was partially supported by EU FP7 SyncFree project (609551) and FCT/MCT projects UID/CEC/04516/2013 and UID/EEA/50014/2013.

References
1. Ajoux, P., Bronson, N., Kumar, S., Lloyd, W., Veeraraghavan,

K. 2015. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, Switzerland. Usenix Association.

2. Almeida, P. S., Baquero, C., Gonçalves, R., Preguiça, N. M., Fonte, V. 2014. Scalable and accurate causality tracking for eventually consistent stores. In *Proceedings of the Distributed Applications and Interoperable Systems,* held as part of the Ninth International Federated Conference on Distributed Computing Techniques, Berlin, Germany: 67–81.
3. Birman, K. P., Joseph, T. A. 1987. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems* 5(1): 47–76.
4. Charron-Bost, B. 1991. Concerning the size of logical clocks in distributed systems. *Information Processing Letters* 39(1): 11–16.
5. Fidge, C. J. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10(1): 56–66.
6. Fink, B. 2010. Why vector clocks are easy. Basho Blog; http://basho.com/posts/ technical/why-vector-clocks-are-easy/.
7. Hoff, T. 2014. How League of Legends scaled chat to 70 million players—it takes lots of minions. High Scalability; http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.
8. Lamport, L. 1978. Time, clocks, and the ordering of events

in a distributed system. *Communications of the ACM* 21(7): 558–565.

9. Lloyd, W., Freedman, M. J., Kaminsky, M., Andersen, D. G. 2011. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, New York, NY: 401–416.
10. Mattern, F. 1988. Virtual time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, Gers, France: 215– 226.
11. Neville-Neil, G. 2015. Time is an illusion. acmqueue 13(9). 57 - 72
12. Parker, D.S., Popek, G. J., Rudisin, G., Stoughton, A., Walker, B. J., Walton, E., Chow, J. M., Edwards, D., Kiser, S., Kline, C. 1983. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering* 9(3): 240–247.
13. Preguiça, N. M., Baquero, C., Almeida, P. S., Fonte, V., Gonçalves, R. 2012. Brief announcement: efficient causality tracking in distributed storage systems with dotted version vectors. In *ACM Symposium on Principles of Distributed Computing,* eds. D. Kowalski and A. Panconesi: 335–336.
14. Schwarz, R., Mattern, F. 1994. Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distributed Computing* 7(3): 149–174.
15. Sheehy, J. 2010. Why vector clocks are hard. Basho Blog;

http://basho.com/posts/ technical/why-vector-clocks-are-hard/.
16. Sheehy, J. 2015. There is no now. *acmqueue* 13(3): 20-27.

LOVE IT, HATE IT? LET US KNOW feedback@queue.acm.org

Carlos Baquero *is assistant professor of computer science and senior researcher at the High-Assurance Software Laboratory, Universidade do Minho and INESC Tec. He obtained his MSc and PhD degrees from Minho Universidade do in 1994 and 2000. His research interests are focused on distributed systems, in particular causality tracking, data types for eventual consistency, and distributed data aggregation. He can be reached at cbm@di.uminho.pt and as @xmal on Twitter.*

Nuno Preguiça *is associate professor in the Department of Computer Science, Faculty of Science and Technology, Universidade NOVA de Lisboa, and leads the computer systems group at NOVA Laboratory for Computer Science and Informatics. He received a PhD in computer science from Universidade NOVA de Lisboa in 2003. His research interests are focused on the problems of replicated data management and processing of large amounts of information in distributed systems and mobile computing settings. He can be reached at nuno.preguica@fct.unl.pt and as @nunopreguica on Twitter.*
Copyright © 2016 held by owner/author. Publication rights licensed to ACM.

6.4 Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. Commun. ACM, April 2016. ACM.

DOI:10.1145/2890782

Article development led by acmqueue
queue.acm.org

Sometimes all you need is the right language.

BY CARLOS BAQUERO AND NUNO PREGUIÇA

Why Logical Clocks Are Easy

ANY COMPUTING SYSTEM can be described as executing sequences of actions, with an *action* being any relevant change in the state of the system. For example, reading a file to memory, modifying the contents of the file in memory, or writing the new contents to the file are relevant actions for a text editor. In a distributed system, actions execute in multiple locations; in this context, actions are often called events. Examples of events in distributed systems include sending or receiving messages, or changing some state in a node. Not all events are related, but some events can cause and influence how other, later events occur. For example, a reply to a received email message is influenced by that message, and maybe by prior messages received.

Events in a distributed system can occur in a close location, with different processes running in the same machine, for example; or at nodes inside a datacenter; or geographically spread across the globe; or even at a larger scale in the near future. The relations of potential cause and effect between events are fundamental to the design of distributed algorithms. These days hardly any service can claim not to have some form of distributed algorithm at its core.

To make sense of these cause-and-effect relations, it is necessary to limit their scope to what can be perceived inside the distributed system itself—*internal causality.* Naturally, a distributed system interacts with the rest of the physical world outside of it, and there are also cause-and-effect relations in that world at large. For example, consider a couple planning a night out using a system that manages reservations for dinner and a movie. One person makes a reservation for dinner and lets the other person know with a phone call. After receiving the phone call, the second person goes to the system and reserves a movie. A distributed system has no way of knowing the first reservation has actually caused the second one.

This *external causality* cannot be detected by the system and can only be approximated by *physical time*. (Time, however, totally orders all events, even those unrelated—thus, it is no substi-

tute for causality—and wall clocks are never perfectly synchronized.[11,16]) This article focuses instead on internal causality—the type that can be tracked by the system.

Happened-Before Relation

In 1978, Leslie Lamport defined a partial order, referred to as *happened before*, that connects events of a distributed system that are potentially causally linked.[8] An event c can be the cause of an event e, or c happened before e, iff (if and only if) both occur in the same node and c executed first, or, being at different nodes, if e could know about the occurrence of c thanks to some message received from some node that knows about c. If neither event can know about the other, then they are said to be concurrent.

Using the example of dinner and movie reservations, Figure 1 shows a distributed system with three nodes. An arrow between nodes represents a message sent and delivered. Both Bob's positive answer to the dinner suggestion by Alice and Chris's later request to join the party are influenced by Alice's initial question about plans for dinner.

Figure 1. Happened-before relation.

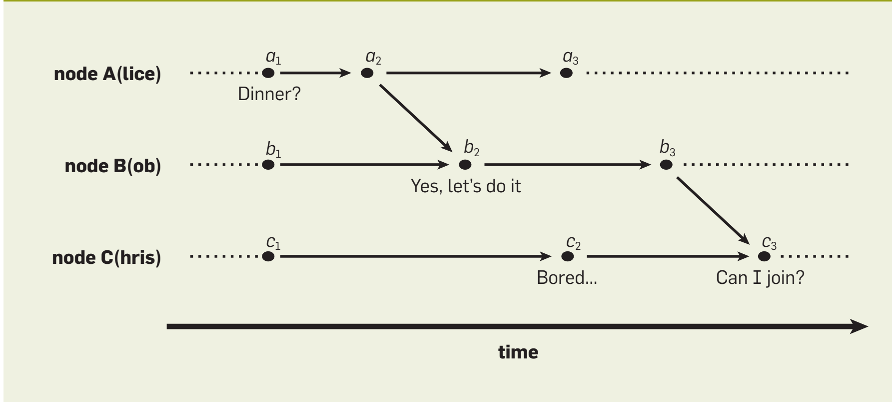


Figure 2. Causal histories.

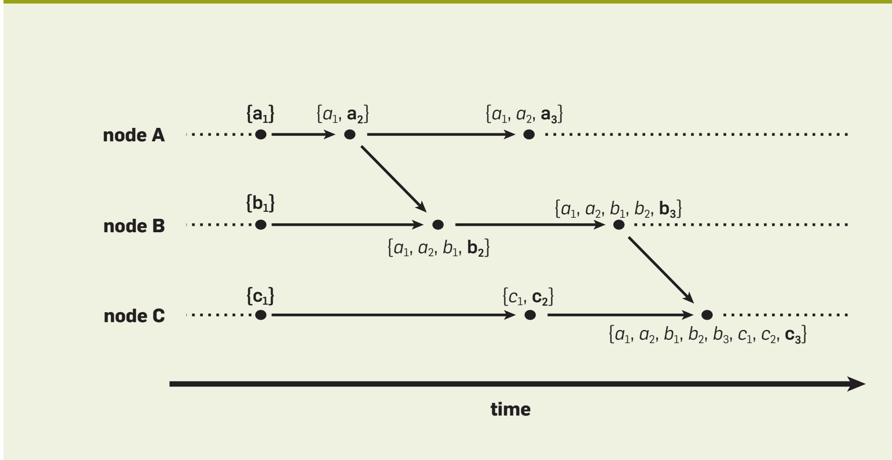
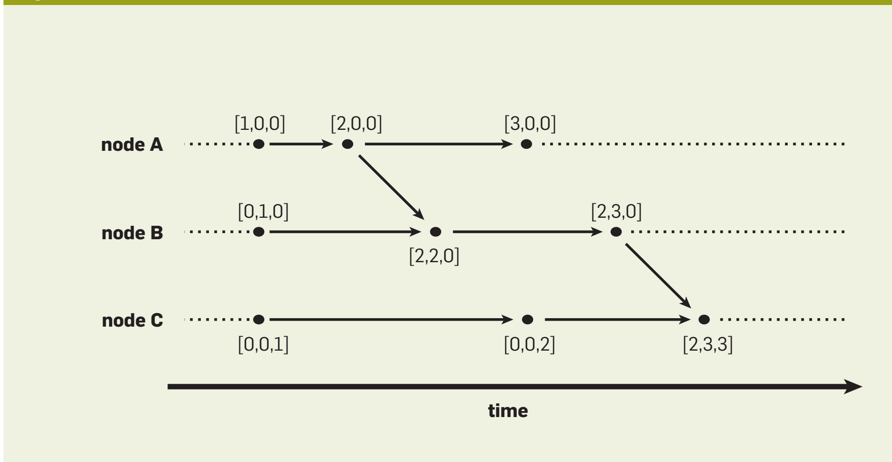


Figure 3. Vector clocks.



In this distributed computation, a simple way to check if an event c could have caused another event e (c happened before e) is to find at least one directed path linking c to e. If such a connection is found, this partial order relation is marked $c \rightarrow e$ to denote the happened-before relation or potential causality. Figure 1 has $a_1 \rightarrow b_2$ and $b_2 \rightarrow c_3$ (and, yes, also $a_1 \rightarrow c_3$, since causality is transitive). Events a_1 and c_2 are concurrent (denoted $a_1 \parallel c_2$), because there are no causal paths in either direction. Note $x \parallel y$ if and only if $x \nrightarrow y$ and $y \nrightarrow x$. The fact Chris was bored neither influenced Alice's question about dinner, not the other way around.

Thus, the three possible relations between two events x and y are: (a) x might have influenced y, if $x \rightarrow y$; (*b*) y might have influenced x, if $y \rightarrow x$; (*c*) there is no known influence between x and y, as they occurred concurrently $x \parallel y$.

Causal Histories

Causality can be tracked in a very simple way by using *causal histories*.[3,14] The system can locally assign unique names to each event (for example, node name and local increasing counter) and collect and transmit sets of events to capture the known past.

For a new event, the system creates a new unique name, and the causal history consists of the union of this name and the causal history of the previous event in the node. For example, the second event in node C is assigned the name c_2, and its causal history is $H_c = \{c_1, \mathbf{c}_2\}$ (shown in Figure 2). When a node sends a message, the causal history of the send event is sent with the message. When the message is received, the remote causal history is merged (by set union) with the local history. For example, the delivery of the first message from node A to B merges the remote causal history $\{a_1, a_2\}$ with the local history $\{b_1\}$ and the new unique name b_2, leading to $\{a_1, a_2, b_1, \mathbf{b}_2\}$.

Checking causality between two

events x and y can be tested simply by set inclusion: $x \rightarrow y$ iff $H_x \in H_y$. This follows from the definition of causal histories, where the causal history of an event will be included in the causal history of the following event. Even better, marking the last local event added to the history (distinguished in bold in the figure) allows the use of a simpler test: $x \rightarrow y$ iff $x \in H_y$ (for example, $a_1 \rightarrow b_2$, since $a_1 \in \{a_1, a_2, b_1, b_2\}$). This follows from the fact a causal history includes all events that (causally) precede a given event.

Vector Clocks

It should be obvious by now that causal histories work but are not very compact. This problem can be addressed by relying on the following observation: the mechanism of building the causal history implies if an event b_3 is present in H_y, then all preceding events from that same node, b_1 and b_2, are also present in H_y. Thus, it suffices to store the most recent event from each node. Causal history $\{a_1, a_2, b_1, b_2, b_3, c_1, c_2, c_3\}$ is compacted to $\{a \mapsto 2, b \mapsto 3, c \mapsto 3\}$ or simply a vector [2, 3, 3].

Now the rules used with causal histories can be translated to the new compact vector representation.

Verifying that $x \rightarrow y$ requires checking if $H_x \subsetneq H_y$. This can be done, verifying for each node, if the unique names contained in H_x are also contained in H_y and there is at least one unique name in H_y that is not contained in H_x. This is immediately translated to checking if each entry in the vector of x is smaller or equal to the corresponding entry in the vector of y and one is strictly smaller (such as, $\forall i : V_x[i] \leq V_y[i]$ and $\exists j : V_x[j] < V_y[j]$). This can be stated more compactly as $x \rightarrow y$ iff $V_x < V_y$.

For a new event the creation of a new unique name is equivalent to incrementing the entry in the vector for the node where the event is created. For example, the second event in node C has vector [0, 0, 2], which corresponds to the creation of event c_2 of the causal history.

Finally, creating the union of the two causal histories H_x and H_y is equivalent to taking the pointwise maximum of the corresponding two vectors V_x and V_y (such as, $\forall i : V[i] = \mathbf{max}(V_x[i], V_y[i])$). Logic tells us that, for the unique names generated in each node, only the one with the largest counter needs to be kept.

When a message is received, in addition to merging the causal histories, a new event is created. The vector representation of these steps can be seen, for example, when the first message from a is received in b, where taking the pointwise maximum leads to [2, 1, 0] and the new unique name finally leads to [2, 2, 0], as shown in Figure 3.

This compact representation, known as a *vector clock*, was introduced around 1988.[5,10] Vector comparison is an immediate translation of set inclusion of causal histories. This equivalence is often forgotten in modern descriptions of vector clocks and can turn what is a simple encoding problem into an unnecessarily complex and arcane set of rules, going against logic.

As shown thus far, when using causal histories, knowing the last event could simplify comparison by simply checking if the last event is included in the causal history. This can still be done with vectors, if you keep track of the node in which the last event has been created. For example, when questioning if $x = [\mathbf{2}, 0, 0] \rightarrow y = [2, \mathbf{3}, 0]$, with boldface indicating the last event in each vector, you can simply test if $x[0] \leq y[0]$ ($\mathbf{2} \leq 2$) since you have marked the last event in x was created in node A (that is, it corresponds to the first entry of the vector). Since marking numbers in bold is not a practical implementation, however, the last event is usually stored outside the vector (and is sometimes called a *dot*): for example, [2, **2**, 0] can be represented as $[2, 1, 0]b_2$. Notice that now the vector represents the causal past of b2, excluding the event itself.

In an important class of applications there is no need to register causality for all the events in a distributed computation. For example, to modify replicas of data, it often suffices to register only those events that change replicas. In this case, when thinking about causal histories, you need only to assign a new unique name to these relevant events. Still, you need to propagate the causal histories when messages are propagated from one site to another and the remaining rules for comparing causal histories remain unchanged.

Figure 4 presents the same example as before, but now with events that are not registered for causality tracking denoted with $\circ$. If the run represents the updates to replicas of a data object,

Figure 4. Causal histories with only some relevant events.

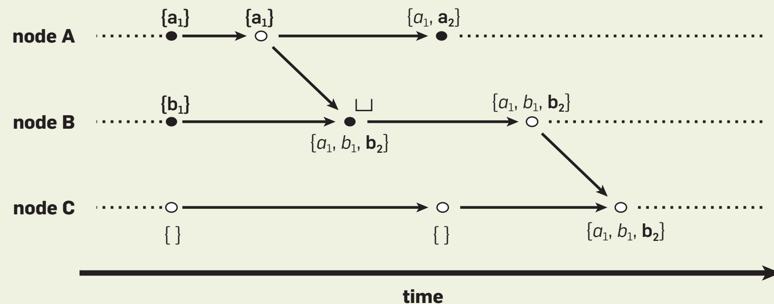
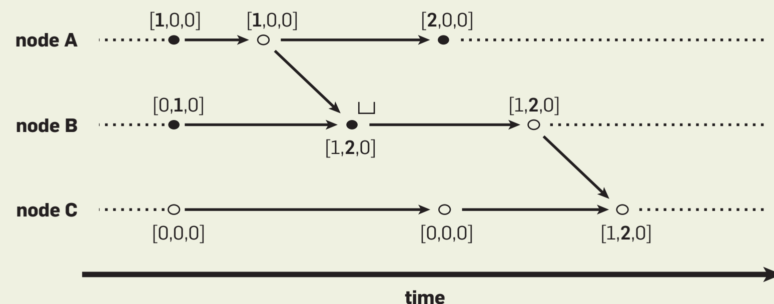


Figure 5. Version vectors with only some relevant events.



then after nodes A and B are concurrently modified, the state of replica a is sent to replica b (in a message). When the message is received in node B, it is detected two concurrent updates have occurred, with histories $\{a_1\}$ and $\{b_1\}$, as neither $a_1 \rightarrow b_1$ nor $b_1 \rightarrow a_1$. In this case, a new version that merges the two updates is created (merge is denoted by the join symbol $\sqcup$), which requires creating a new unique name, leading to $\{a_1, b_1, b_2\}$. When the state of replica b is later propagated to replica c, as no concurrent update exists in replica c, no new version is created.

Again, vectors can compact the representation. The result, known as a version vector, was created in 1983,[12] five years before vector clocks. Figure 5 presents the same example as before, represented with version vectors.

In some cases when the state of one replica is propagated to another replica, the two versions are kept by the system as conflicting versions. For example, in Figure 6, when the message from node A is received in node B, the system keeps each causal history $\{a_1\}$ and $\{b_1\}$ associated with the respective version. The causal history associated with the node containing both versions is $\{a_1, b_1\}$, the union of the causal history of all versions. This approach allows later checking for causality relations between each version and other versions when merging the states of additional nodes. The conflicting versions could also be merged, creating a new unique name, as in the example.

Figure 6. Causal histories with versions not immediately merged.

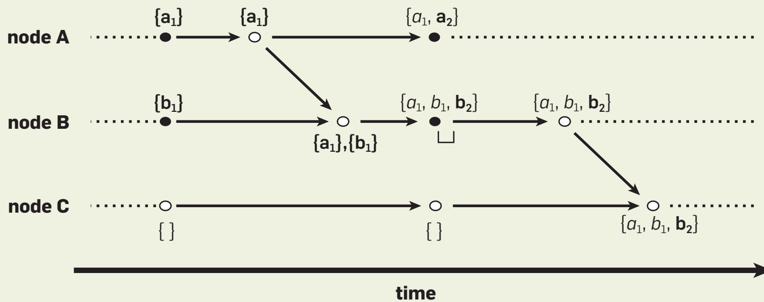


Figure 7. Causal histories in a distributed storage system.

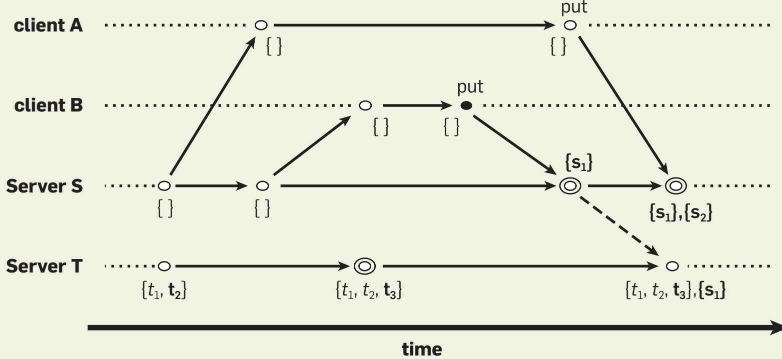
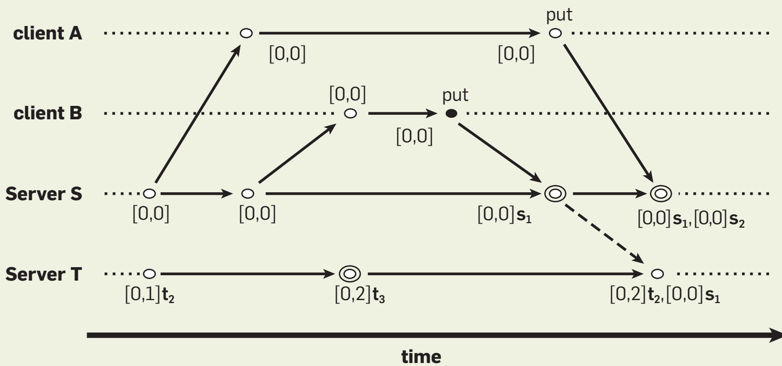


Figure 8. Dotted version vectors in distributed storage system.



One limitation of causality tracking by vectors is that one entry is needed for each source of concurrency.[4] You can expect a difference of several orders of magnitude between the number of nodes in a datacenter and the number of clients they handle. Vectors with one entry per client do not scale well when millions of clients are accessing the service.[7] Again, a look at the foundation of causal histories shows how to overcome this limitation.

The basic requirement in causal histories is each event be assigned a unique identifier. There is no requirement this unique identifier be created locally or immediately. Thus, in systems where nodes can be divided into clients and servers and where clients communicate only with servers, it is possible both to delay the creation of a new unique name until the client communicates with the server and to use a unique name generated in the server. The causal history associated with the new version is the union of the causal history of the client and the newly assigned unique name.

Figure 7 shows an example where clients A and B concurrently update server S. When client B first writes its version, a new unique name, s_1, is created (in the figure this action is denoted by the symbol ⊚) and merged with the causal history read by the client $\{\}$, leading to the causal history $\{\mathbf{s_1}\}$. When client A later writes its version, the causal history assigned to this version is the causal history at the client, $\{\}$, merged with the new unique name s_2, leading to $\{\mathbf{s_2}\}$. Using the normal rules for checking for concurrent updates, these two versions are concurrent. In the example, the system keeps both concurrent updates. For simplicity, the

interactions of server T with its own clients were omitted, but as shown in the figure, before receiving data from server S, server T had a single version that depicted three updates it managed—causal history $\{t_1, t_2, \mathbf{t}_3\}$—and after that it holds two concurrent versions.

One important observation is that in each node, the union of the causal histories of all versions includes all generated unique names until the last known one: for example, in server S, after both clients send their new versions, all unique names generated in S are known. Thus, the causal past of any update can always be represented using a compact vector representation, as it is the union of all versions known at some server when the client read the object. The combination of the causal past represented as a vector and the last event, kept outside the vector, is known as a *dotted version vector*.[2,13] Figure 8 shows the previous example using this representation, which, as the system keeps running, eventually becomes much more compact than causal histories.

In the condition expressed before (clients communicate only with servers and a new update overwrites all versions previously read), which is common in key-value stores where multiple clients interact with storage nodes via a *get/put* interface, the dotted version vectors allow causality to be tracked between the written versions with vectors of the size of the number of servers.

Final Remarks

Tracking causality should not be ignored. It is important in the design of many distributed algorithms. And not respecting causality can lead to strange behaviors for users, as reported by multiple authors.[1,9]

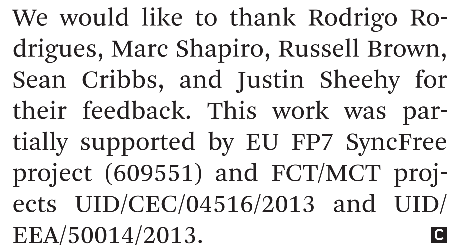
The mechanisms for tracking causality and the rules used in these mechanisms are often seen as complex,[6,15] and their presentation is not always intuitive. The most commonly used mechanisms for tracking causality—vector clocks and version vectors—are simply optimized representations of causal histories, which are easy to understand.

By building on the notion of causal histories, you can begin to see the logic behind these mechanisms, to identify how they differ, and even consider possible optimizations. When confronted with an unfamiliar causality-tracking mechanism, or when trying to design a new system that requires it, readers should ask two simple questions: Which events need tracking? How does the mechanism translate back to a simple causal history?

Without a simple mental image for guidance, errors and misconceptions become more common. Sometimes, all you need is the right language.

Acknowledgments

We would like to thank Rodrigo Rodrigues, Marc Shapiro, Russell Brown, Sean Cribbs, and Justin Sheehy for their feedback. This work was partially supported by EU FP7 SyncFree project (609551) and FCT/MCT projects UID/CEC/04516/2013 and UID/EEA/50014/2013.



Related articles on queue.acm.org

The Inevitability of Reconfigurable Systems
Nick Tredennick, Brion Shimamoto
http://queue.acm.org/detail.cfm?id=957767

Abstraction in Hardware System Design
Rishiyur S. Nikhil
http://queue.acm.org/detail.cfm?id=2020861

Eventually Consistent: Not What You Were Expecting?
Wojciech Golab, et al.
http://queue.acm.org/detail.cfm?id=2582994

References

1. Ajoux, P., Bronson, N., Kumar, S., Lloyd, W., Veeraraghavan, K. Challenges to adopting stronger consistency at scale. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems*, Kartause Ittingen, Switzerland. Usenix Association, 2015.
2. Almeida, P.S., Baquero, C., Gonçalves, R., Preguiça, N.M., Fonte, V. Scalable and accurate causality tracking for eventually consistent stores. In *Proceedings of the Distributed Applications and Interoperable Systems*, held as part of the Ninth International Federated Conference on Distributed Computing Techniques (Berlin, Germany, 2014), 67–81.
3. Birman, K.P., Joseph, T.A. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems 5*, 1 (1987), 47–76.
4. Charron-Bost, B. Concerning the size of logical clocks in distributed systems. *Information Processing Letters 39*, 1 (1991), 11–16.
5. Fidge, C.J. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference 10*, 1 (1988), 56–66.
6. Fink, B. Why vector clocks are easy. Basho Blog, 2010; http://basho.com/posts/ technical/why-vector-clocks-are-easy/.
7. Hoff, T. How League of Legends scaled chat to 70 million players—it takes lots of minions. High Scalability; http://highscalability.com/blog/2014/10/13/how-league-of-legends-scaled-chat-to-70-million-players-it-t.html.
8. Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978), 558–565.
9. Lloyd, W., Freedman, M.J., Kaminsky, M., Andersen, D.G. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (New York, NY, 2011), 401–416.
10. Mattern, F. Virtual time and global states in distributed systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (Gers, France, 1988), 215– 226.
11. Neville-Neil, G. Time is an illusion. *acmqueue 13*, 9 (2015). 57–72
12. Parker, D.S. et al. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering 9*, 3 (1983), 240–247.
13. Preguiça, N.M., Baquero, C., Almeida, P.S., Fonte, V., Gonçalves, R. Brief announcement: Efficient causality tracking in distributed storage systems with dotted version vectors. In *ACM Symposium on Principles of Distributed Computing*. D. Kowalski and A. Panconesi, Eds. (2012), 335–336.
14. Schwarz, R., Mattern, F. Detecting causal relationships in distributed computations: in search of the Holy Grail. *Distributed Computing 7*, 3 (1994), 149–174.
15. Sheehy, J. Why vector clocks are hard. Basho Blog, 2010; http://basho.com/posts/ technical/why-vector-clocks-are-hard/.
16. Sheehy, J. There is no now. *acmqueue 13*, 3 (2015), 20–27.

Carlos Baquero (cbm@di.uminho.pt) is assistant professor of computer science and senior researcher at the High-Assurance Software Laboratory, Universidade do Minho and INESC Tec. His research interests are focused on distributed systems, in particular causality tracking, data types for eventual consistency, and distributed data aggregation.

Nuno Preguiça (nuno.preguica@fct.unl.pt) is associate professor in the Department of Computer Science, Faculty of Science and Technology, Universidade NOVA de Lisboa, and leads the computer systems group at NOVA Laboratory for Computer Science and Informatics. His research interests are focused on the problems of replicated data management and processing of large amounts of information in distributed systems and mobile computing settings.

Copyright held by authors. Publication rights licensed to ACM. $15.00

6.5 Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. Geo-replication: Fast if possible, consistent if necessary. IEEE Data Engineering Bulletin (to appear), 2016.

Geo-Replication: Fast If Possible, Consistent If Necessary*

Valter Balegas[1], Cheng Li[2], Mahsa Najafzadeh[4], Daniel Porto[3], Allen Clement[2,5], Sérgio Duarte[1], Carla Ferreira[1], Johannes Gehrke[6], João Leitão[1], Nuno Preguiça[1], Rodrigo Rodrigues[3], Marc Shapiro[4], Viktor Vafeiadis[2]
[1]NOVA LINCS, DI, FCT, Universidade NOVA de Lisboa [2]Max Planck Institute for Software Systems (MPI-SWS)
[3]INESC-ID / IST, University of Lisbon [4]Inria & Sorbonne Universités, UPMC Univ Paris 06, LIP6
[5]Currently at Google [6]Microsoft

Abstract

Geo-replicated storage systems are at the core of current Internet services. Unfortunately, there exists a fundamental tension between consistency and performance for offering scalable geo-replication. Weakening consistency semantics leads to less coordination and consequently a good user experience, but it may introduce anomalies such as state divergence and invariant violation. In contrast, maintaining stronger consistency precludes anomalies but requires more coordination. This paper discusses two main contributions to address this tension. First, RedBlue Consistency enables blue operations to be fast (and weakly consistent) while the remaining red operations are strongly consistent (and slow). We identify sufficient conditions for determining when operations can be blue or must be red. Second, Explicit Consistency further increases the space of operations that can be fast by restricting the concurrent execution of only the operations that can break application-defined invariants. We further show how to allow operations to complete locally in the common case, by relying on a reservation system that moves coordination off the critical path of operation execution.

1 Introduction

A geo-replicated system maintains copies of the service state across geographically dispersed locations. Geo-replication is not only employed today by virtually all the providers of major Internet services, who typically manage several data centers spread across the globe, but is also accessible to anyone outsourcing their computational needs to cloud providers, since cloud services allow computations or VMs to be instantiated in different data centers.

There are two main reasons for deploying geo-replicated systems. The first reason is disaster tolerance, i.e., the ability to tolerate the unplanned outage of an entire data center, due to catastrophic events such as natural disasters [1]. The second reason is to reduce the latency between the users and the machines that provide the service. The importance of this aspect is demonstrated by several studies that point out an inverse correlation between response times and user satisfaction for important Internet services such as search [30].

Copyright 0000 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.
Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*Student authors are followed by faculty author names, both in alphabetical order. Cheng Li and Valter Balegas are the lead authors of the work.

However, there is a fundamental tension between latency and consistency: intuitively, ensuring strong consistency requires coordination between replicas before returning a reply to the user, while, alternatively, a fast response can be given without replica coordination, but only ensuring weak consistency guarantees. This tension has led the providers of global-scale Internet services to choose, for some parts of their services, storage systems offering weak consistency guarantees such as eventual consistency [13], and, for other components, systems with strong consistency such as serializability [10].

This paper revisits in a unified way two of our recent results in trying to achieve a balance between performance and consistency, by devising methods to build geo-replicated systems that introduce a small amount of coordination between replicas to achieve the desired semantics, i.e., systems that are fast when possible and consistent when necessary [21, 7]. In the first result, we improve the performance of geo-replicated systems by (1) allowing different operations to execute in either a weakly consistent (fast) or strongly consistent (slow) manner; and (2) identifying a set of principles for making safe use and increasing the space of fast operations. In the second result, we further increase the space of operations that can execute fast by (1) identifying the operations that can break application invariants when executing concurrently; and (2) deploying concurrency control mechanisms that remove coordination from the critical path of operation execution, while preserving invariants.

We start the presentation by laying out our terminology and system model in Section 2. We present an initial approach based on a coarse-grained classification into strong and weak consistency in Section 3. One key aspect of this approach is operation commutativity, and we explain how to achieve it using CRDTs in Section 4. Then we present an approach that makes use of fine-grained coordination between pairs of operations in Section 5. We discuss related work in section 6 and conclude in Section 7.

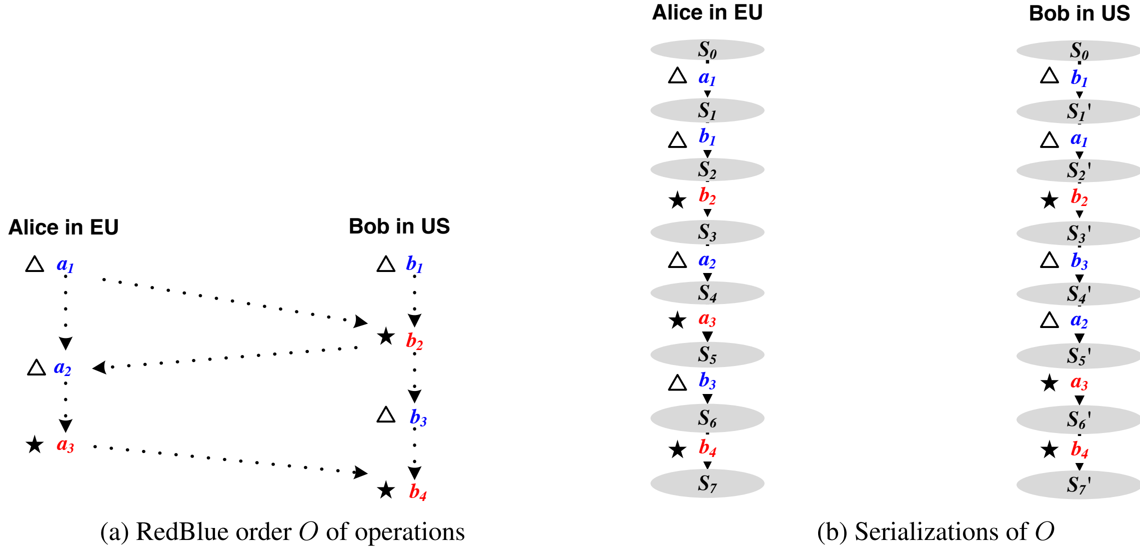
2 System model

Our system model is that of a fully replicated distributed system, where replicas are located in different data centers. Each replica follows a deterministic state machine: there is a set of operations $\mathcal{U}$, which manipulate a set of reachable system states $\mathcal{S}$. Each operation u is initially submitted at a given replica (preferably in the closest data center), which we call the *origin replica* of u. When the remaining replicas receive a request to replicate this operation, they will apply the operation against their local state.

Throughout our explanation we will highlight two important properties that the replicated system should obey. First, there is the **state convergence** property, which says that all the sites that have executed the same set of operations against the same initial state are in the same final state. This is important to prevent a situation where the system quiesces (no more updates are received) and read-only queries return different results depending on which sites the users are connected to. The second property is to **preserve application-specific invariants**, which comprise a specification for the behavior of the system. To define these, we introduce the primitive *valid*(S) to be *true* if state S obeys these invariants and *false* otherwise.

3 Mixing consistency levels in RedBlue consistency

In this section, we present a hybrid consistency model called RedBlue consistency, where weakly consistent operations, labeled blue, can be executed at a single replica and propagated in the background, with mostly no coordination with concurrent actions at other replicas, while others, labeled red, require a stronger consistency level and thus require cross-replica coordination. RedBlue consistency is one of several systems that propose labeling operations according to their consistency levels [18, 33, 21, 36], but improves on these systems by offering a precise method for labeling operations.



(a) RedBlue order O of operations (b) Serializations of O

Figure 1: RedBlue order and serializations for a system spanning two sites. Operations marked with $\star$ are red, and operations marked with $\triangle$ are blue. Dotted arrows in a indicate the partial ordering of operations.

3.1 Defining RedBlue consistency

RedBlue consistency relies on three components: (1) a partitioning of operations into weakly consistent blue operations whose order of execution can vary from site to site, and red operations that must be executed in the same order at all sites, (2) a RedBlue order, which defines a partial order of operations where red operations have to be ordered with respect to each other, and (3) a set of site-specific serializations (i.e., total orders) in which the operations are locally applied. More precisely:

Definition 1 (RedBlue consistency): A replicated system is *RedBlue consistent* if each site i applies operations according to a linear extension of a RedBlue order O of the operations that were invoked, where O is a partial order among those operations with the requirement that red operations are totally ordered in O.

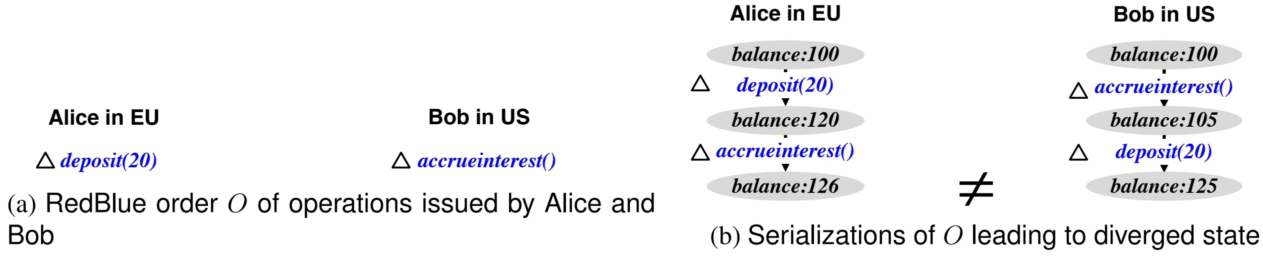
Figure 1 shows a RedBlue order and two serializations, i.e., the linear extensions of that order in which operations are applied at two different sites. In systems where every operation is labeled red, RedBlue consistency is equivalent to serializability [10]; in systems where every operation is labeled blue, RedBlue consistency becomes a form of causal consistency [37, 23, 25], since the partial order conveys the necessary causality between operations.

When applying RedBlue consistency to an application, we would like to label all operations blue to obtain best performance. However, this could lead to state divergence and invariant violation, when operations are not commutative. We describe a set of sufficient conditions to guide the classification of operations in order to safely use weak consistency when possible.

3.2 Ensuring state convergence

In the context of RedBlue consistency, we can formalize state convergence as follows:

Definition 2 (State convergence): A RedBlue consistent system is state convergent if all serializations of the underlying RedBlue order O reach the same state S w.r.t. any initial state S_0.



(a) RedBlue order O of operations issued by Alice and Bob

(b) Serializations of O leading to diverged state

Figure 2: A RedBlue consistent account with initial balance of 100 and final diverged state.

To find a correct labeling for maintaining state convergence while providing low latency access, we describe a simple banking example, in which users may share an account that is modified via three operations, namely `deposit`, `withdraw` and `accrueinterest`[1]. Keeping in mind that one of the goals of RedBlue consistency is to make the target service as fast as possible, we tentatively label all these operations blue. According to this labeling result, we construct a RedBlue order of deposits and interest accruals made by two users Alice and Bob and two possible serializations applied at both branches of the bank, as shown in Figure 2. This example shows that the labeling of these operations as described is not state convergent. This is because RedBlue consistency allows the two sites to execute blue operations in a different order, but two of the blue operations in the example are non-commutative, namely `deposit` and `accrueinterest`. To prevent this situation, a sufficient condition to guarantee state convergence in a system supporting RedBlue consistency is that every blue operation commutes with all other operations, blue or red.

However, when applying this condition to the banking example, it implies that we need to label all three operations red (`deposit`, `withdraw` and `accrueinterest`). This is equivalent to running the system under serializability, which requires coordination across replicas for executing all these operations. To address the problem that it is difficult to find operations that commute with all other operations in the system, we observe that, in many cases, while operations may not be commutative, we can make the changes they induce on the system state to commute. In the banking example, we can engineer `accrueinterest` commute with the remaining two operations by first computing the amount of interested accrued at the primary replica and then treating that value as a deposit.

To exploit this observation and increase operation commutativity, we propose a change to our original system model, where we split each original application operation u into two components: a *generator operation* g_u with no side-effects, which is executed only at the primary site against some system state S and produces a *shadow operation* $h_u(S)$, which is executed at every site (including the primary site). The generator operation decides which state transitions should be made while the shadow operation applies the transitions in a state-independent manner.

3.3 Preserving invariants

Although the concept of shadow operation helps produce more commutative operations, labeling too many shadow operations as blue may introduce the problem of breaking application invariants. In the banking example, assuming that the shared bank account has an initial balance of 100, if both Alice and Bob withdraw 70 and 60 respectively, the final balance would be -30. This violates the invariant that a bank balance should never be negative. To determine which shadow operations can be safely labeled blue, we begin by defining that a shadow operation is invariant safe if, when applied to a valid state, it always transitions the system into another valid state. This allows us to define the following sufficient condition: a RedBlue consistent system preserves

[1] `accrueinterest` computes a new balance by multiplying the old balance value and (1 + `interest rate`).

invariants (meaning that all its sites are always in valid states) if all shadow operations that are not invariant safe are labeled red (i.e., strongly consistent).

3.4 What can be blue? What must be red?

In summary, the two conditions above lead to the following procedure for deciding which shadow operations can be blue or must be red if a RedBlue consistent system is to provide both state convergence and invariant preservation:

1. For any pair of non-commutative shadow operations h_u and h_v, label both h_u and h_v red.
2. For any shadow operation h_u that is not invariant safe, label h_u red.
3. Label all remaining shadow operations blue.

4 State convergence

In the previous section we discussed how RedBlue consistency achieves state convergence by relying on shadow operations that commute with each other. With this approach, defining a new operation also implies writing one or more commutative shadow operations, each of which corresponds to a distinct side effect. The major challenge of doing this manual work is that, in an application with a large number of operations, this process may be complex and error-prone.

We now discuss an alternative principled approach to create commutative operations by design. Our approach builds on conflict-free replicated data types (CRDTs) [31], which are specially-designed data structures that can be replicated and modified concurrently, and include mechanisms to merge concurrent updates in a deterministic way. Application operations consist of updates to these elementary data types, thus guaranteeing state convergence.

4.1 CRDTs

A CRDT is a data type that can be replicated at multiple replicas. As such, it defines an interface with a set of operations to read and to modify its state. A CRDT replica can be modified by locally executing an update operation. When different replicas of the same object are modified concurrently, they temporarily diverge. CRDTs have built-in support for achieving strong eventual consistency [31], in which all replicas will eventually reach the same (equivalent) state after applying the same set of updates, without relying on a distributed conflict arbitration process.

Two main flavors of CRDTs have been studied in the literature: operation-based CRDTs and state-based CRDTs. For each of these, sufficient conditions for achieving strong eventual consistency have been established.

In operation-based CRDTs (or commutative replicated data types), updates are propagated by broadcasting operations to every replica in causal order. Interestingly, this proposal matches the operation execution decomposition presented in RedBlue consistency (Section 3), where operations are divided in two components, a *generator* operation that executes in the local replica, has no side effect and produces a *shadow* operation, which is propagated and executed in all replicas. The two types of operations are analogous to *prepare* and *downstream* operations in the context of operation-based CRDTs, respectively, with the main difference that shadow operations are assigned a consistency level in RedBlue consistency. Similarly to the consequence of commutative shadow operations, the replicas of an operation-based CRDT converge to the same state after executing the same set of updates (in any order that respects causality) if the execution of any two concurrent downstream operations commutes [31].

SQL type	CRDT	Description
FIELD*	LWW	Use last-writer-wins to solve concurrent updates
	NUMDELTA	Add a delta to the numeric value
TABLE	AOSET, UOSET, AUSET, ARSET	Sets with restricted operations (add, update, and/or remove). Conflicting operations are logically executed by timestamp order.

Table 2: Commutative replicated data types (CRDTs) for relational data. * FIELD covers primitive types such as integer, float, double, datetime and string.

A state-based CRDT (or convergent replicated data type) defines, in addition to the operations to read and update its state, an operation to merge the state of two replicas. Replicas synchronize by exchanging the full replica states: when a new state is received, the new updates are incorporated in the local replica by executing the merge function. It has been shown that the replicas of a state-based CRDT converge to the same state after all replicas synchronize (directly or indirectly) if: (1) all the possible states of an object are partially ordered, forming a join-semilattice; (2) the merge operation between two states is the semilattice join; and (3) an update monotonically increases the state according to the defined partial order [31].

4.2 Examples

CRDTs have been used in a number of research systems, such as Walter [35] and SwiftCloud [39], and commercial systems, such as Riak [2] and SoundCloud [3]. These systems include CRDTs that implement several data types, such as registers, counters, sets, maps, and flags. For each such data type, it is possible to define and implement different semantics to handle concurrent updates, leading to different CRDTs. These semantics define which is the final state of a CRDT when concurrent updates occur. For example, for sets, it is possible to define an *add-wins* semantics, where, in the presence of a concurrent add and remove of some element e, the final state will contain e (or, more precisely, there exists an add of e that does not happen before a remove of e). It is also possible to define a *remove-wins* semantics, where the remove will win over a concurrent add. Other semantics can also be implemented, such as a *last-writer-wins* strategy where an element will belong to the set or not depending on which was the last operation executed, according to the order among operations.

When creating an application, an application developer must select the CRDT with the most appropriate semantics for its goal. For example, in the bank account example, the balance of an account can be modeled as a counter and the set of accounts of a client can be maintained in an add-win set or map CRDT.

In general, an application operation will manipulate multiple data objects. When using CRDTs, it is possible to maintain replicas of these objects in multiple nodes. An operation can execute by accessing a single replica of each object it accesses. These updates can later be propagated to other nodes, with CRDT rules guaranteeing that the replicas of each object will converge to the same state. By propagating the updates to all objects modified in an operation atomically, it is possible to guarantee that all effects of an operation are observed at the same time.

CRDTs for relational databases In relational databases, it is also possible to model data using CRDTs. Table 2 presents the mapping proposed in SIEVE [20]. Regarding table fields, we defined only two CRDTs. The *LWW* CRDT can be used with any field type and implements a *last-writer-wins* strategy for defining the final value of a field. The *NUMDELTA* CRDT can be used with numeric fields, and transforms each update operation in a downstream operation that adds or subtracts a constant to the value of the field. This can be used to support account balances, counters, etc.

A database table can be seen as a set of tuples. In the general case, and following the semantics of the *ARSET* CRDT, when concurrent *insert*, *update* and *delete* operations occur, the following rules can be used: (1) concurrent inserts of tuples with the same key are treated as an insert followed by a sequence of updates;

(2) for concurrent updates, the rules defined for fields are used to deterministically define the final value; (3) a delete will only take effect if no concurrent update or insert was executed.

While using CRDTs guarantees that all replicas converge to the same state, it does not guarantee that the convergence rules executed independently by different CRDTs maintain application invariants. Next, we show how we can address this problem by restricting the concurrent execution of operations that can break application invariants.

5 Preserving invariants with minimal coordination

As mentioned before, in the banking example, the `withdraw` operation, despite being commutative, cannot execute under weak consistency, as the concurrent execution of multiple withdrawals can break the invariant that the account balance cannot be negative. To avoid the possibility of breaking the invariant, RedBlue consistency would label all withdrawals as red, requiring replicas to coordinate the execution of every withdraw operation. In practice, however, only in a few cases the cumulative effects of all concurrent withdrawals will surpass the actual balance of the account.

To relieve the strong constraint imposed by RedBlue consistency, we propose a more efficient coordination plan: given some account balance, replicas can coordinate beforehand to split the balance among them. Until a replica consumes its allocated share of the balance, it can execute operations locally, without coordination with other replicas, with the guarantee that the balance will not become negative, i.e., the application invariant will not be broken.

The above idea has been previously explored in the context of escrow transactions [9, 27]. We revisit and generalize the concept of escrow transactions, to allow replicas to assess the safety of operations without coordination when executing operations. In our generalization, when replicas cannot ensure an operation is safe by reading local state, they contact remote peers to update their vision of the database to decide the fate of the operation. In addition, we discuss how we avoid the coordination across sites for all red operations, which is required for totally ordering them. Instead, we identify a small set of coordination requirements between operations, and show how to enforce those rules at runtime.

5.1 Explicit Consistency in a nutshell

We present a new consistency model, called Explicit Consistency, that extends RedBlue consistency to avoid the coordination of red operations when possible. The idea is that instead of labeling shadow operations as red or blue, programmers specify the application invariants. The system must execute operations while guaranteeing that these invariants are not broken.

To this end, we propose the following methodology for creating applications that adhere to Explicit Consistency. First, programmers must specify the application invariants and operation effects. Second, we provide a tool to analyze the specification of the application and identify the pairs of conflicting shadow operations. Non-conflicting shadow operations execute without any restrictions, as blue operations. We include a library of CRDTs to help programmers define commutative operations. Third, for each pair of conflicting shadow operations, the programmer can use a specialized concurrency control mechanism that restricts the concurrent execution of these operations. This mechanism executes coordination outside of the critical path of operation execution, allowing these operations to execute locally without the need to coordinate with other replicas.

The following sections provide additional details on these steps to use the Explicit Consistency model.

5.2 Application specification

Programmers specify application invariants and the post-conditions of shadow operations as first order logic expressions. Invariants must be written as universally quantified formulas in prenex normal form, while the

```
01: //Invariant declaration
02: @Invariant( "forall(aId : Aid) :- balance(aId) >= 0" )
03: @Invariant( "forall(cId : Cid, aId : Aid) :- userAccount(cId, aId) =>
04:                                         registeredUser(cId)")
05: public interface Bank{
06:
07:   @decrement(balance(accountId), amount)
08:   boolean withdraw( CId clientId, Aid accountId, Int amount);
09:
10:   @true(userAccount(clientId, accountId))
11:   boolean assignAccount( CId clientId, AId accountId);
12:
13:   @false(userAccount(clientId, accountId))
14:   boolean closeAccount( CId clientId, AId accountId);
16:
17:   @false(registeredUser(clientId))
18:   boolean endContract( CId clientId);
19:
20: }
```

(a) Specification written with Java Annotations.

Operations X *Operations*		*Conflict*
withdraw(cId, aId, amount)	withdraw(cId, aId, amount)	**Non-idempotent**
…		
assignAccount(cId,aId)	closeAccount(cId, aId)	**Opposing**
…		
assignAccount(cId,aId)	endContract(cId)	**Conflict**
…		

(b) Conflicting pairs of operations for the Bank example.

Figure 3: Bank application specification and analysis results.

grammar for specifying applications post-conditions is restricted to predicate assignments, that assert the truth value of some predicate, and function clauses, which define the relation between the value of some predicate before and after the execution of the operation.

The code snippet in figure 3a shows the specification of the banking application. We extended this example to illustrate different invariant violations. In the extended version, clients must have a valid contract with the bank to be able to access an account. Clients might have multiple accounts and must close all of them before finishing the contract. In Line 2, the invariant guarantees that an account balance is never negative. In line 3, the invariant states that, for every open account, the account holder must be registered with the bank.

5.3 Analysis

The analysis checks which are the shadow operations whose concurrent execution might produce a database state that is invalid with respect to the declared invariants. Conceptually, for each pair of operations and for every valid state where these operation can execute, the algorithm verifies if the execution of both operations will lead to a state that is not valid according to the invariants of the application. Obviously, checking every pair of operations in every valid state exhaustively is unfeasible. Instead, our algorithm relies in the Z3 satisfiability modulo theory (SMT) solver to perform this verification efficiently. A full description of the algorithm is given in our prior publication [7].

Figure 3b summarizes the conflicts in the example of Figure 3a: two concurrent successful withdrawals might make the balance negative (non-idempotence); assigning and removing an account concurrently for the same user might leave the system in an inconsistent state, because each shadow operation writes different values for the predicate $userAccount(cId, aId)$ (opposing post-conditions); and finally, the pair $createAccount(cId, aId)$ and $endContract(cId)$ might violate the integrity constraint of line 3, because a new account is being added to a user that is ending a contract with the bank.

5.4 Code instrumentation

After identifying which operations can lead to conflicts, the programmer must instrument the application to avoid them.

Some conflicts can be handled by simply relying on CRDTs to automatically solve them. For example, our analysis can report that operations have opposing post-conditions: e.g., operations `assignAccout` and

`remAccount` assign the value *true* and *false* to predicate $userAccount(cId, aId)$. In this situation, the programmer can choose a preferred value for the predicate and use a CRDT that automatically implements the selected decision[2].

Other conflicts must be handled by restricting the concurrent execution of operations that can cause invariants to be broken. To this end, we provide a set of specialized reservation-based concurrency control mechanisms.

For conflicts on numeric invariants, like the one that withdraw causes, we support an escrow reservation for allowing some decrements of numeric values to execute without coordination. In an escrow reservation, each replica is assigned a budget of decrements, based on the initial value of the data. In our example, when a replica receives a withdraw request, if the local budget is sufficient, the generator operation executes immediately without coordination, generating a shadow operation that decrements the balance. This local execution is safe, guaranteeing that the invariant still holds after executing all concurrent operations, because the sum of the budgets of all replicas is equal to the value of the initial value. If the local budget is not enough to satisfy the request, the replica needs to contact remote replicas to increase its budget, until it can satisfy the request. If that is not possible, because there are not enough resources globally, then the generator operation fails, generating no shadow operation.

For conflicts on generic invariants, we include a multi-value lock reservation. This lock can be in one of the following three states: (1) shared forbid, giving the shared right to forbid some action to occur; (2) shared allow, giving the shared right to allow some action to occur; (3) exclusive allow, giving the exclusive right to execute some action. The idea is that, for a conflicting pair of operations, (o_1, o_2), the lock will be associated with the execution of one of the operations, say o_1. To execute o_1, a replica must hold the lock in the shared allow mode. This right can be shared by multiple replicas. To execute o_2, a replica must hold the lock in the shared forbid mode. As before, when executing the generator operation, if the replica already holds the necessary locks (in the required mode to execute the operation), it can execute locally and generate the corresponding shadow operation. If not, it must contact other replicas to obtain the necessary locks.

Besides these two locks, we also proposed other locks that can efficiently restrict the concurrent execution of operations that conflict in other types of invariants, including conditions on the number of elements that satisfy a given condition and disjunctions. In a related work, Gotsman et. al. [16] have shown how to prove that a given set of locks is sufficient for maintaining invariants.

6 Related work

Many cloud storage systems supporting geo-replication have emerged in recent years. Some of these systems offer variants of eventual consistency, where operations produce responses right after being executed in a single data center (usually the closest one) and are replicated in the background, so that user observed latency is improved [13, 23, 24, 4, 19]. These variants target different requirements, such as: reading a causally consistent view of the database (causal consistency) [23, 4, 14, 6]; supporting limited transactions where a set of updates are made visible atomically [24, 5]; supporting application-specific or type-specific reconciliation with no lost updates [13, 23, 35, 2], etc.

While some systems implement eventual consistency by relying on a simple last-writer-wins strategy, others have explored the semantics of applications (and data types). Semantic types [15] have been used for building non-serializable schedules that preserve consistency in distributed databases. Conflict-free replicated data types [31] explore commutativity for enabling the automatic merge of concurrent updates to the same data types.

Eventual consistency is insufficient for some applications that require some operations to execute under strong consistency for correctness. To this end, several systems support strong consistency. Spanner provides strong consistency for the whole database, at the cost of incurring coordination overhead for all updates [12].

[2]In our experience, boolean predicates can be implemented using Set CRDTs with add-wins and remove-wins policies to enforce that the corresponding predicate becomes true or false respectively.

Transaction chains support transaction serializability with latency proportional to the latency to the first replica that the corresponding transaction accesses [40]. MDCC [17] and Replicated Commit [26] propose optimized approaches for executing transactions but still incur inter-data center latency for committing transactions.

Some systems combine the benefits of weak and strong consistency models by allowing both levels to coexist. In Walter [35], transactions that can execute under weak consistency run fast, without needing to coordinate with other datacenters. Bayou [37] and Pileus [36] allow operations to read data with different consistency levels, from strong to eventual consistency. PNUTS [11] and DynamoDB [34] also combine weak consistency with per-object strong consistency relying on conditional writes, where a write fails in the presence of concurrent writes. RedBlue consistency also combines weak and strong consistency in the same system. Unlike other systems, RedBlue consistency splits operations into generator and shadow parts to allow more operations to commute, and define a procedure to help programmers labeling shadow operations as weak or strong.

Escrow transactions [27] offer a mechanism for enforcing numeric invariants under concurrent execution of transactions. By enforcing local invariants in each transaction, they can guarantee that a global invariant is not broken. This idea can be applied to other data types, and it has been explored for supporting disconnected operation in mobile computing [38, 28, 32]. Balegas et al. [8] proposed the bounded counter CRDT that can be used to enforce numeric invariants in weakly consistent cloud databases. The demarcation protocol [9] aims at maintaining invariants in distributed databases. Although its underlying protocols are similar to escrow-based approaches, it focuses on maintaining invariants across different objects. Warranties [22] provide time-limited assertions over the database state, which can improve latency of read operations in cloud storages. Indigo builds on similar ideas for enforcing application invariants, but it is the first piece of work to provide an approach that, starting from application invariants expressed in first-order logic, leads to the deployment of the appropriate techniques for enforcing such invariants in a geo-replicated weakly consistent data store. Gotsman et. al. [16] propose a proof rule for establishing that the use of a given set of techniques is sufficient to ensure the preservation of invariants.

The static analysis of code is a standard technique used extensively for various purposes, including in a context similar to ours. SIEVE [20] combines static and dynamic analysis to infer which operations should use strong consistency and which operations should use weak consistency in a RedBlue system [21]. Roy et al. [29] present an analysis algorithm that describes the semantics of transactions. These works are complementary to ours, since the proposed techniques could be used to automatically infer application side effects.

7 Conclusion

In this paper we summarized two of our recent results in addressing the fundamental tension between latency and consistency in geo-replicated systems. First, RedBlue consistency [21] offers fast geo-replication by presenting sufficient conditions that allow programmers to safely separate weakly consistent (fast) operations from strongly consistent (slow) ones in a coarse-grained manner. To increase the space of potential fast operations and simplify the programmer's task of defining commutative operations, we propose the use of conflict-free replicated data types. Second, Explicit Consistency [7] enables programmers to make fine-grained decisions on consistency level assignments by connecting application invariants to ordering conflicts between pairs of operations, and explores efficient reservation techniques for coordinating conflicting operations with low cost.

Acknowledgments

The research of Rodrigo Rodrigues is supported by the European Research Council under an ERC Starting Grant. This research was also supported in part by EU FP7 SyncFree project (609551), FCT/MCT SFRH/BD/87540/2012, PEst-OE/ EEI/ UI0527/ 2014, NOVA LINCS (UID/CEC/04516/2013), and INESC-ID (UID/CEC/50021/2013).

References

[1] 7 Data Center Disasters You'll Never See Coming. http://www.informationweek.com/cloud/7-data-center-disasters-youll-never-see-coming/d/d-id/1320702. Accessed Feb-2016.

[2] Using data types – riak documentation. http://docs.basho.com/riak/latest/dev/using/data-types/. Accessed Feb-2016.

[3] Consistency without Consensus: CRDTs in Production at SoundCloud. http://www.slideshare.net/InfoQ/consistency-without-consensus-crdts-in-production-at-soundcloud Accessed Feb-2016.

[4] S. Almeida, J. Leitão, and L. Rodrigues. ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication. In *EuroSys '13*, 85–98, 2013. ACM.

[5] P. Bailis, A. Fekete, J. M. Hellerstein, A. Ghodsi, and I. Stoica. Scalable Atomic Visibility with RAMP Transactions. In *SIGMOD '14*, 27–38, 2014. ACM.

[6] P. Bailis, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Bolt-on Causal Consistency. In *SIGMOD '13*, 761–772, 2013. ACM.

[7] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys '15*, 6:1–6:16, 2015. ACM.

[8] V. Balegas, D. Serra, S. Duarte, C. Ferreira, M. Shapiro, R. Rodrigues, and N. Preguica. Extending eventually consistent cloud databases for enforcing numeric invariants. In *SRDS '15*, 31–36, Sept 2015.

[9] D. Barbará-Millá and H. Garcia-Molina. The demarcation protocol: A technique for maintaining constraints in distributed database systems. *The VLDB Journal*, 3(3):325–353, July 1994.

[10] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[11] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, Aug. 2008.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's Globally-distributed Database. In *OSDI '12*, 251–264, 2012. USENIX Association.

[13] 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 *SOSP '07*, 205–220, 2007. ACM.

[14] J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel. Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks. In *SOCC '13*, 11:1–11:14, 2013. ACM.

[15] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[16] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro. 'cause i'm strong enough: Reasoning about consistency choices in distributed systems. In *POPL 2016*, 371–384, 2016. ACM.

[17] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data Center Consistency. In *EuroSys '13*, 113–126, 2013. ACM.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Trans. Comput. Syst.*, 10(4):360–391, Nov. 1992.

[19] A. Lakshman and P. Malik. Cassandra: A Decentralized Structured Storage System. In *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[20] C. Li, J. Leitão, A. Clement, N. Preguiça, R. Rodrigues, and V. Vafeiadis. Automating the choice of consistency levels in replicated systems. In *ATC '14*, 281–292, 2014. USENIX Association.

[21] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues. Making Geo-replicated Systems Fast As Possible, Consistent when Necessary. In *OSDI '12*, 265–278, 2012. USENIX Association.

[22] J. Liu, T. Magrino, O. Arden, M. D. George, and A. C. Myers. Warranties for faster strong consistency. In *NSDI '14*, 2014. USENIX Association.

[23] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-area Storage with COPS. In *SOSP '11* , 401–416, 2011. ACM.

[24] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen. Stronger Semantics for Low-latency Geo-replicated Storage. In *NSDI '13*, 313–328, 2013. USENIX Association.

[25] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: cloud storage with minimal trust. In *OSDI*, 2010.

[26] H. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency Multi-datacenter Databases Using Replicated Commit. *Proc. VLDB Endow.*, 6(9):661–672, 2013.

[27] P. E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, Dec. 1986.

[28] N. Preguiça, J. L. Martins, M. Cunha, and H. Domingos. Reservations for Conflict Avoidance in a Mobile Database System. In *MobiSys '03*, 43–56, 2003. ACM.

[29] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD '15*, 1311–1326, 2015.

[30] E. Schurman and J. Brutlag. Performance related changes and their user impact. Presented at velocity web performance and operations conference. `http://slideplayer.com/slide/1402419/`, 2009.

[31] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *SSS '11*, 386–400, 2011. Springer-Verlag.

[32] L. Shrira, H. Tian, and D. Terry. Exo-leasing: Escrow Synchronization for Mobile Clients of Commodity Storage Servers. In *Middleware '08*, 42–61, 2008. Springer-Verlag New York, Inc.

[33] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, and P. Maniatis. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *NSDI'09*, 169–184, 2009.

[34] S. Sivasubramanian. Amazon DynamoDB: A Seamlessly Scalable Non-relational Database Service. In *SIGMOD '12*, 729–730, 2012. ACM.

[35] Y. Sovran, R. Power, M. K. Aguilera, and J. Li. Transactional Storage for Geo-replicated Systems. In *SOSP '11*, 385–400, 2011. ACM.

[36] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh. Consistency-based Service Level Agreements for Cloud Storage. In *SOSP '13*, 309–324, 2013. ACM.

[37] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP '95*, 172–182, 1995. ACM.

[38] G. D. Walborn and P. K. Chrysanthis. Supporting Semantics-based Transaction Processing in Mobile Database Applications. In *SRDS '95*, 31–40, 1995. IEEE Computer Society.

[39] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write fast, read in the past: Causal consistency for client-side applications. In *Middleware '15*, 75–87, 2015. ACM.

[40] Y. Zhang, R. Power, S. Zhou, Y. Sovran, M. K. Aguilera, and J. Li. Transaction Chains: Achieving Serializability with Low Latency in Geo-distributed Storage Systems. In *SOSP '13*, 276–291, 2013. ACM.

6.6 Valter Balegas, Sérgio Duarte, Carla Ferreira, Nuno Preguiça, and Rodrigo Rodrigues. Making Weak Consistency Great Again. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016.

Making Weak Consistency Great Again

Valter Balegas, Sérgio Duarte
Carla Ferreira, Nuno Preguiça
NOVA LINCS / FCT, Universidade NOVA de Lisboa

Rodrigo Rodrigues
INESC-ID / IST, University of Lisbon

ABSTRACT

This talk focus on the problem of implementing web applications on top of weakly consistent geo-replicated system. Several techniques, such as CRDTs, have been proposed to achieve state convergence on a per-data type basis. However, this is not sufficient for guaranteeing application correctness, as convergence rules executed individually at each object may lead to an invalid state.

We advocate that it is possible to address these problems and implement correct applications under weak consistency. To this end, it is necessary to combine CRDTs with novel semantics, a judicious selection of CRDTs used by applications and transform application operation to guarantee that convergence rules applied on a per-object basis always lead to a valid application state. individually Achieving this is complex, requiring tools that can help programmers taming the complexity of programming on top of weak consistency and make the technology more accessible.

We conclude our talk with the demonstration of a tool that we are developing that is capable of detecting concurrency conflicts on applications and proposes transformations to make them conflict-free.

1. INTRODUCTION

The pervasiveness of the internet in people's day-to-day activities has lead to a paradigm shift in the way developers build web applications. Nowadays systems need to scale to unprecedented levels and provide good quality of service at a world-wide scale. Centralized systems are inherently hard to scale, prone to failure and provide poor quality of service, for users that access from remote locations. This has lead developers to become more interested in developing distributed systems that do not suffer from the limitations of the former. In this context, geo-replication appears as a core technique to implement scalable and responsive distributed systems for clients scattered across the globe.

Systems that use geo-replication typically provide weaker forms of consistency in order to allow replicas to process requests without contacting remote replicas. As a consequence, when the same objects are updated at different locations, their values diverge and need to be reconciled later. Conflict-free Replicated Data types (CRDTs) [7] are a principled approach to handle replica convergence. These data types support the reconciliation of multiple divergent copies of the same object, with a well-defined semantics. Various systems use CRDTs to provide richer programming models for developers [5, 8].

Despite the effort to make weak consistency systems easier to program, it remains difficult. In systems that use strong consistency, application correctness is always ensured as long as the application's individual operations respect the invariants. That is not the case for applications implemented on top of weak consistency, where developers must carefully ensure that application invariants hold under concurrent execution. The root of this problem is that the reconciliation of concurrent updates might produce a state that is invalid with respect to the application invariants. Bailis et al. have studied applications available online and found that many applications built on top of weak consistency do not provide the expected semantics [2].

In our recent work with bounded counter [4], we have designed a counter that can maintain numeric invariants under concurrent update execution. The same strategy can be used with other data types to provide other invariants. However a more general problem is yet to solve: how to maintain invariants across multiple objects, without loss of availability? An example of such invariants is to ensure referential integrity in relational databases. Previous works have addressed this issue by constraining the operations that can be executed in each replica in order to preserve data integrity [3, 6]. Whenever a replica receives a request to execute an operation that might violate an invariant, the replica must coordinate with remote replicas to ensure that the operation is safe. Although this coordination may sometime be execute outside of the critical execution path of operations, a replica can always be forbidden to execute some operation because it needs to contact a remote replica that is unavailable.

In this talk, we study the example of referential integrity and show how to transform an application to provide that invariant on top of weak consistency. We discuss different semantics to solve the conflict without constraining concurrency, showing that it is possible to implement scalable and correct applications on top of weak consistency. We also present the current status of a tool we are developing to help developers in that process.

2. RUNNING EXAMPLE

We chose referential integrity as running example due to its importance in relational databases and concurrent programming in general. We consider a toy database composed of two entities, A and B. We assume, without loss of generality, that each entity has a single attribute. There is a one-to-many relationship $R_{a \to b}$ from elements of A to elements of B.

Consider the implementation of this example using an object-relational mapping approach, where entities are mod-

eled as two distinct sets and the relationship between them are modeled by a third set of pairs $(a, b) : a \in A, b \in B$.

We assume that the storage system stores each set in separate objects and that it provides causal consistency and atomic updates across multiple objects.

The integrity constraint of this model is broken when $\exists(a, b) \in R_{a \to b} : a \notin A \vee b \notin B$, i.e. there is a relationship between entity a and b, but one or both of them do not exist. We consider, for simplicity, that the application is correct under strong consistency, i.e. any sequential execution of the program does not violate the invariant. An invariant violation only occurs when a client issues an operation to create a new relation (a, b) while another client issues an operation to remove a or b from A or B, respectively.

3. BETTER SAFE THAN SORRY

To allow fast execution without constraining concurrency, every replica must be able to reply to a request without depending on remote state. Under these circumstances it is not possible to avoid concurrent executions that might leave the database in an inconsistent state. Since detecting conflicts and fixing invalid database state is expensive, we propose solving conflicts beforehand, so that execution is always safe. The idea is that an operation can have extra effects in order to avoid generating an invalid state when replicas are reconciled. As a trade-off, the semantics of operations that are implemented this way is limited, but, as we show next, interesting semantics can be provided with proper use of convergence rules.

In the next section we describe two alternative solutions for the described problem. In the first solution we rely exclusively on existing CRDT semantics, while in the second solution we devise a new convergence rule for concurrent operations to implement an alternative semantics.

3.1 Adding missing elements

When a new element (a, b) is added to $R_{a \to b}$, the operation that adds this element to the relations set must ensure that $a \in A$ and $b \in B$ to preserve referential integrity. These elements might be removed concurrently at other replicas leading to an invariant violation after replicas reconcile. To avoid this conflict, we modify the operation that adds (a, b) to $R_{a \to b}$ to also add a to A and b to B, atomically, and set the convergence rule of each set to use a Add-Wins policy. This policy ensures that if an add and remove operations execute concurrently for the same element, then the element will be present in the set, cancelling the effects of the remove operation. The consequence of our modifications is that any operation to remove elements a or b will be cancelled by the additional effect of the operation that adds (a, b), if they execute concurrently.

3.2 Ensuring that elements are removed

In the previous solution, whenever the conflicting operations execute, the operation that adds the element to $R_{a \to b}$ takes precedence over the remove operations. We might want the opposite semantics, i.e. that whenever a remove operation for a or b is issued, we want to cancel any concurrent operation that adds and element to $R_{a \to b}$ containing one of those values. This example is different from the previous and cannot be solved in the same way, because we do not know the possible pairs containing a or b that might be added to the set, and it would be too expensive to consider the whole domain of A or B. In this talk we present a new set CRDT that prevents concurrently adding elements to a set that match a given criteria, without specifying their values.

The intuition behind this new set is to provide a special *touch(Predicate p)* operation that accepts a predicate that specifies which elements we want to prevent adding concurrently to the set. This way, whenever we execute a remove operation for elements a or b, we also execute a *touch* in $R_{a \to b}$ that prevent the addiction of any pair matching $(a, *)$ or $(*, b)$, where $*$ means any element.

4. TOOLS FOR PROGRAMMING WEAK CONSISTENCY

In the previous section we have seen how to preserve referential integrity in applications developed on top of weak consistency. Even though the transformations to the operations are easy to explain, it might be difficult for the average programmers to devise them. For this reason, we are also working on tools that can ease identifying invariant violations in applications and proposes possible solutions.

We are building a tool that, given the specification of an application's operations and invariants, identifies conflicts that might arise due to concurrent executions and proposes transformations to the operations to fix them, without strengthening the consistency model employed. The algorithm for identifying conflicts has already been published in our previous work [3]. We are extending this tool to propose transformations to the operations like the ones we described before. In the talk we show the tool in action to solve the referential integrity example.

5. FINAL REMARKS

In the course of the talk we have shown how to maintain a specific invariant by making good use of data type semantics. We started with a known invariant and the knowledge of the operations that might violate it, and devised two solutions for the problem. In practice, the programmer might not be as sensitive to the anomalies of weak consistency, might not be aware of the conflicts in his applications and might have difficulty in fixing violations, which makes development cumbersome.

Many recent works abandoned weak consistency towards stronger consistency models as these are easier to program and understand [9]. But the downsides of strong consistency come in many flavors: high latency, bad fault tolerance and bad scalability. To mitigate these problems, people have tried to pinpoint where weak consistency can be used and use strong consistency for everything else [1, 6]. Despite the performance improvements for operations that remain correct under weak consistency, it does not improve operations that require strong consistency. The bottom line of this talk is that is essential to to keep building on techniques to use weak consistency correctly, by preserving invariants and, at the same time, having the low latency, high availability benefits of the model. To sustain our argument, we have shown how to design proper convergence semantics to provide specific invariants and presented a tool that can relieve the programmer of the complexity of identifying and solving conflicts.

6. REFERENCES

[1] Alvaro, P., Conway, N., Hellerstein, J. M., and Marczak, W. R. Consistency analysis in bloom: A calm and collected approach. In *In Proceedings 5th Biennial Conference on Innovative Data Systems Research* (2011), pp. 249–260.

[2] Bailis, P., Fekete, A., Franklin, M. J., Ghodsi, A., Hellerstein, J. M., and Stoica, I. Feral concurrency control: An empirical investigation of modern application integrity. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (New York, NY, USA, 2015), SIGMOD '15, ACM, pp. 1327–1342.

[3] Balegas, V., Duarte, S., Ferreira, C., Rodrigues, R., Preguiça, N., Najafzadeh, M., and Shapiro, M. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems* (New York, NY, USA, 2015), EuroSys '15, ACM, pp. 6:1–6:16.

[4] Balegas, V., Serra, D., Duarte, S., Ferreira, C., Shapiro, M., Rodrigues, R., and Preguiça, N. M. Extending eventually consistent cloud databases for enforcing numeric invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015* (2015), pp. 31–36.

[5] Basho. Riak. `http://basho.com/riak/`, 2014. Accessed Fev/2016.

[6] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N., and Rodrigues, R. Making geo-replicated systems fast as possible, consistent when necessary. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2012), OSDI'12, USENIX Association, pp. 265–278.

[7] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Proceedings of the 13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)* (Grenoble, France, Oct. 2011), X. Défago, F. Petit, and V. Villain, Eds., vol. 6976 of *Lecture Notes on Computer Science*, Springer, pp. 386–400.

[8] Zawirski, M., Preguiça, N., Duarte, S., Bieniusa, A., Balegas, V., and Shapiro, M. Write fast, read in the past: Causal consistency for client-side applications. In *Proceedings of the 16th Annual Middleware Conference* (New York, NY, USA, 2015), Middleware '15, ACM, pp. 75–87.

[9] Zhang, I., Sharma, N. K., Szekeres, A., Krishnamurthy, A., and Ports, D. R. K. Building consistent transactions with inconsistent replication. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015* (2015), pp. 263–278.

6.7 Carlos Baquero, Paulo Sérgio Almeida, and Carl Lerche. The problem with embedded CRDT counters and a solution. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016.

The problem with embedded CRDT counters and a solution

Carlos Baquero
HASLab, INESC TEC & Universidade do Minho
Braga, Portugal
cbm@di.uminho.pt

Paulo Sérgio Almeida
HASLab, INESC TEC & Universidade do Minho
Braga, Portugal
psa@di.uminho.pt

Carl Lerche
Portland, Oregon
me@carllerche.com

ABSTRACT

Conflict-free Replicated Data Types (CRDTs) can simplify the design of deterministic eventual consistency. Considering the several CRDTs that have been deployed in production systems, counters are among the first. Counters are apparently simple, with a straightforward *inc/dec/read* API, but can require complex implementations and several variants have been specified and coded. Unlike sets and registers, that can be adapted to operate inside maps, current counter approaches exhibit anomalies when embedded in maps. Here, we illustrate the anomaly and propose a solution, based on a new counter model and implementation.

Keywords

Distributed Counting, Eventual Consistency, CRDTs.

1. INTRODUCTION

In order to support high-availability and low response latency in geo-replicated data storage systems, developers have successfully explored relaxed consistency models, such as eventual consistency [8, 1], and supporting frameworks, such as conflict-free replicated data types (CRDTs) [6, 7]. This trend towards fast querying and data manipulation at the edge, possibly under partitions, will likely become more prevalent with the growth of IoT deployments.

Complex CRDT deployments require mechanisms for composing together several base data types. A common strategy [5] is to define a replicated map data structure that maps keys to CRDT instances. In the Riak data store, maps can store sets, registers, flags, counters and even, recursively, other maps [3] (as they are CRDTs themselves). Maps need to support the addition and removal of entries (key bindings), and allow data type dependent updates over the stored CRDT instances.

2. EMBEDDED COUNTERS ANOMALY

In order to provide a sound semantics for key removal, CRDT maps behave in a way such that a non-present key (e.g., after removal), when fetched returns the default initial state, i.e., bottom, of the embedded CRDT. Efficient implementations require the storable CRDT data types to provide a special *reset* operation that brings the instance back to bottom, which need not be stored in the map, while allowing the map meta-data to remember the reset state in a efficient way, without requiring any per-key metadata; i.e., no per-key tombstone. Technically this is done by keeping a global causal context for the whole map, that is common to all the recursively embedded CRDTs. This can be thought of as a *observed-reset*, in the sense that all operations that have been observed to be applied to the map when the reset is issued, should be equivalently reset on another instance which observes the reset upon a join. Bellow is an example of a correct reset over an *add-wins set*.

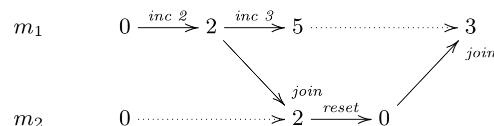
```
m1["friend"].add("alice");
m2.join(m1);
m2.remove("friend"); // m2: {}
m1["friend"].add("bob");
m1.join(m2); // m1: {"friend" -> {"bob"}}
```

After `m2.join(m1)` both replicas hold a mapping to a AWSet with a single "alice" element. Once replica `m2` removes the "friend" entry from the map, the set becomes implicitly empty. Concurrently, replica `m1` adds a new element "bob" to the set. Later, after joining the two replicas, we see that the *reset*, implicitly called on the set when removing the entry, only undoes the `add("alice")` and not the `add("bob")`.

We now change the example to illustrate the ideal (sound) semantics when counters are embedded. Initially we increment by 2 the "friend" entry and then we concurrently remove it and increment by 3. Ideally, removing the entry should undo the "increment by 2", which when merged to `m1`, would leave the "increment by 3" as the only remaining operation, and counter value of 3.

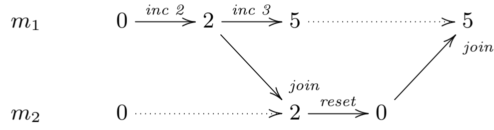
```
m1["friend"].inc(2);
m2.join(m1); m2.remove("friend");
m1["friend"].inc(3);
m1.join(m2); // m1: {"friend" -> 3}
```

The desired counter evolution would be:



Copyright is held by the owner/author(s). Publication rights licensed to ACM.
PaPoC'16 April 18, 2016, London, UK
Copyright 2016 ACM 978-1-4503-2716-9/14/04 ...$15.00.

However, embedding a simple CRDT counter implementation, and namely the behaviour of Riak DT Counters, exhibit an anomaly, leading to the following outcome:



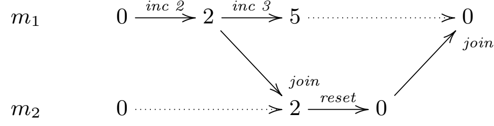
The problem is that the reset does not undo all observed operations (here the initial increment by 2) when merging to other replica, if such replica concurrently updates the counter. This limitation is known in Riak DT and it was an open problem to find an alternative solution [4].

3. A NEW EMBEDDED COUNTER

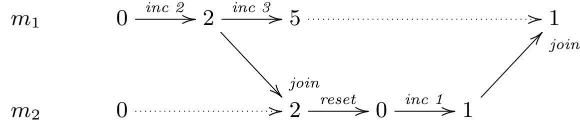
Our approach to address the problem is to try to obtain the desired observed-reset behaviour without compromising the scalability of the underlying meta-data. We found that this can be obtained in a *remove wins counter* design. In this counter all increments (and decrements) that are observed in a replica are correctly reset upon entry removal. Moreover, any concurrent operations are also affected by the reset, thus the remove wins behaviour. Lets illustrate this in our example.

```
m1["friend"].inc(2);
m2.join(m1); m2.remove("friend");
m1["friend"].inc(3);
m1.join(m2); // m1: {"friend" -> 0}
```

Leading to the counter evolution:



Notice that although concurrent operations are affected (both increments and decrements), any operations that causally follow the reset are not affected. Thus if we had done `m2["friend"].inc(1)` after `m2.remove("friend")` the outcome would have been 1:



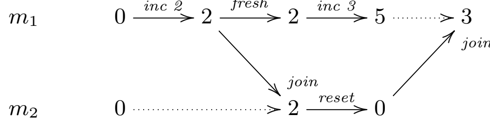
4. A SEMANTIC TRADE-OFF WITH STATE

Although we fixed the observed-reset anomaly, some applications might have a need for an *add wins counter* where reset only affects the (observed) past operations and leaves concurrent ones unaffected.

This behaviour is simple to obtain, but at a meta-data cost. The idea is to provide a `fresh()` operation which has the effect of protecting subsequent updates from being affected by resets concurrent to it. The following example shows that, by calling `fresh()`, the `inc(3)` operation is not affected by the concurrent reset, and the outcome is 3, as desired.

```
m1["friend"].inc(2);
m2.join(m1); m2.remove("friend");
m1["friend"].fresh(); m1["friend"].inc(3);
m1.join(m2); // m1: {"friend" -> 3}
```

Leading to the counter evolution:



In the counter presented below, if no `fresh()` calls are made the counter scalability is $O(r \log o)$, as usual, where r is the number of replicas that issued operations and o is the number of operations done. If we consider an arbitrary number of fresh calls $f \geq r$, the scalability becomes $O(f \log o)$, when fresh calls are made. Therefore, if meta-data size is a concern then `fresh()` should be called sparingly. The good news is that, upon resets, meta-data comes back to $O(r \log o)$, so only the number of observed `fresh()` calls until a reset is relevant. Moreover, to obtain the ideal semantics, it is enough to perform a `fresh()` only after shipping the state to other replicas. A possible direction towards obtaining state-scalability could involve a combination of the use of `fresh`, together with periodic resets through some replica coordination.

5. COUNTER SPECIFICATION

Figure 1 shows a mathematical specification of the proposed counter. The state is a pair (m, c) formed by a map m from replica generated ids to a pair of integers, and by a map c (referred to as causal context) from replica ids to integers that compactly encodes causality (essentially a version vector). The first map is what we call a dot store; each key, called a dot, serves as globally unique id, being formed by a pair of replica id and a monotonically increasing counter; the value is a pair of integers that contain the positive and negative partial counts registered under that key.

Initially both the dot store and the causal context are empty, and the reported count value, by query function value_i, returns 0. In order to increment or decrement the counter at replica i an active entry for i must be found, or created, in the dot store m. Mutator functions inc_i and dec_i invoke an auxiliary mutator upd_i with a pair (either $(1, 0)$ or $(0, 1)$) containing the number of increments and decrements to be applied to the partial count on an active entry for replica i. Thus, function upd_i before updating must check if an active entry is already available for replica i in the dot store m or create a new one by calling fresh_i; if an active entry exists in m its key corresponds to the more recent dot known in the causal context in i, i.e., dot $(i, c(i))$.

The fresh_i mutator will always create a new entry in m by creating a new (globally unique) dot, with replica id i and the next sequence number, $c(i) + 1$, and map it to the $(0, 0)$ *positive-negative* partial count. It leaves other entries untouched and, therefore, does not change the counter value. The query function value_i simply sums up all the positive values in the active map and subtracts the corresponding negative ones. By calling reset_i all mappings are removed from the dot store and only the causal context is preserved; thus, the reported value will be again 0. When counter CRDTs are embedded inside maps, a *reset* is called on the

$$
\begin{aligned}
\mathsf{Counter} &= (\mathbb{I} \times \mathbb{N} \hookrightarrow \mathbb{N} \times \mathbb{N}) \times (\mathbb{I} \hookrightarrow \mathbb{N}) \\
\bot &= (\{\}, \{\}) \\
\mathsf{inc}_i(s) &= \mathsf{upd}_i(s, (1, 0)) \\
\mathsf{dec}_i(s) &= \mathsf{upd}_i(s, (0, 1)) \\
\mathsf{upd}_i((m, c), u) &= (m'\{d \mapsto m'(d) + u\}, c') \text{ where } d = (i, c'(i)), \\
&\quad (m', c') = \begin{cases} \mathsf{fresh}_i((m, c)) & \text{if } (i, c(i)) \notin \mathsf{dom}\, m \\ (m, c) & \text{otherwise} \end{cases} \\
\mathsf{fresh}_i((m, c)) &= (m\{(i, c(i) + 1) \mapsto (0, 0)\}, c\{i \mapsto c(i) + 1\}) \\
\mathsf{reset}_i((m, c)) &= (\{\}, c) \\
\mathsf{value}_i((m, c)) &= \sum_{d \in \mathsf{dom}\, m} \mathsf{fst}\, m(d) - \mathsf{snd}\, m(d) \\
(m, c) \sqcup (m', c') &= (\{d \mapsto m(d) \sqcup m'(d) \mid d \in \mathsf{dom}\, m' \cap \mathsf{dom}\, m\} \cup \\
&\quad \{((j, n), v) \in m \mid n > c'(j)\} \cup \{((j, n), v) \in m' \mid n > c(j)\}, \\
&\quad c \sqcup c')
\end{aligned}
$$

Figure 1: Resettable Counter, replica i.

counter instance when the corresponding key is removed. Since CRDTs embedded in maps all share a common causal context, removing a map entry effectively removes all the state associated to the counter instance.

Finally, the join function will: look for entries in common among the two maps m, m' and join the corresponding values by taking the pairwise maximum of the two *positive-negative* values; and for entries that are present only in one map, only those whose dot was never seen in the other causal context are preserved. This is done by checking if the number n in the dot (j, n) is strictly higher than the highest entry known for j in the other causal context. The joined causal context is simply obtained, as usual for version vectors, by coordinate-wise maximum between the maps c and c'.

6. FINAL REMARKS

The new counter design we propose in this paper addresses the problem that prevented counters to be embedded in a map and still provide a *reset* that would correctly remove all past operations, to be used when removing an entry from the map. Even when not embedded, current counters still have that problem if applications require a reset operation.

The solution we propose is efficient, in terms of meta-data cost, under a *remove-wins* semantics. The alternative, *add-wins*, that protects operations from being cancelled by concurrent resets, has considerable meta-data cost. This cost can be reduced by a system design that only creates *fresh* entries after the counter state is sent to other replicas, possibly accepting the trade-off of a low rate of dissemination and less overall recency. Further research is needed, to evaluate this cost and to attempt garbage collection of entries possibly through *reset* together with some coordination.

Reference implementations for the various counters, including the Riak DT counter (CCounter) and the proposed counter (RWCounter), are publicly available in GitHub [2] for C++. *Rust* implementations are under development.

7. ACKNOWLEDGMENTS

We thank the following funding sources: Project Norte-01-0145-FEDER-000020 is financed by the North Portugal Regional Operational Programme (Norte 2020), under the Portugal 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF). Funding from the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement 609551, SyncFree project.

8. REFERENCES

[1] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Queue*, 11(3):20:20–20:32, Mar. 2013.

[2] C. Baquero. Delta-enabled-crdts. URL `http://github.com/CBaquero/delta-enabled-crdts`, Retrieved 22-dec-2015.

[3] Basho. Riak datatypes. URL `http://github.com/basho`, Retrieved 22-dec-2015.

[4] R. Brown. Personal Communication, Jan 2016.

[5] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM.

[6] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, INRIA, Rocquencourt, France, Jan. 2011.

[7] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Int. Symp. on Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976 of *Lecture Notes in Comp. Sc.*, pages 386–400, Grenoble, France, Oct. 2011. Springer-Verlag.

[8] W. Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, Oct. 2008.

6.8 Albert van der Linde, João Leitão, and Nuno Preguiça. Δ-CRDTs: Making δ-CRDTs Delta-Based. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016.

Δ-CRDTs: Making δ-CRDTs Delta-Based

Albert van der Linde
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa

João Leitão
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa

Nuno Preguiça
NOVA LINCS, DI, FCT,
Universidade NOVA de Lisboa

ABSTRACT

Replication is a key technique for providing both fault tolerance and availability in distributed systems. However, managing replicated state, and ensuring that these replicas remain consistent, is a non trivial task, in particular in scenarios where replicas can reside on the client-side, as clients might have unreliable communication channels and hence, exhibit highly dynamic communication patterns. One way to simplify this task is to resort to CRDTs, which are data types that enable replication and operation over replicas with no coordination, ensuring eventual state convergence when these replicas are synchronized. However, when the communication patters, and therefore synchronization patterns, are highly dynamic, existing designs of CRDTs might incur in excessive communication overhead. To address those scenarios, in this paper we propose a new design for CRDTs which we call Δ-CRDT, and experimentally show that under dynamic communication patters, this novel design achieves better network utilization than existing alternatives.

1. INTRODUCTION AND CONTEXT

Web applications running in cloud infrastructures often use geo-replication for providing high availability and low latency to clients. To be able to continue operating during network partitions, these systems often adopt weakly consistent data replication protocols [2]. Such protocols allow replicas to be modified concurrently, requiring some reconciliation mechanism to merge these concurrent updates.

CRDTs [5] have been proposed as a principled approach for providing convergence of general purpose data type. CRDTs come in two main flavors. State-based CRDTs synchronize by having replicas exchange their full local state (including metadata). This is inefficient when the size of these data objects grow significantly (for instance, in a large Set, the full Set needs to be propagated whenever a single element is added).The second flavor of CRDTs is called Operation-based CRDTs, where instead of exchanging the full state, replicas only propagate among them the operations that mutate their state. In this case, operations have to be propagated respecting the causality of operations, which not only introduces additional overhead (to keep track of causality) but also fits poorly in scenarios where there are large number of replicas, and where communication patterns among these replicas are highly dynamic, for instance, due to poor connectivity among these replicas.

A recent alternative, named δ-CRDTs [1] has been proposed as a middle ground between the two approaches. δ-CRDTs assumes that communication is mostly pairwise, with each replica maintaining a communication buffer for each of its peers where it stores the operations that have not been propagated (and acknowledged) to the remote peer. These buffers are used to compress multiple operations into a single delta, and enforce FIFO communication semantics between each pair of replicas. Whenever a new synchronization path is established between two replicas, the whole state of both replicas has to be synchronized by resorting to a mechanism similar to those employed in State-based CRDTs. Thus, this approach works well, only in settings with continued and static synchronization patterns among replicas.

In this paper, we introduce an extension to δ-CRDTs that we name Δ-CRDTs. Δ-CRDTs were specially designed to support dynamic communication patterns among a potentially large number of replicas, and removes the assumption that pairs of replicas are continuously communicating to synchronize their state. Additionally, Δ-CRDTs do not resort to specialized pairwise communication buffers, minimizing the space overhead imposed over each individual replica. Instead, we use the CRDT internal metadata to compute the minimal Delta that needs to be propagated to a remote replica, based on a causal context (usually, a vector clock) that replicas exchange[1]. Due to this, Δ-CRDTs are well suited to be used in decentralized dissemination protocols, such as gossip protocols [4]. To achieve its properties, when compared to optimized δ-CRDTs, Δ-CRDTs needs to temporarily maintain additional metadata (tombstones). However, this metadata can be garbage collected locally at any time, at the price of being unable to synchronize by sending only a delta when the garbage-collected information is needed for computing the delta. If this happens, the full state needs to be exchanged (as it is always the case when starting a new connection in δ-CRDTs).

We have run a set of preliminary experiments, using a

[1]Riak support for big sets uses a similar idea for efficiently identifying removed elements [3].

Copyright is held by the owner/author(s). Publication rights licensed to ACM.
PaPoC'16 April 18, 2016, London, UK
Copyright 2016 ACM 978-1-4503-2716-9/14/04 ...$15.00.

Algorithm 1: Δ-CRDT replication

```
upon onVersionVector(VV, REPLICA) do
    Δ ⟵ getDelta(vv)
    if Δ.size() > 0
        REPLICA.send(Δ)
    optionally do (push model)
        if vv after self.versionVector
            REPLICA.send(self.versionVector)

upon delta(Δ) do
    self.state.applyDelta(Δ)
    self.versionVector.update(Δ)

periodically do (pull model)
    r ⟵ randomReplica()
    r.send(self.versionVector)

on local operation do (push model)
    r ⟵ randomReplica()
    r.send(self.versionVector)
```

web-based framework, where we compare the performance of our Δ-CRDTs with that of State-based and Operation-based CRDTs, and have observed that our approach enables a better network usage.

2. Δ-CRDTS

Δ-CRDTs are replicated by propagating a delta (Δ) of the current state that is missing in a particular replica. To compute the Δ, a *getDelta* function is called with the causal context of the replica which initiated the communication (i.e., which requires missing updates). This causal context can be sent by a requesting replica (*pull model*) or, when local operations are performed, sent to other replicas (*push model*)[2]. Algorithm 1 shows how a simple replication protocol can be created by leveraging Δ-CRDTs. A replica receives a causal context (version vector) from a replica and computes a Δ that is to be shipped back. A replica can receive a causal context (version vector) from a replica that is simultaneously older and newer than its own context. This means that both replicas have executed operations (concurrently) that the other has not yet seen, and thus both a Δ and a causal context have to be shipped (as to ensure the other replica also computes and send a Δ back to that node).

To create Δ-CRDT the following methods have to be implemented: a *delta* function must be implemented to be able to compute a Δ from a given point in time (i.e., the causal history, typically in the form of a version vector); a *applyDelta* function must be implemented which applies a given delta to the current state.

In container like data types, such as Sets and Maps, CRDTs typically associate an unique timestamp to each data-item. To avoid concurrent add-remove anomalies, typically these data-types use a remove-set of unique timestamps, which are called tombstones. In our Δ-CRDTs we use as unique timestamp pairs of *replicaID* and *operationNumber*. This ensures that each existing data-item and tombstone can be related to any given version vector (as to be before or after that point).

Notice that causality is maintained by the same principle associated with shipping the whole state when using State-based CRDTs. *getDelta* always returns the complete Δ and thus all missing operations on the other replica are sent in a single message. A Δ is always added to the local state in a single execution step (i.e., no other methods should be able to access the internal data-structures), and thus causality is implicitly maintained. Note however, that when two replicas are synchronized, or when a replica receives a causal context that is in its future, the generated Δ will be empty.

To be able to compute the delta from a given causal context, Δ-CRDTs need to maintain metadata about deleted elements (note that δ-CRDTs also need to maintain such information in pairwise communication buffers). In order to keep the amount of wasted space small we remove old metadata periodically (i.e., we provide a mechanism to garbage collect old tombstones). A *garbageCollection* function is added which removes old metadata associated with all operations that happened before a given point in time (also denoted by a version vector).

When garbage collection occurs, the previously described *applyDelta* function has to be able to still infer if some portion of the current local state is outdated (i.e., removed data-items whose's tombstones have been garbage collected). The *getDelta* function is adapted to handle the (typically rare) case where the local replica's garbage collection point is further ahead in time than the sender's causal context. In this case, a Δ-CRDT falls back to a State-based CRDT merge procedure, where the whole state, including the causal context of the last garbage collection step, have to be shipped and integrated by the remote replica.

The main drawback of using Δ-CRDTs is expected to be an increase in latency for replicas to receive operations. Typically State-based CRDTs and Operations-based CRDTs use a *push model* to propagate local changes to a replica. Though these data-types are able to immediately send the changes, Δ-CRDTs need an additional communication step between replicas. Typically, a version vector is first sent, and then a delta is sent back which can be locally applied. A version vector can also be piggy-backed along with the delta, as to ensure the initiating replica also ships any locally applied changes that the remote replica has not yet received. When using Δ-CRDTs with stable communication patterns, the additional communication step is paid only when establishing the connection.

When used in a scenario with dynamic communication patterns and compared to δ-CRDTs, Δ-CRDTs have the following advantages: (1) Δ-CRDTs do not require each replica to maintain a buffer for each of its connections; (2) by using the information exchanged in the vector clocks, a replica will only send the minimal Delta needed by the remote replica, instead of sending all the information stored in the Delta (that might have arrived to the remote replica through a different communication path).

3. PRELIMINARY RESULTS

To have an initial feel for the feasibility of Δ-CRDTs in comparison to Delta-based CRDTs or Operation-based CRDTs in a real setting we compare the usage of each type of CRDT in a peer-to-peer setting. We implemented Δ-CRDTs, State-based CRDTs, and Operation-based CRDTs, namely an Observe-Remove Set, extending an existing browser-based peer-to-peer framework which has support for State-based CRDT and Operation-based CRDT replication.

We run multiple nodes (each node owns a replica of a single replicated set) in a peer-to-peer setting. The interactions between active peers is dynamic, i.e., replicas communicate

[2]The distinction between *pull* and *push* can be found in [4]

with a random sub-set of all existing replicas at each synchronization step.

3.1 Implementation

Communication between replicas happens every T seconds. In a synchronization step, a random subset of the currently connected neighbours are selected by a peer. At this point, the causal context of the initiating replica is set to those peers (hence, we use a pull communication model). In contrast State-based CRDTs the whole state is shipped to the randomly selected peers; when using Operation-based CRDTs the version vector is also shipped. This happens because in our experimental setting there is no continuous flow of messages between pairs of nodes (i.e., the communication patterns change at each synchronization step). The alternative would require each replica in the system to maintain information about all operations which have previously been sent and acknowledged. The remote replica will use this vector clock to send back missing operations (and its own version vector).

Note that while we use a *push model* when propagating State-based CRDTs, a *pull model* is employed for Operation-based CRDTs and Δ-CRDTs.

3.2 Experimental Setup

We run 8 clients in a browser-based peer-to-peer framework where each client continuously issues operations over the replicated CRDT Set. Each client was ran in its own Google Chrome instance, on a local machine (MacBook Pro Retina, 16GB RAM). All reported results are the mean result of three independent runs.

We compare the sizes of messages sent between clients when using Δ-CRDTs, State-based CRDTs, and Operation-based CRDTs. The Set is updated, by each peer, twice per second. Each peer, per update, has a 30 % change to remove an existing data-item and 70 % change to add a new data-item (a string with 14 characters, with 2 bytes per char resulting in 28 bytes per key). Each peer contacts 2 randomly selected peers, every 5 seconds, as to begin state reconciliation between them (as discussed previously).

3.3 Results

Figure 1 reports the obtained results showing the size, in bytes, of all messages exchanged between replicas, with a sampling interval of one second (object related messages only, including state, operations, Δs, and version vectors when applicable). As expected, State-based CRDTs have an always growing load on the network. As more operations are executed more state has to be exchanged between replicas. The currently implemented Operation-based CRDTs are not optimized for the employed communication model and thus incur an initial load penalty. As only operations are sent over the network (along with version vectors), eventually the network load becomes lower than state propagation. Δ-CRDTs propagate less data over the network as, when the total amount of applied operations increases, what is shipped between clients is always a Δ where this Δ is much smaller that the whole state of the object.

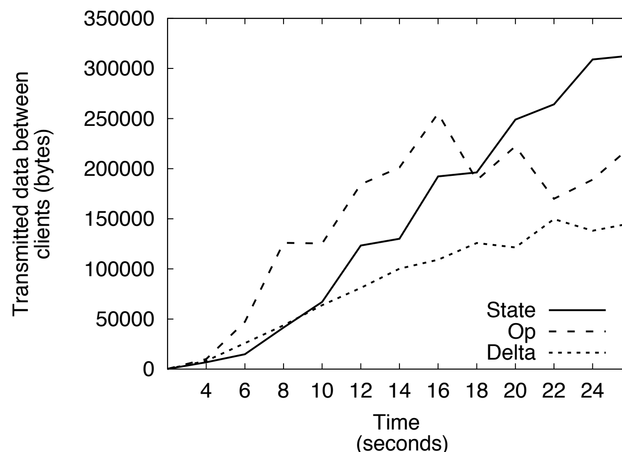


Figure 1: Communication cost, results for 8 replicas sharing a Observe-Remove Set. Each replica issues an operations every 500ms. The workload is composed of 70% inserts and 30% removes.

4. CONCLUSION AND FUTURE WORK

The results that we have reported show that, in a scenario with highly dynamic communication patterns, Δ-CRDTs clearly outperform, from the standpoint of network usage, the competing alternative (we are currently updating our browser-based peer-to-peer framework to make use of Δ-CRDTs due to this). We intend to further implement and evaluate Δ-CRDTs in a distributed setting (scenarios with geo-replication for instance). We will also continue working on improving the current implementation of Δ-CRDTs, as the current implementations have unbounded growth on the version vector size (i.e., it is bounded by the amount of replicas in the system which can grow significantly when pushing replicas to the client side).

5. REFERENCES

[1] P. S. Almeida, A. Shoker, and C. Baquero. Efficient state-based crdts by delta-mutation. In A. Bouajjani and H. Fauconnier, editors, *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, volume 9466 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2015.

[2] E. Brewer. Towards robust distributed systems (abstract). In *ACM PODC*, page 7, 2000.

[3] R. Brown. Riak support for big sets (private communication), 2015.

[4] J. Leitão. *Topology Management for Unstructured Overlay Networks.* PhD thesis, Technical University of Lisbon, Sept. 2012.

[5] M. Shapiro, N. M. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011.

6.9 Mahsa Najafzadeh, Alexey Gotsman, Hongseok Yang, Carla Ferreira, and Marc Shapiro. The CISE Tool: Proving Weakly-Consistent Applications Correct. In Proceedings of the Second Workshop on Principles and Practice of Consistency for Distributed Data (to appear), PaPoC '16. ACM, 2016.

The CISE Tool: Proving Weakly-Consistent Applications Correct

Mahsa Najafzadeh
Sorbonne-Universités-UPMC & Inria, Paris, France

Alexey Gotsman
IMDEA Software Institute, Spain

Hongseok Yang
University of Oxford, UK

Carla Ferreira
NOVA LINCS, DI, FCT, U. NOVA de Lisboa, Portugal

Marc Shapiro
Sorbonne-Universités-UPMC & Inria, Paris, France

Abstract

Designers of a replicated database face a vexing choice between strong consistency, which ensures certain application invariants but is slow and fragile, and asynchronous replication, which is highly available and responsive, but exposes the programmer to unfamiliar behaviours. To bypass this conundrum, recent research has studied hybrid consistency models, in which updates are asynchronous by default, but synchronisation is available upon request. To help programmers exploit hybrid consistency, we propose the first static analysis tool for proving integrity invariants of applications using databases with hybrid consistency models. This allows a programmer to find minimal consistency guarantees sufficient for application correctness.

1 Introduction

To achieve availability and scalability, many modern distributed systems rely on *replicated databases*, which maintain multiple replicas of shared data. Clients can access the data at any of the replicas, and these replicas communicate changes to each other using message passing. For example, large-scale Internet services use data replicas in geographically distinct locations, and applications for mobile devices keep replicas locally to support offline use. Ideally, we would like replicated databases to provide *strong consistency*, i.e., to behave as if a single centralised node handles all operations. However, achieving this ideal usually requires synchronisation among replicas, which slows down the database and even makes it unavailable if network connections between replicas fail [1, 7].

For this reason, modern replicated databases often eschew synchronisation completely; such databases are commonly dubbed *eventually consistent* [14]. In these databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately returns to the client; the *effect* of the operation is propagated to the other replicas only *eventually*. Unfortunately, this way of processing operations exposes applications to undesirable concurrency behaviours, which may cause bugs such as state divergence or invariant violation [6].

For instance, consider a bank account replicated at different bank branches, which supports operations `deposit` and `withdraw`. A programmer would like to ensure an *integrity invariant* that the balance is never negative. Assume that the balance is initially €100. Eventual consistency will allow two users to concurrently withdraw €60 at different branches and thus violate the integrity invariant. To ensure the invariant in this example, we have to introduce synchronisation between replicas, and, since synchronisation is expensive, we would like to introduce it sparingly. To allow this, some research [3, 9, 12, 13] and commercial [2, 4, 10] databases now provide *hybrid* consistency models that allow the programmer to request stronger consistency for certain operations and thereby introduce synchronisation. For example, to preserve the integrity invariant in our banking application, only `withdraw` operations need to use strong consistency, and hence, synchronise to ensure that the account is not overdrawn; `deposit` operations may use eventual consistency and hence proceed without synchronisation.

Unfortunately, using hybrid consistency models effectively is far from trivial. Requesting stronger consistency in too many places may hurt performance and availability, and requesting it in too few places may violate correctness. Striking the right balance requires the programmer to reason about the application behaviour on the subtle semantics of the consistency model, taking into account which anomalies are disallowed by a particular consistency strengthening and whether disallowing these anomalies is enough to ensure correctness.

To help programmers exploit hybrid consistency models, we propose the first static analysis tool (called CISE: 'Cause I'm Strong Enough) for proving integrity invariants of applications using replicated databases with a range of hybrid models. Our tool is based on a novel proof rule, which we have proved sound [8]. The tool automates the proof rule by discharging its obligations using an SMT solver. If an obligation fails, the tool provides a counter-example, which the developer can use to understand the source of the problem.

Using the tool, we have verified several example applications that require strengthening consistency in nontrivial ways [8]. These include an extension of the above banking application, an online auction service and a course registration system. A demo of the tool is available online [11].

In the rest of this paper, we explain our static analysis by the example of the above banking application.

2 System Model

An application consists of a set of operations Op over some set of objects, and invariants over the objects. The database system consists of a set of replicas, each maintaining a full copy of the database state State.

The replication model uses a Read-One-Write-All (ROWA) approach [5]. A client operation is initially executed at a single replica, which we refer to as its *origin* replica. This updates the replica state deterministically, and immediately returns a value to the client. After this, the replica sends a message to all other replicas containing the *effector* of the operation, which describes the updates done by the operation to the database state. Upon receipt, the replicas apply the effector to their state. Effectors of causally-dependent operations are executed in the same order at every replica; effectors of independent (concurrent) operations are executed in any order.

More precisely, the semantics of operations is defined by a partial function

$$\mathcal{F} \in \mathsf{Op} \rightarrow (\mathsf{State} \rightharpoonup (\mathsf{Val} \times (\mathsf{State} \rightarrow \mathsf{State}) \times \mathcal{P}(\mathsf{Token}))).$$

Given a state $\sigma \in \mathsf{State}$ in which an operation $o \in \mathsf{Op}$ executes at its origin replica, $\mathcal{F}(o)(\sigma)$ determines:

- The return value of the operation, from a set Val. We use a special value $\perp$ for operations that return no value.
- A function defining the *effector* of the operation. This will be applied by every replica to its state: immediately at the origin replica, and after receiving the corresponding message at all other replicas.
- A set of *tokens*, used to introduce synchronisation. We explain them later.

For instance, consider the naïve banking application in Figure 1. A client can read the balance from the local replica, make deposits to and withdrawals from the account, and compute interest, all without communicating with the other replicas. Each operation is associated with a *precondition*—a predicate over the state of its origin replica and parameters that determines when the operation can be safely executed (and the $\mathcal{F}$ function defined). A minimal precondition of the $\mathsf{deposit}(\mathsf{amount})$ and $\mathsf{withdraw}(\mathsf{amount})$ operations is $\mathsf{amount} \geq 0$. Their effectors add amount to (respectively, subtract it from) the balance. The $\mathsf{interest}$ operation's precondition is true and its effector multiplies the balance by the interest rate. (As we will see later, the analysis shows that the precondition of $\mathsf{withdraw}$ needs to be strengthened, and that this effector of $\mathsf{interest}$ is unsafe.)

$$\begin{aligned}
\sigma_{\mathsf{init}} &= 0 \\
I &= (\mathsf{balance} \geq 0) \\
\mathsf{Token} &= \emptyset \\
\mathcal{F}_{\mathsf{deposit(amount)}}(\mathsf{balance}) &= (\perp, (\lambda \mathsf{balance}'.\ \mathsf{balance}+ \\
&\quad \mathsf{amount}), \emptyset) \\
\mathcal{F}_{\mathsf{interest()}}(\mathsf{balance}) &= (\perp, (\lambda \mathsf{balance}'.\ (1.05* \\
&\quad \mathsf{balance}')), \emptyset) \\
\mathcal{F}_{\mathsf{withdraw(amount)}}(\mathsf{balance}) &= (\perp, (\lambda \mathsf{balance}'.\ \mathsf{balance}- \\
&\quad \mathsf{amount}), \emptyset)
\end{aligned}$$

Precondition	Operation
$\mathsf{amount} \geq 0$	$\mathsf{deposit}(\mathsf{amount})$
true	$\mathsf{interest}()$
$\mathsf{amount} \geq 0$	$\mathsf{withdraw}(\mathsf{amount})$

Figure 1: Simple banking application (incorrect).

3 CISE Analysis

3.1 Effector Safety Analysis

The first CISE proof obligation, called the effector safety analysis, verifies that the effector of every operation maintains the invariant when applied to *any* state where the operation's precondition is true (not necessarily the one in which the operation was generated).

Let's try out the effector safety analysis on the simple banking application of Figure 1. According to the analysis, the effectors of $\mathsf{deposit}$ and $\mathsf{interest}$ always maintain the invariant. However, for $\mathsf{withdraw}$, the obligation fails and our tool produces a counter-example: if the balance is zero, a non-zero $\mathsf{withdraw}$ operation makes the balance negative. Therefore, we must fix the issue by strengthening its precondition, so that the amount debited is less or equal than the current balance.

With this correction, the effector safety analysis succeeds. The corrected preconditions are shown at the bottom of Figure 2.

3.2 Commutativity Analysis

Effectors of concurrent operations may execute in different orders at different replicas. The second CISE obligation, called the commutativity analysis, checks if all pairs of effectors of such operations *commute*: executing them in any order yields the same result, whatever the starting state.

$$\sigma_{\mathsf{init}} = 0$$
$$I = \mathsf{balance} \geq 0$$
$$\mathsf{Token} = \{\tau\}$$
$$\bowtie = \{(\tau, \tau)\}$$
$$\mathcal{F}_{\mathsf{deposit(amount)}}(\mathsf{balance}) = (\bot, (\lambda \mathsf{balance}'.\, \mathsf{balance} + \mathsf{amount}), \emptyset)$$
$$\mathcal{F}_{\mathsf{interest()}}(\mathsf{balance}) = (\bot, (\lambda \mathsf{balance}'.\, (\mathsf{balance}' + 0.05 * \mathsf{balance})), \emptyset)$$
$$\mathcal{F}_{\mathsf{withdraw(amount)}}(\mathsf{balance}) = (\bot, (\lambda \mathsf{balance}'.\, \mathsf{balance} - \mathsf{amount}), \{\tau\})$$

Precondition	Operation
$\mathsf{amount} \geq 0$	deposit(amount)
true	interest()
$\mathsf{balance} \geq \mathsf{amount} \geq 0$	withdraw(amount)

Figure 2: Corrected banking application.

Let us check this obligation for the specification in Figure 1. Predictably, applying the commutativity analysis proves that deposit and withdraw effectors commute. However, the effector of interest does not commute with that of the other operations, and the tool returns a counter-example. Consider two replicas 1 and 2. The balance is initially €100. Replica 1 is the origin for an interest operation. Replica 2 is the origin for a deposit(20) operation. Replica 1 first applies the effector of interest and then that of deposit, whereas replica 2 applies them in the opposite order. Depending on the order of execution, the result is different, and the replicas diverge.

We fix this by changing the interest operation to compute the absolute interest at the origin replica and letting its effector add this amount to the local balance of every replica (Figure 2). With this corrected specification, our tool proves that the effector of interest does commute with those of the other operations.

3.3 Stability Analysis

The effector safety analysis verified that that the effector of each operation o maintains the invariant when executed in a state satisfying the precondition of the operation. The precondition holds at o's origin replica, but how do we know that it will hold when o's effector is applied at a different replica, which concurrently executes effectors of other operations? The third obligation of CISE analysis, called stability analysis, checks if executing the effector of any other operation o' maintains the precondition of o.

Let us illustrate the stability analysis of the withdraw operation in Figure 1. The precondition of the withdraw operation is stable under the effectors of deposit and interest, but it is not stable under the effector of withdraw. The tool returns the following counter-example. Let the balance be €2. The precondition to withdraw(1) is verified. However, a concurrent withdraw(2) (whose precondition is also OK) at a different replica makes the balance zero, now violating the precondition of withdraw(1). If we were to continue, and to apply the effector of the first withdrawal operation at the second replica, the balance would become negative.

To fix the problem, the developer of the banking application may disallow the execution of withdrawals without synchronisation. To model such concurrency control, we use *tokens* $\mathsf{Token} = \{\tau, \ldots\}$ and a symmetric *conflict relation* $\bowtie \subseteq \mathsf{Token} \times \mathsf{Token}$ between pairs of them. In the banking application, we associate a token τ to withdraw such that $\tau \bowtie \tau$ (similarly to a mutual exclusion lock). This ensures that any two withdrawals synchronise.

Figure 2 presents the corrected banking application, incorporating all the changes outlined above. Our static analysis confirms that this application indeed maintains the integrity invariant.

4 Future Work

In the future, we plan to study proof rules for reasoning about integrity invariants on consistency models weaker than causal consistency. We also intend to automate the analysis of counter-examples in order to generate corrections semi-automatically.

References

[1] D. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.

[2] Amazon. Supported operations in DynamoDB. `http://docs.aws.amazon.com/amazondynamodb/latest/developerguide/APISummary.html`, 2015.

[3] V. Balegas, N. Preguiça, R. Rodrigues, et al. Putting consistency back into eventual consistency. In *Euro. Conf. on Comp. Sys. (EuroSys)*, pp. 6:1–6:16, Bordeaux, France, Apr. 2015.

[4] Basho Inc. Using strong consistency in Riak. `http://docs.basho.com/riak/latest/dev/advanced/strong-consistency/`, 2015.

[5] P. Bernstein, V. Radzilacos, and V. Hadzilacos. *Concurrency Control and Recovery in Database Systems*. Addison Wesley Publishing Company, 1987.

[6] J. C. Corbett, J. Dean, M. Epstein, et al. Spanner: Google's globally-distributed database. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 251–264, Hollywood, CA, USA, Oct. 2012.

[7] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002. ISSN 0163-5700.

[8] A. Gotsman, H. Yang, C. Ferreira, et al. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *Symp. on Principles of Prog. Lang. (POPL)*, pp. 371–384, St. Petersburg, FL, USA, 2016.

[9] C. Li, D. Porto, A. Clement, et al. Making geo-replicated systems fast as possible, consistent when necessary. In *Symp. on Op. Sys. Design and Implementation (OSDI)*, pp. 265–278, Hollywood, CA, USA, Oct. 2012.

[10] Microsoft. Consistency levels in DocumentDB. `http://azure.microsoft.com/en-us/documentation/articles/documentdb-consistency-levels/`, 2015.

[11] M. Najafzadeh and M. Shapiro. Demo of the CISE tool, Nov. 2015. `https://youtu.be/HJjWqNDh-GA`. Video of demo, with explanations.

[12] Y. Sovran, R. Power, M. K. Aguilera, et al. Transactional storage for geo-replicated systems. In *Symp. on Op. Sys. Principles (SOSP)*, pp. 385–400, Cascais, Portugal, Oct. 2011.

[13] D. B. Terry, V. Prabhakaran, R. Kotla, et al. Consistency-based service level agreements for cloud storage. In *SOSP*, 2013.

[14] W. Vogels. Eventually consistent. *CACM*, 52(1), 2009.

7 Submitted papers

7.1 Mathias Weber, Annette Bieniusa, and Arnd Poetzsch-Heffter. Access control for weakly consistent cloud-storage systems. Submitted for publication, 2016.

Access Control for Weakly Consistent Cloud-Storage Systems

Mathias Weber
University of Kaiserslautern
m_weber@cs.uni-kl.de

Annette Bieniusa
University of Kaiserslautern
bieniusa@cs.uni-kl.de

Arnd Poetzsch-Heffter
University of Kaiserslautern
poetzsch@cs.uni-kl.de

***Abstract*—Access control is an important aspect of information systems as these systems store sensitive information. Security policies describe the rules that applied to determine whether a user is allowed to perform a specific operation. Typically, these rules need to be adaptable over time, leading to the access control system has to be updated at runtime. For strongly consistent systems, the implementation of access control is well understood since the order in which operations are processed is the same for all replicas. In weakly consistent systems, however, concurrent modifications and out-of-order delivery of data updates and policy changes impose security threats due to inconsistencies among policies and data operations.**

In this paper, we present an access control model for systems for eventually consistent data stores that avoids information leakage, unauthorized modifications and guarantees convergence of the copies of the security policy among replicas. In particular, we address policy changes which restrict the visibility of the effect of data operations where re-ordering of updates temporarily has the data operation unprotected. Our model allows to implement access-matrix based models such as the read-write-own model employed in file systems and can be implemented efficiently in state-of-the-art weakly consistent data stores.

***Index Terms*—Access control, security, weak consistency.**

I. Introduction

Information systems often store sensitive information of customers, clients and users in cloud storage facilities. To protect this information from unauthorized access, the organization running such an information system needs to define a security policy to determine who may access and/or modify which subset of the data. This security policy is implemented in an access control system which permits only those operations that satisfy the policy. In general, a security policy is not immutable. New information is constantly entered into the system, and organizational changes cause adaptations of the security policy. For example, in social networks, users want to be able to restrict access to their personal information when interpersonal relations change. After such a policy change, the new policy must be employed in the access control system for all operations happening afterwards. Since restarting the system for each policy change is not feasible due to availability requirements, the access control system must support dynamic changes of the policy at runtime.

For strongly consistent systems, the topic of dynamically adaptable access control is well understood. Several access control models have been proposed [7, 11, 12, 14, 16] which implicitly rely on some total ordering of the operations as strongly consistent systems induce on all operations. However, Brewer's conjecture [5, 8], having become known as the CAP theorem, states that in a network without total synchronization of clocks and with possible message loss, no service can implement strong consistency, high availability and partition tolerance. As partitioning cannot be prevented in computer networks due to network hardware and replicas failing, implementors of data stores have to make the trade-off between high availability and strong consistency. In many systems [1, 2, 3, 6], consistency is traded over high availability. In such weakly consistent systems, updates are accepted at any replica and propagated asynchronously to the other replicas. These synchronization messages coming from different replicas can arrive in an arbitrary order on a node. To achieve higher throughput, one usually uses a connectionless transmission protocol such as UDP, which can even lead to reordering of messages from the same replica during transmission. Though there is usually a well-defined order in which the operations happen on a replica, there is no total global order of all operations issued.

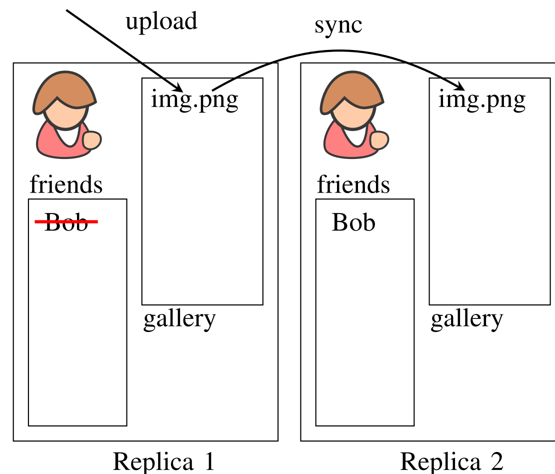


Fig. 1. Social Network Example.

In Figure 1, we illustrate this problem with a concrete example. Consider a social network where users can create galleries for uploaded pictures and share them with the community. By default, the system denies access to a user's personal page to arbitrary users. Only users that have been added to a friend

list have full access to this user's galleries.

Assume that Alice has an existing list of friends, one of them being Bob. After a heated discussion with him, Alice does not want Bob to have access to her most recent photos. She therefore removes Bob from her friend list before uploading her new photos of the last party.

When Alice interacts with the system through replica 1, the remove operation of Bob is issued before the upload of the new photos. In a system with strong consistency guarantees, it would not be possible for Bob to access the new photos on any replica unless he gets re-added to the friend list of Alice. In a replicated system with weak consistency guarantees, however, there might be some replica 2 that receives the upload of the photos before the update of the friend list. This gives Bob the possibility to access the new photos before the change to the friend list of Alice is known to replica 2.

As the example shows, receiving the policy update and the data modification out-of-order in reversed order can lead to leakage of sensitive information. If the policy modification restricts the visibility of the subsequent data modification, the permutation of the operations leaves the effect of the data modification temporarily unprotected. In the case above, Bob may be able to access the personal information of Alice before the policy modification is eventually received and restricts the access. In order for the policy to be correctly implemented in the access control system, the order between policy changes and subsequent data operations need to be retained on all replicas.

Contributions

In summary, we present a model for access control in weakly consistent cloud-storage systems. In this setting, all replicas are trusted and operate in a secure environment, but exchange updates only asynchronously between them. Our contributions are as follows:

- We give an abstract model for weakly consistent cloud-storage systems with access control systems (Section II).
- We provide a notion of correctness for access control in weakly consistent cloud-storage systems (Section III).
- We propose an access control system for the presented setting and highlight its applicability and flexibility by instantiating it for typical application scenarios (Section IV).
- Finally, we sketch how to efficiently implementation the access control model in state-of-the-art cloud-storage systems (Section V).

II. Formal system model

To discuss the notion of correctness for access control systems in weakly consistent data stores, we start with a formal model of the system setting.

The basis for the model is a distributed weakly consistent *data store*. We refer to the dataset subject to replication as the *data state*. The data state consists of *objects*, $o \in Objects$. The operations modifying objects are called *data operations*, $op \in Op$. Each data operation has a target object, $\mathsf{target}(op) \in Objects$. The set of operations can be divided into disjoint sets of read-only operations Op_R and modifying operations Op_W. Typically, at least one of these modifying operations allows to create new objects. We associate each operation op with the subject s who issued the operation, using a tuple (op, s). The every objects is created in an initial empty state and evolves by applying data operations.

The system consists of a (fixed) number of nodes hosting the data store. The system can tolerate downtimes of replicas as long as the network partitioning is not permanent (crash recovery model). The communication between the replicas is peer-to-peer. Messages can propagate through peer nodes and the nodes do not require point-to-point connections. To simplify the model, we assume full replication which means that every object is replicated at each node. We relate the nodes therefore to the replica and restrict ourselves to the notion of replica in the following.

Updates can originate at any replica. We call these user-driven operations *upstream operations*. Replicas synchronize by exchanging messages about their local data operations. We call the operations carried by these synchronization messages *downstream operations*. Only update operations are exchanged between the replicas; read-only operations are local to the replica where the operation was issued. We further assume that synchronization messages are eventually delivered to each replica. The synchronization messages carry additional information about the user that issued the operation on the origin replica.

Downstream operations from other replicas are received at a replica in an arbitrary order. We assume that the data state is eventually consistent meaning that the data state eventually converges to a common value when the system receives no new updates. For example, data stores providing conflict-free replicated data types [3, 18, 20] provide guarantees that conflicts in updates are resolved identically at every replica.

We expect that the data store has support for causality tracking [4], even though the data state itself does not need to be causally consistent [13, 17]. At each replica, all operations, upstream and downstream, are assumed to be processed at each replica in a serialized order. Analogous to Lamport's notion of causality [13], we define $a \rightarrow b$ to mean operation a happened before operation b on some replica R. This means that operation a was visible on replica R at the time when operation b was executed on R. Let $C_R\langle a\rangle$ denote the local time at which operation a was executed on replica R. We make no assumption about the relation of $C_R\langle a\rangle$ to the (global) physical time. But we assume that if $a \rightarrow b$ than $C_R\langle a\rangle < C_R\langle b\rangle$.

Policies and trust

Each data store operates in an environment consisting of the organization operating the data store, the physical servers running the store, the building in which the servers are located and so on. The organization needs to define rules which restrict the operations that may be performed on the objects by the individual subjects. In a concrete system, the subjects can be

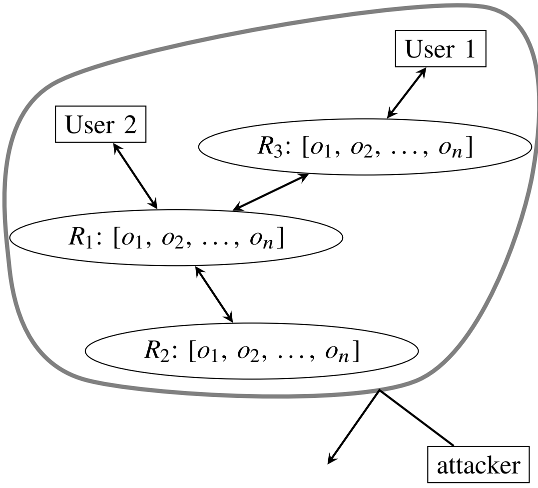


Fig. 2. System setting: A firewall protects the message exchange between the replicas.

users, processes or applications executing operations on the data store. A security policy assigns to each subject-object-pair a specific right. We write such a rights assignment as a triple $(r, s, o) \in \mathit{Rights} \times \mathit{Subjects} \times \mathit{Objects}$. A right allows to execute a set of operations. In the social network example, we have seen the right $friend$ that allows to access the pictures in the gallery of a user. The interpretation of a right is given by the $\vdash$-relation where $r \vdash op$ means that operation op is allowed by right r. A rights assignment (r, s, o) *permits* an operation $(r, s, o) \models (op, s')$ iff $r \vdash op$, $\mathsf{target}(op) = o$ and $s = s'$. Each replica has a representation of the security policy called the *policy state*. We give further details of how the policy state is modeled for typical settings in Section IV. The policy state of a replica is updated using *policy modifying operations* which update the rights assignments. As with data operations, these modifications of rights assignments are synchronized with the other replicas using synchronization messages.

We make assumptions about the trust between the replicas of the data store: The upstream operations are checked by a access control system for adherence to the security policy. The downstream operations are applied without checking their correctness. This assumes that all replicas are trusted to check the upstream operations and that the network connection is secure. Our assumption matches the properties of cloud-storage systems as in this setting all replicas are operated by the same (trusted) organization in a controlled environment. Communication between the replicas is usually secured using encryption technology and additional physical access restrictions are in place to guarantee the integrity of the replicas.

Fig. 2 illustrates our expected system environment. The replicas R_1 to R_3 can freely communicate in a peer-to-peer fashion. Attackers cannot directly access the replicas nor forge message that are exchanged between the nodes. Trusted users log into the system and temporarily become part of the protected environment. When they log out, they leave the protected environment again.

III. Correctness of Access Control in Weakly Consistent Systems

Informally, an access control system is correct if it enforces the security policy on all replicas. For strongly consistent systems, this means that an operation op_D performed by subject s should only be allowed by the access control system if the current rights assignment of the subject entails it, that is $(r, s, o) \models (op, s)$. Since the security policy is subject to modifications, the permission to execute operation op implies that the corresponding rights assignment happened before op: $(r, s, o) \rightarrow (op, s)$. We do not only want to permit new operations, but we also want to restrict the visibility of objects. If a rights assignment does not permit a read operation, $(r, s, o) \not\models (op_R, s)$, then after applying the policy change the operation should not happen, that is, $(r, s, o) \not\rightarrow (op_R, s)$. To enable the read operation again, the policy has to be changed accordingly. This means that a new rights assignment (r', s, o) is installed such that $(r', s, o) \models (op_R, s)$, which allows that $(r, s, o) \overset{!}{\rightarrow} (r', s, o) \rightarrow (op_R, s)$.

The strong consistency of the system makes it possible to talk about *the* current security policy. Strong consistency induces a global order of all operations that occur in the system. To find the current policy that applies to an operation op, we take the initial policy and apply all policy modifications that happened before op.

For weakly consistent systems, there is not such a total order between all operations. At each replica, there is a total order in which operations are applied, but this order can vary for different replicas as explained above. Without enforcing additional and costly synchronization between the nodes, we can therefore only refer to the local copy of the security policy represented by the policy state of the local replica.

Data operations may depend on policy modifications in two different ways:

1) A data operation is valid because a change in the policy permits it:

$$(r, s, o) \rightarrow (op_W, s) \text{ and } (r, s, o) \models (op_W, s)$$

2) A policy modification protects the modification of the state of an object by revoking access for specific subject:

$$(r, s_1, o) \rightarrow (op_W, s_2) \text{ and } (r, s_1, o) \not\models (op_R, s_1)$$

The first case only concerns the consistency between the local security policy and the data state. Wobber et al. [19] have discovered this relation in the implementation of their access control system for weakly consistent replication. Their implementation blocks a data operation on a replica until the respective policy update arrives. In contrast, our system model trusts the replicas to check the consistency of operations and updates to the security policy. The second case is more interesting since it can lead to leakage of information as seen in the social network example in the introduction.

We sketch the situation in Fig. 3. Let us assume that the current rights assignment of the local security policy of replica R_2 grants the read access to object o to s_1, $(r', s_1, o) \models (op_R, s_1)$.

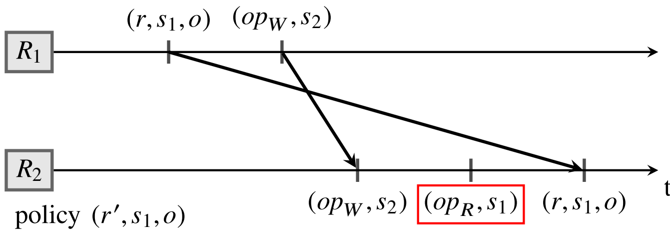


Fig. 3. Scenario leading to a policy violation.

Now, an authorized subject modifies the security policy on replica R_1 to (r, s_1, o). Afterwards, subject s_2 issues a write operation to the object.

If R_2 receives (op_W, s_2) out-of-order, we can have the situation that subject s_2 meanwhile tries to read object o with $(op_W, s_2, o) \rightarrow (op_R, s_1)$, thus observing the updates issued by s_2. The current security policy on R_2 grants the right to s_1 for reading the object. This is a violation of the policy enforcement and thus should be considered incorrect in an access control system.

In the social network example given in the introduction, a similar policy modification by removing Bob from the list of friends protected the upload of further pictures to the gallery. If the order between the policy modification and the upload is not preserved on the other replicas, the uploaded pictures can be leaked to Bob even though the policy prohibits access of Bob.

Protection relation

For an access control system for distributed weakly consistent systems to be correct, operations that depend on policy changes globally need to be applied after the policy modifications that possibly restrict the access to the object. A rights assignment (r, s_1, o) is in *protection relation* with a modifying operation (op_W, s_2) if $\mathsf{target}(op_W) = o$. We denote the protection relation as $(r, s_1, o) \lhd (op_W, s_2)$. The relation covers all policy modifications (r, s, o) targeting the same object o as a modifying operation op_W, which is an over-approximation since only policy modifications restricting the access to o need to be covered.

A distributed access control systems retains the protection relation if for all modifying operations op_W with $\mathsf{target}(op_W) = o$, it holds that all policy modifications (r, s, o) that happened before are applied on all replicas before applying op_W:

$$(r, s_1, o) \rightarrow (op_W, s_2) \land \mathsf{target}(op_W) = o \implies C_i\langle(r, s_1, o)\rangle < C_i\langle(op_W, s_2)\rangle$$

The protection relation can be violated by synchronizing a concurrent policy modification. The situation is sketched in Fig. 4. We assume that $(r, s_1, o) \lhd (op_W, s_2)$ and that $(r', s_1, o) \models (op_R, s_1)$. The rights assignments (r, s_1, o) and (r', s_1, o) happen concurrently on replicas R_1 and R_3. The order in which these two policy modifications are applied on R_2 is arbitrary, which means that (r', s_1, o) could be applied after (r, s_1, o) and before the protected modifying operation (op_W, s_2). In this case, we have that $(r, s_1, o) \rightarrow (op_W, s_2) \rightarrow (op_R, s_1)$ which violates that protective property of the security policy. The problem in this case is the concurrent modification of the policy. The integration of this policy also needs to make sure that the protection relation stays intact.

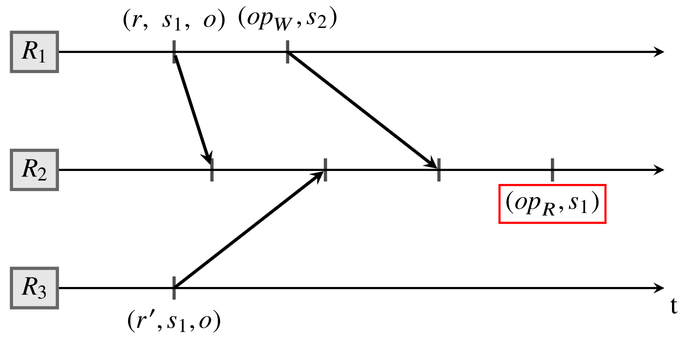


Fig. 4. Concurrent right assignment violating the protection relation.

To implement a correct access control system, all replicas have to retain the protection relation and make sure that integrating concurrent policy modifications do not violate it.

IV. Protection-preserving Access Control

In this section we describe ProPreAC, our protection-preserving access control model. The model has to support two major requirements:

1) it must support local modifications at the replica that are eventually propagated and reflected in the policies of all other replicas;
2) the protection relation has to be retained.

The first point means that it should be possible to change a rights assignment visible on the local replica thereby updating the policy representation of remote replicas through the system's synchronization mechanism. The second point deals with the problem of concurrent policy modification that have to be integrated in such a way that they do not violate the protection relation.

The policy state P consist of rights-assignment triples (r, s, o). For each subject-object-pair there may be multiple such rights-assignments, one for each concurrently issued policy modification. We define the set

$$P\mid_{s,o} = \{(r, s', o') \mid (r, s', o') \in P \land s' = s \land o' = o\}$$

to be the set of all concurrent rights-assignments for subject s on object o. We assume that each policy modification can be uniquely identified.

To correctly implement a policy model that retains the protection relation, we need to consider the causality relation [13, 17] between policy modifications. We therefore need a strategy how to integrate downstream policy changes into the local copy of the security policy without violating the protection relation. Downstream policy updates are given in

form of a rights assignment, $p_{down} = (r,s,o)$. When applying it locally, it replaces all rights assignments (r',s,o) in the receiver replica that target the same object and subject and happened before, i.e. $(r',s,o) \rightarrow (r,s,o)$.

$$P\mid^{new}_{s,o} = \{p \mid p \in P\mid^{old}_{s,o} \cup \{p_{down}\} : \neg\exists p' \in P\mid^{old}_{s,o} \cup \{p_{down}\} : p \rightarrow p'\}$$

The new downstream policy is added to the set of policies for subject s and object o. The policies which have been overwritten are removed from this set. This construction makes sure that concurrent policy modifications are tracked separately and that no such modification is lost. Since rights assignments that were visible on a replica are replaced by policy modifications that happen afterwards, policy updates propagate through the system without being lost.

To prevent violations of the protection relation, we require that the set of rights *Rights* forms a lattice. For two rights r_1 and r_2, $r_1 < r_2$ means that r_1 grants less operations than r_2, that is, r_1 is stricter than r_2. To deduce the policy for subject s and object o determined by a security policy $P\mid_{s,o}$, the minimal right is a safe interpretation of the granted rights:

$$\min(P_{s,o}) = (\min(\mathsf{rights}(P_{s,o})), s, o)$$

where $\mathsf{rights}(P) = \{r \mid (r,s,o) \in P\}$.

If we have two concurrent policy modification (r',s_1,o) and (r,s_1,o) with $(r,s_1,o) \lhd (op_W,s_2)$ and $(r,s_1,o) \not\models (op_R,s_1)$, then $\min(\{(r,s_1,o),(r',s_1,o)\}) \not\models (op_R,s_1)$. This guarantees that $(op_W,s_2) \not\rightarrow (op_R,s_1)$. In that way, the above construction guarantees that downgrades of permissions do not get lost and therefore the protection relation is preserved under concurrent policy modifications.

In addition, the given construction supports that policy updates can be applied out-of-order since the construction does not depend on causal consistency of the rights assignments. Modifying operations only have to wait for their accompanying policy modifications on the same object to arrive. In contrast to previous systems, our system does not only wait for the policy modifications enabling the modifying operation, but also for the policy changes protecting the data operation.

To illustrate the formal model, we will discuss now how to instantiate ProPreAC for a classical system with hierarchical read-write-own policies and a fine-grained operation-based access control.

Consider a system with three subjects: Alice, Bob and Eve. In this example, the subjects are actual users of the system. Alice and Bob are special admin users with the special right to modify the security policy, which is tracked outside of the access control system. The data store maintains as stored objects maps which associate keys with values. Maps can be modified by assigning a value v to a key k in map m (operation $\mathtt{put}(m,k,v)$). Operation $\mathtt{get}(m)$ returns all key-value assignments of map m. Deleting a map from the data store is supported by the $\mathtt{del}(m)$ operation, which removes all assignments from map m. The target object is for each operation, respectively, defined as:

$$\begin{aligned} \mathsf{target}(\mathtt{get}(m)) &= m \\ \mathsf{target}(\mathtt{put}(m,k,v)) &= m \\ \mathsf{target}(\mathtt{del}(m)) &= m \end{aligned}$$

Further, we assume that the data store ensures eventual consistency of the map objects under concurrent modification (e.g. by implementing Map CRDTs [3]). For the sake of the example, we assume that when issuing two `put` operations on distinct keys k_1 and k_2 concurrently, both assignments are retained.

Hierarchical Read-Write-Own Model

A classical policy model in access control systems is the read-write-own hierarchy. A user which has read access to an object can read the object. The write access also grants the permission to read the object, and in addition the user can also modify the object. The own access grants the same rights as the write access and in addition allows to delete the object. This system is based on a total order between the different access rights $Rights = \{none, read, write, own\}$ where $none < read < write < own$.

The rights entail the operations $\vdash$ as follows for some map m:

$read \vdash \mathtt{get}(m)$

$write \vdash \mathtt{get}(m),\ write \vdash \mathtt{put}(m,\ k,\ v)$

$own \vdash \mathtt{get}(m),\ own \vdash \mathtt{put}(m,\ k,\ v),\ own \vdash \mathtt{del}(m)$

Because the lattice of rights builds on a total order, we call this model a hierarchical read-write-own model.

Let us assume that users Alice and Bob have write access to map m, whereas Eve has only read access to m. In addition, Alice and Bob have the permission to assign new permissions for map m, and may thus modify the security policy. Consider now two replicas R_1 and R_2, both converged to the same state with an identical security policy:

$$\begin{aligned} P\mid^{old}_{\text{Alice},m} &= \{(write, \text{Alice}, m)\} \\ P\mid^{old}_{\text{Bob},m} &= \{(write, \text{Bob}, m)\} \\ P\mid^{old}_{\text{Eve},m} &= \{(read, \text{Eve}, m)\} \end{aligned}$$

Now, the system state evolves as follows: On R_1, Alice first sets the permission of Eve for m to *none*, because she thinks that Eve is not trustworthy, and afterwards writes a value v_1 to key k_1 in map m which Eve should not read. In parallel, Bob first sets the permission of Eve for m to *write*, because he thinks that Eve is trustworthy, and shares a value v_2 with Eve by writing it to key k_2 in m. Fig. 5 illustrates the situation.

After forwarding the data state and the policy modifications to the respective other replica, the common converged state looks like this: The data state for map m is extended by the two write operations $\mathtt{put}(m,k_1,v_1)$ and $\mathtt{put}(m,k_2,v_2)$. When reading from m, both the assignment to k_1 and the assignment to k_2 are visible. The policy state is updated by both rights assignments, from R_1 and from R_2, thereby

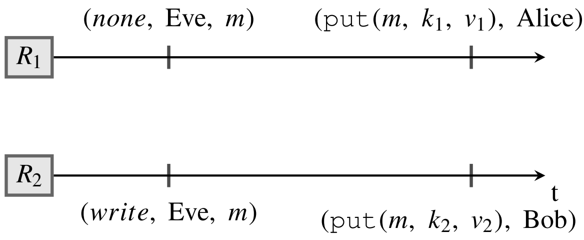


Fig. 5. Read-Write-Own Parallel Policy Change.

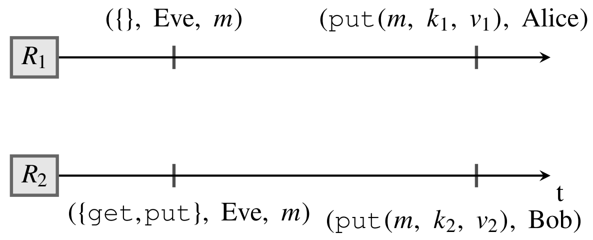


Fig. 6. Operation-based concurrent policy modifications.

overwriting previous rights assignments. Since these policy modification happened in parallel, the policy state is now

$$P\,|^{new}_{\mathrm{Eve},m} = \{(none, \mathrm{Eve}, m),\ (write, \mathrm{Eve}, m)\}$$

When asked for the current policy for Eve, the system computes the minimum of rights assigned to Eve, which in this case is the policy $(none, \mathrm{Eve}, m)$. In this state, the operation $\mathtt{get}(m)$ is not allowed to be performed by Eve since $(none, \mathrm{Eve}, m) \not\models (\mathtt{get}(m), \mathrm{Eve})$. This is consistent with the fact that the assignment (k_1, v_1) should not be readable by Eve because of the protection relation $(none, \mathrm{Eve},\ m) \lhd (\mathtt{put}(m, k_1, v_1), \mathrm{Alice})$ induced by the operations on R_1.

Fine-grained operation-based Model

ProPreAC supports also the implementation of a fine-grained operation-based model. In contrast to the read-write-own model presented in the previous paragraph, this operation-based model allows more control over which operation is permitted for each user. If multiple modifying operations are available for a data type, like setting the value of an integer object and incrementing the value of the object, each of the operations can be granted and revoked individually in such a system. As the set *Rights*, we use sets of operation identifiers $Rights = 2^{\{\mathtt{get},\mathtt{put},\mathtt{del}\}}$ and base the lattice on the subset relation instead of totally ordering the rights. Under this model, a user maybe able to read and delete an object, but not write it. In this way, we have more fine-grained control over which operations are allowed for each user.

For the previous example, this means that the policies for the users Alice, Bob and Eve in the converged state of the replicas R_1 and R_2 looks like this:

$$P\,|^{old}_{\mathrm{Alice},m} = \{(\{\mathtt{get},\mathtt{put}\}, \mathrm{Alice}, m)\}$$
$$P\,|^{old}_{\mathrm{Bob},m} = \{(\{\mathtt{get},\mathtt{put}\}, \mathrm{Bob}, m)\}$$
$$P\,|^{old}_{\mathrm{Eve},m} = \{(\{\mathtt{get}\}, \mathrm{Eve}, m)\}$$

Fig. 6 shows again the operations that happen concurrently on replicas R_1 and R_2. Alice assigns the rights for Eve to the empty set, revoking all permissions she previously had, before storing v_1 into the map at key k_1. This policy modification protects the subsequent write

$$(\{\}, \mathrm{Eve}, m) \lhd (\mathtt{put}(, k_1, v_1), \mathrm{Alice}).$$

Bob, on the other hand, gives full read- and write-access to the map by assigning the permission to perform get- and put-operations on the map. The resulting policy for Eve on m after synchronizing the states is

$$P\,|^{new}_{\mathrm{Eve},m} = \{(\{\}, \mathrm{Eve}, m),\ (\{\mathtt{get},\mathtt{put}\}, \mathrm{Eve}, m)\}.$$

Computing the minimum of rights granted to Eve is done by intersect of all right sets $\min(rs) = \bigcap rs$. In this case, the result is the empty set $\{\}$, meaning Eve has no permission to access the map m, which preserves as required the protection relation

$$(\{\}, \mathrm{Eve}, m) \lhd (\mathtt{put}(m, k_1, v_1), \mathrm{Alice}).$$

Operations for policy modifications

Up to this point, we have only considered data operations and the policy restricting these operations. The model can easily be modified to support also operations that modify the policy. For example, many systems have a `grant`-operation, which enables a user to give other users the permission to access an object. To this end, we extend the set of operations to also include policy modifying operations:

$$Op = Op_R \uplus Op_W \uplus Op_{policy}$$

The policy modifying operations Op_{policy} are akin to data operations issued locally on some replica. The effect of such an operation $op_p \in Op_{policy}$ is a policy modification $\mathsf{effect}(op_p) = (r, s, o) \in Rights \times Subjects \times Objects$, which is synchronized with the other replicas. As before, the upstream operations are checked the same way data operations are checked against the local security policy.

Let us consider the `grant`-operation in more detail. It typically has three parameters, $\mathtt{grant}(s, o, perm)$, where $s \in Subjects$, $o \in Objects$ and $perm \in Rights$. Adding the `grant`-operation to the read-write-own system works by extending the $\vdash$-relation with:

$$own \vdash \mathtt{grant}(s, o, perm)$$

The target of the `grant`-operation is the object given as parameter $\mathsf{target}(\mathtt{grant}(s, o, perm)) = o$. The effect of the operation $\mathsf{effect}(\mathtt{grant}(s, o, perm)) = (perm, s, o)$ is a rights assignment, not a data state modification as for the data operations. For Bob, to grant Eve the write permission on the map m, he would need the *own* permission for m and execute a $\mathtt{grant}(\mathrm{Eve},\ m,\ write)$ operation. The corresponding policy change $(write, \mathrm{Eve}, m)$ is synchronized with the other replicas.

This scheme allows to store the complete security policy in the data store and to user the access control system to enforce

the complete policy. Otherwise, the permissions to modify the policy would have to be tracked and enforced by a separate system.

V. Implementation

In this section, we sketch how our model ProPreAC can be efficiently implemented in state-of-the-art weakly consistent storage systems. The base for our implementation is an eventually consistent data store.

As discussed in Section IV, the model requires tracking of the causality between two policy modifications as well as between policy modifications and data operations. Causality tracking as implemented in Amazon Dynamo [6] using vector clocks is known to not scale very well. The problem is that the meta-data needed to track causality information grows with the number of clients or it grows only with the number of servers but is then less accurate. Tracking the information per server does not allow to express concurrent or causally related operations on the same server which can lead to false concurrency information regarding operations on the same server. Almeida et al. [4] propose a more efficient and accurate solution using dotted version vectors. This technology allows to track causality of operations accurately by combining server-based version vectors with more detailed tracking of individual operations. Gonalves et al. [9] presents an implementation of this technology which is not only efficient and accurate, but also has low overhead in the distribution of the causality information.

For the implementation of ProPreAC, we have to integrate an access control check into the processing layer of upstream operations. These operations have to be directly checked against the current local security policy represented by the policy state of the replica. The local policy for an operation (op_D, u) is computed by taking the set of rights-assignments for the subject-object-pair $P\mid_{s,\mathsf{target}(op_D)}$ and computing the minimum rights as described in Section IV.

The downstream processing layer has to wait when receiving a downstream modifying operation (op_W, s), until all policy modifications in the protection relation $(r,s,o) \lhd (op_W, s)$ have been received. This can be efficiently implemented based on the information about the causality between policy changes and data operations. The processing of (op_W, s) has to be postponed until all rights assignments (r,s,o) have been received where $\mathsf{target}(op_W) = o$ and $(r,s,o) \rightarrow (op_W, s)$. Concurrent policy changes can be applied in any order without harming integrity. Similarly, operations on different objects can be applied concurrently if the data store supports eventual consistency.

We are currently working on an implementation of the model based on the Riak key-value store [3]. Dotted version vectors and CRDTs are implemented in Riak [3] since version 2.0., thus showing the feasibility of the technique in large scale real-world applications and systems.

VI. Related Work

In the area of access control for weakly consistent systems, there are two major tracks of related work: weakly consistent data stores and collaborative editors.

Eventually consistent data stores: With the implementation of cloud systems, weakly consistent data stores have moved into the focus of distributed systems research. Though many protocols and implementation schemes have been proposed and implemented, the topic of access control for these systems has received surprisingly little attention. The original version of Amazon Dynamo [6] did not offer authentication and authorization capabilities. Several other related eventually consistent data stores offer meanwhile techniques to implement access control, but the granularity is not fine enough to provide access control on the application level. Riak KV [3], MongoDB [2] and Couchbase [1] all support the management of users, roles and permissions. But the smallest granularity is on the level of buckets or collections, comparable with tables in relational data bases. Typical permissions on this level allow to read, write, modify, or delete any value of the bucket or collection. A more fine-grained permission level relating to the operations on the application level is not supported.

Regarding eventually consistent data stores, Samarati et al. [15] describe a high-level approach to authorization. The general idea is to optimistically accept all operations and compensate the operations which were executed despite the security policy by performing rollbacks. While this approach guarantees convergence of the security policy, it is not clear for each operation how to undo the effect of this operation after it has been executed. One of the problems is the potential binding between operations and effects in the data store and changes of the real world. For example, a banking system allows to withdraw money from an account and the ATM outputs the money. In this case, it is hard to undo the withdrawal because the person with the money has already walked away. In addition, the guarantees given by such an optimistic system remain unclear. Effects of operations can be perceived by a user of the data store before the rollback, thereby possibly leaking sensible information.

Wobber et al. [19] present an access control model for weakly consistent mutually distrustful replicated systems. Their focus work is on partial replication with different access policies per replica. While we consider a different setting of cloud-storage systems, similar problems can be identified. The causality between a policy and the subsequent operation that are permitted by the policy is captured by their model by waiting for the required policy change to arrive. However, the causality between a policy change that restricts the visibility of the effect of an operation and the subsequent execution of this operation is not captured. As such, the model still allows leaking sensitive information because of the possible violation of the protection relation between a policy change and a subsequent data operation.

Collaborative Editors: Imine et al. [10] present an access control model for distributed collaborative editors. The similarities to data stores lies in the fact that the modifications of the document in the editor can be seen as data operations and the document needs to be eventually consistent on all distributed editor instances. As such, the editors can be seen

as replicas managing the document as the data state. In this setting, the policy modifications consider objects such as a character or section of text. To prevent divergence, all operations on an object are ordered by the authoring editor instance. This implies a one-to-one relation between parts of the text and the responsible editor instance, which again leads to a single point of failure. A addition, the model only considers modifying operations; all users are allowed to read the complete document. The implementation by Imine et al. is very similar to an implementation of causality tracking for a data store. In contrast, our model considers read operations, which allows us to talk about leakage of information because the policy can restrict the read access, thereby protecting data modifications. Further, we introduce neither bottlenecks nor single-points-of-failure by building on the causality relation.

VII. Conclusion

We have introduced ProPreAC, an access control system for eventually consistent cloud storage. To the best of our knowledge, it is the first system that considers a causal relation between policy modifications and subsequent data operations to prevent information leakage and unauthorized data modification.

We presented a formal model of access control in weakly consistent replicated systems and formulated a definition of the correctness for access control in such a system. The definition is strongly based on the protection relation between a policy modification which restricts access to an object and a subsequent data operation modifying the object. ProPreAC preserves this protection relation, thereby guaranteeing correct enforcement of the security policy. We showed how to instantiate our system with different policy schemes such a hierarchical read-write-own policy schema, thus illustrating that the model is flexible and applicable for different schemes. Finally, we gave a sketch of how an efficient implementation of the access control model in state-of-the-art cloud-storage systems like Riak [3] can be achieved. In contrast to currently employed access control systems, ProPreAC supports high availability, does not hinder scalability and allows to implement fine-granular and flexible security policies as required by modern information systems.

References

[1] Couchbase. URL http://www.couchbase.com/.

[2] MongoDB for GIANT Ideas | MongoDB. URL https://www.mongodb.org/.

[3] Riak KV. URL http://basho.com/products/riak-kv/.

[4] P. S. Almeida, C. Baquero, R. Gonalves, N. M. Preguia, and V. Fonte. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In K. Magoutis and P. Pietzuch, editors, *Distributed Applications and Interoperable Systems - 14th {IFIP} {WG} 6.1 International Conference, {DAIS} 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014*, volume 8460 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2014. URL http://dx.doi.org/10.1007/978-3-662-43352-2_6.

[5] E. A. Brewer. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 7–, New York, NY, USA, 2000. ACM. ISBN 1-58113-183-6. URL http://doi.acm.org/10.1145/343477.343502.

[6] 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 Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. URL http://doi.acm.org/10.1145/1294261.1294281.

[7] D. Ferraiolo and R. Kuhn. Role-Based Access Control. In *In 15th NIST-NCSC National Computer Security Conference*, pages 554–563, 1992.

[8] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51, 2002. ISSN 01635700. URL http://doi.acm.org/10.1145/564585.564601.

[9] R. Gonalves, P. S. Almeida, C. Baquero, and V. Fonte. Concise Server-Wide Causality Management for Eventually Consistent Data Stores. In A. Bessani and S. Bouchenak, editors, *Distributed Applications and Interoperable Systems*, number 9038 in Lecture Notes in Computer Science, pages 66–79. Springer International Publishing, June 2015. ISBN 978-3-319-19128-7 978-3-319-19129-4. URL http://link.springer.com/chapter/10.1007/978-3-319-19129-4_6.

[10] A. Imine, A. Cherif, and M. Rusinowitch. A Flexible Access Control Model for Distributed Collaborative Editors. In *Secure Data Management, 6th VLDB Workshop, SDM 2009, Lyon, France, August 28, 2009. Proceedings*, pages 89–106, 2009. URL http://dx.doi.org/10.1007/978-3-642-04219-5_6.

[11] X. Jin, R. Krishnan, and R. S. Sandhu. A Unified Attribute-Based Access Control Model Covering DAC, MAC and RBAC. *DBSec*, 12:41–55, 2012. URL http://link.springer.com/content/pdf/10.1007/978-3-642-31540-4.pdf#page=52.

[12] X. Jin, R. Sandhu, and R. Krishnan. RABAC: role-centric attribute-based access control. In *Computer Network Security*, pages 84–96. Springer, 2012. URL http://link.springer.com/chapter/10.1007/978-3-642-33704-8_8.

[13] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. URL http://doi.acm.org/10.1145/359545.359563.

[14] P. Samarati and S. D. C. d. Vimercati. Access Control: Policies, Models, and Mechanisms. In *FOSAD*, pages 137–196, 2000.

[15] P. Samarati, P. Ammann, and S. Jajodia. Maintaining

Replicated Authorizations in Distributed Database Systems. *Data Knowl. Eng.*, 18(1):55–84, 1996. URL http://dx.doi.org/10.1016/0169-023X(95)00000-I.

[16] G. Saunders, M. Hitchens, and V. Varadharajan. An Analysis of Access Control Models. In *ACISP*, pages 281–293, 1999.

[17] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the holy grail. *Distrib Comput*, 7(3):149–174, Mar. 1994. ISSN 0178-2770, 1432-0452. URL http://link.springer.com/article/10.1007/BF02277859.

[18] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free Replicated Data Types. In *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, SSS'11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-24549-7. URL http://dl.acm.org/citation.cfm?id=2050613.2050642.

[19] T. Wobber, T. L. Rodeheffer, and D. B. Terry. Policy-based access control for weakly consistent replication. In *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*, pages 293–306, 2010. URL http://doi.acm.org/10.1145/1755913.1755943.

[20] M. Zawirski, N. Preguiça, S. Duarte, A. Bieniusa, V. Balegas, and M. Shapiro. Write Fast, Read in the Past: Causal Consistency for Client-side Applications. In *Proceedings of the Annual ACM/IFIP/USENIX Middleware Conference*, Vancouver, BC, Canada, Dec. 2015. ACM/IFIP/Usenix, ACM.

7.2 Christopher Meiklejohn. Loquat: A partially replicated, secure, broadcast protocol for edge computation. Submitted for publication, 2016.

Loquat: A Partially Replicated, Secure, Broadcast Protocol for Edge Computation

Christopher Meiklejohn
Université catholique de Louvain
Louvain-la-Neuve, Belgium
Email: christopher.meiklejohn@gmail.com

***Abstract*—We provide a work-in-progress report on a partially replicated, secure, broadcast protocol for edge computation. This protocol, called Loquat, is inspired by the authors previous work on edge computation; specifically a gossip-based runtime system for a distributed, declarative, programming model for edge computation, called Lasp. This protocol is designed to scale to a large number of clients while providing both partial replication of data, and secure transmission of confidential information, both extremely useful properties of multi-tenant sensor and edge networks. We motivate our work with an industrial use case where security and efficiency in data dissemination is paramount.**

I. Introduction

Efficient large-scale computation continues to grow in importance as the reach of mobile Internet applications increases, and new and different types of devices are connected to the Internet to support "Internet of Things" applications such as sensor networks and smart cities. Moving computation to the edge is becoming extremely important as the amount of data generated by the edge make traditional approaches that centralize computation around the data center intractable.

A. Motivating Example

We consider the industry use case of a "Internet of Things" (IoT) service provider that builds sensors for monitoring the temperature of refrigerators deployed at hospitals in the San Francisco Bay Area [1]. In this application, refrigerators store vital organs and tissue samples that make it critical that refrigerators are monitored for temperature anomalies and that alerts triggered when the temperature is outside of a specified range.

In Figure 1, we depict the design most frequently seen deployed in industry. In this design, refrigerators transmit temperature samples to a central location for processing. More specifically, in this example we use HDFS[1] for aggregation of data in the data center. Providers typically choose one of two routes for processing incoming samples: either a batch processing system, such as Hadoop [2] is used to repeatedly process received samples, or a stream processing system such as Spark (D-Streams) [3] is used to react to samples in realtime. Figure 2 shows a successful execution in this model.

[1]The both commercially available, and open-source Apache Foundation Hadoop Distributed File System.

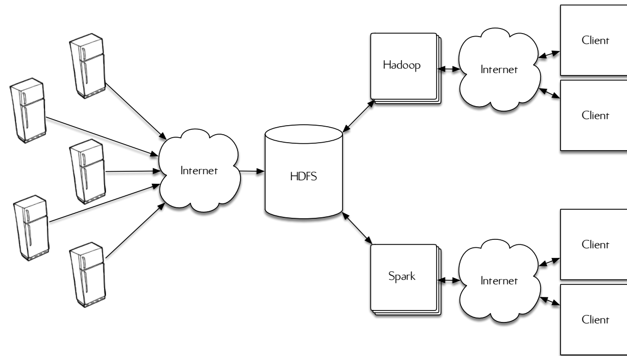


Figure 1: *Typical structure of IoT application. Data is aggregated from sensors at the edge to a central location for processing.*

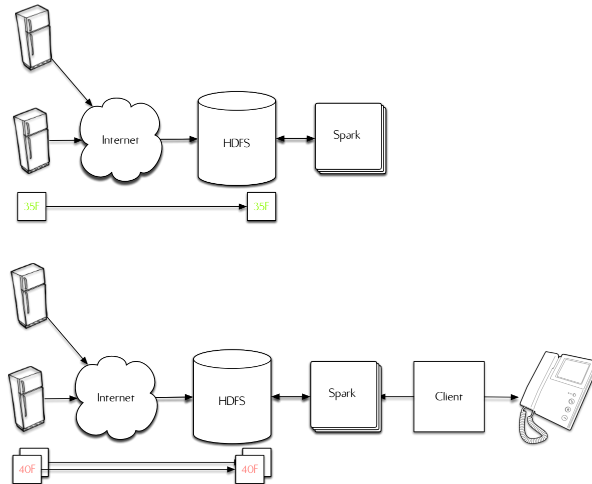


Figure 2: *Successful execution using the centralized model. Once a sample arrives that should trigger an alert, the alert is triggered.*

B. Designing for Uncertainty

This design is desirable to the developers of these applications. It allows them to develop algorithms and reports using a centralized processing model where the developer does not have to worry about the challenges of distributed computation:

specifically, the problems of unreliable asynchronous networks with unpredictable latency. However good this design may be for application developers, this solution to the problem is far from ideal as it involves a significant amount of state transmission and can not tolerate periods without connectivity to the central database where processing occurs.

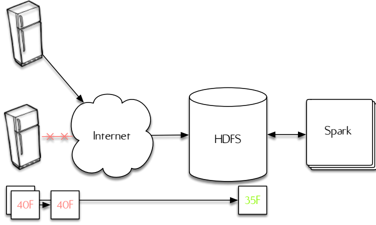


Figure 3: *Connectivity problems with the data center may prohibit the transmission of events that should trigger alerts.*

In this design, connectivity, or lack thereof, is problematic. If a sensor that is generating samples that should trigger an alert (such as our problematic $40°$ sample in Figure 2) does not have connectivity to the central processing location, no action can be taken and we risk the destruction of any items in the refrigerator because of overheating (or rather, not cooling enough). We see an example of this in Figure 3. This scenario is more common than it may seem: sensors are traditionally underpowered, operating on battery over long-range wireless networks. In many cases, operation of the antenna is the biggest power drain of the device, which we will see contributes to the problem of state transmission below, as well.

Therefore, to ensure that we can alert on events regardless of connectivity, the obvious solution is to co-locate hardware alongside each of the sensors, so each sensor can trigger an alert (possibly, via telephone) in the event that connectivity is unavailable. This approach has precedent set in the late 90s and early 2000s[2]. However, this approach is extremely expensive, because all "edge" devices need to have the means to alert, and the processing power to process their own events locally.

One modification of this approach previously proposed by Meiklejohn [4] suggested the use of transitive dissemination of events[3] to reduce the amount of hardware needed at the edge. In this design, there is a specific trade-off between the time-to-alert and the amount of hardware deployed at the edge for alerting.

To demonstrate this design, we refer the reader to Figure 4. In this example, a node might have connectivity to another node in the system, and through that node have transitive connectivity to the data center for dissemination of events. In this case, a gossip protocol [5], that supports a reliable peer-to-peer information dissemination model that is highly resilient to churn and network faults, seems to fit better than the alternative tree-based approaches which are fragile at scale[4].

[2]The author was previous involved in a telecommunications project where T1 lines were backed up by ISDN or 56k modems. EMC, a large manufacturer of storage hardware, previously used telephone lines attached to early NAS products for "phone home" behavior when CRC violations were detected on mounted volumes.

[3]No proceedings are available, but presented at OBT 2016, co-located with POPL 2016.

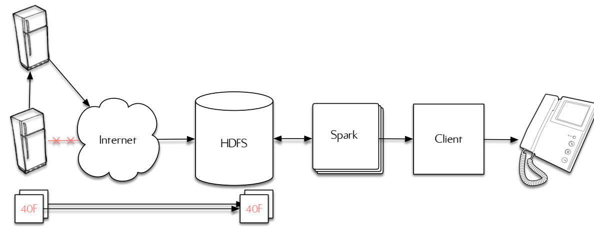


Figure 4: *Connectivity problems with the data center may prohibit the transmission of events that should trigger alerts. However, transitive dissemination through reachable nodes may mitigate connectivity problems.*

C. Drawbacks

Given the ideal design, we identify two remaining problems with the state of the art as it relates to gossip protocols and their deployment at the edge to enable the ideal design.

- **Partial replication.** None of gossip protocols surveyed [6], [7], [5] had support for partial replication of data items. Partial replication differs from traditional "full" replication, where each node in the system stores the same set of data as every other node participating in the cluster. In this design, nodes in the cluster can store potentially disjoint sets of data.
- **Security.** In the ideal design, some systems may not be privileged to transmit data on behalf of other nodes. For instance, the refrigerators used by the science department may not be authorized to transmit data for refrigerators used by the physics department. Therefore, a mechanism is required to ensure that transitive dissemination is only performed by authorized parties.

D. Contributions

The contributions of this work are as follows:

- **Partial replication.** We propose an extension to the Plumtree [7] epidemic broadcast protocol that supports partial replication of data. Our work is inspired by the authors previous work on the Selective Hearing [8] extension to the Lasp programming model [9] for distributed, declarative edge computation.
- **Information flow control.** We provide a work-in-progress design for an extension to this protocol that supports information flow control, where nodes that are not authorized to receive confidential information will not receive dissemination of that information.

[4]This discussion is continued in Section II.

II. Background

This section presents background material on the Plumtree protocol and Information Flow Control.

A. Plumtree

The Plumtree protocol is a hybrid approach to gossip protocols. The protocol combines the use of broadcast trees, for fast dissemination of information to a cluster of nodes, with the use of a epidemic broadcast protocol, for flooding the network with information to detect partitions in the network, or where disconnected nodes in the spanning tree have prevented messages from reaching all nodes in the cluster. This approach has been shown to be both more efficient and more resilient to network failures and churn [7], while being easier to debug than alternative hybrid protocols, such as Bimodal Multicast [6].

The Plumtree protocol operates by maintaining a spanning tree either per cluster, or per node, that contains some sample set of initial nodes[5]. When messages are broadcast to the spanning tree, or the "eager set", nodes that deliver a duplicate message, or a message previously received by another node, are pruned from the "eager set" and moved to the "lazy set". This results in the computation of a minimal spanning tree, represented by a cluster-wide, or node-local "eager set".

Nodes in the "eager set" should receive broadcast messages immediately, whereas nodes in the "lazy set" receive only broadcast messages containing a unique identifier of messages they should receive via the eager broadcast[6]. These messages are referred to as BCast messages and IHave messages, respectively.

In the event that one of these messages is not received within a given time interval through the eager broadcast mechanism, a network partition or failed node must have caused the spanning tree to fail to deliver a message. When this occurs, the following procedure is performed:

1) The receiving node that failed to receive the message that it saw advertised randomly selects one of the sending nodes that sent the IHave message.
2) The receiving node then sends a Graft message adding the sending node back to its "eager set".
3) The sending node then sends a BCast message containing the missing message.

B. Plumtree Optmizations

However, in practice, the Plumtree protocol as designed is not sufficient to guarantee reliable delivery to a client that may be offline for extended period of time (for instance, a long-lived data center partition, or a client that has gone offline for an extended period of time) [11]. Therefore, we consider two extensions to the Plumtree protocol, as originally conceived by Basho Technologies, Inc. in practice.

[5]The Plumtree protocol relies on an external peer sampling services, such as HyParView [10] by the same authors, for managing cluster membership.

[6]This process can happen on a per-message basis or in batches, depending on how the operator desires network utilization by control messages

1) In the original protocol, IHave messages, or broadcasts to the "lazy set", are only transmitted once at the time of the original broadcast. This results in a problem where a partitioned or offline client may never receive a broadcasted message if connectivity to the rest of the cluster is restored after these IHave messages have already been broadcast.
2) The delivery of these IHave messages may not reach all clients as the failure or churn rate of the cluster approaches 70% [7]. Therefore, it is necessary to augment this protocol with an anti-entropy protocol which periodically selects random peers and does a full state exchange, to ensure eventual delivery of all messages.

C. Information Flow Control

Our contributions extend the previous work from Myers and Liskov [12] on Decentralized Information Flow Control (DIFC) and the work on CamFlow from Pasquier *et al.* [13].

Decentralized IFC extends the seminal work on Information Flow Control (IFC) in 1976 by Denning [14] on a Mandatory Access Control mechanism for controlling the flow of information on computer systems based on security classes. IFC proposes two security classes for *secrecy* and *integrity* of entities and process in the computing system. Specifically, the *no read up, no write down* policy for *secrecy* from Bell and LaPadula [15] and the *no read down, no write up* policy for *integrity* from Biba [16]. IFC's policies were generic enough to model a traditional military classification system, where information moves between *public*, *secret*, and *top secret* classifications.

In 1997, Myers and Liskov's work on Decentralized IFC [12] moved the model away from a static, hierarchical model to a flexible, dynamic model based on *principals* and *labels*. *Principals* represent parties who own information, therefore their information privacy is protected, whereas *labels* are the way in which principals express privacy concerns over entities.

Labels take the form of a set of policies, a pair of an owner and a set of readers: $L = \{o_1 : r_1, r_2; o_2 : r_2, r_3\}$; in this example, one policy defines that o_1 is a principal that has only granted access to a given item protected by label L to readers r_1 and r_2. Given this object's label contains two policies, the only valid reader of this object is r_2, the intersection of the policy set as defined by owners o_1 and o_2. This is the idea that information can flow, or be released "by the consensus of all of the parties." [12] A similar idea exists for both the declassification and relabeling of information,; labels must be adjusted through an copy by a principal that has consensus of all of the parties. In this model, both principals and labels form a lattice, where information, and rights to declassify or relabel, can flow upwards.

Finally, the CamFlow model in 2015 by Pasquier *et al.* [13] further extends the work on Decentralized IFC to cloud environments and presents a slightly simpler formalism. In CamFlow, every entity in the system is associated with two *labels*, one for *secrecy* and one for *integrity*. Each labels is

formed by a set of tags, each tag being a user-defined token. Information flow is preserved in a similar way as the previous systems: *secrecy* follows the *no read up, no write down* policy and *integrity* follows the *no read down, no write up* policy.

To clarify the presentation, we include the example from the CamFlow model. Consider a medical patient Bob, being monitored from his home; information from his home would be tagged *bob*, *medical* for *secrecy* and information from his home would be *hospital-device* for *integrity*. Information received at the hospital would enforce the *integrity* policy by refusing any information that was from a device that is not tagged *hospital-device* and hospital devices attempting to read this data would have to have the proper tags for *secrecy*: *bob*, *medical* [13].

III. Loquat: Partial Replication

We provide an overview followed by the formal semantics of Loquat's partial replication algorithm.

A. Overview

Simply put, Loquat implements partial replication by uniquely identifying each message in the system with a namespace, or as we refer to it, a scope. Nodes in the Loquat system subscribe to zero or more of these scopes to designate which messages in the system these nodes will replicate. To ensure complete delivery of all messages to all replicas in the system, nodes track the greatest message identifier, specified using a version vector, for each scope observed, regardless of whether or not they are a replica for that scope. Quiescence is achieved when two criteria are met: all messages have been delivered to all replicas for those messages; and when all nodes have received the maximum message identifier for all scopes in the system.

B. System Model

We assume a dynamic cluster of nodes, where each node in the system has a globally unique node identifier. We assume the the crash-stop failure model, and nodes that crash recover by rejoining the cluster with a new globally unique node identifier and no state: identical to a new node. We assume non-Byzantine network and node behavior.

We assume the standard merge function for version vectors [17], and we assume that objects themselves can be merged together deterministically resulting in a monotonically greater value [18], [19].

C. Preliminaries

We assume a cluster of nodes, each with a globally unique identifier i.

$$N = \{n_i, n_{i'}, \ldots\} \tag{1}$$

We assume **scopes** are unique symbols denoting a group of messages. For simplicity, we quantify over natural numbers.

$$S = \{s_i, s_{i'}, \ldots\} \tag{2}$$

We assume **entities** in the system are the globally defined set of scopes with the globally defined set of nodes.

$$\Sigma = S \cup N \tag{3}$$

We assume, for each node i, that each node's state is a pair, σ_i the set of known scopes and their values at node i, and δ_i the set of scopes node i is responsible for replicating.

$$n_i = (\sigma_i, \delta_i) \tag{4}$$

We define a transition function for each node i, where k denotes the k-th state of an execution.

$$n_i = (\sigma_i, \delta_i) = (\sigma_i^0, \delta_i^0) \rightarrow \ldots \rightarrow (\sigma_i^k, \delta_i^k) \tag{5}$$

We define σ_i as a node's context: a set of pairs containing a scope s and a value v. More formally, σ_i is a function from a scope s to a value v.

$$\sigma_i \subseteq \{(s_i, v_i) \mid i \in \mathbb{N}\} \tag{6}$$

We define δ_i as a set of scopes that a node is interested in[7].

$$\delta_i \subseteq \{s \mid s \in \pi_1(\sigma_i)\} \tag{7}$$

We enforce the invariant that δ will be a subset of the standard projection of σ for a given node i.

$$\delta_i \subseteq \pi_1(\sigma_i) \tag{8}$$

We define values as a pair of a version vector c and a payload p.

$$V = \{(c_i, p_i) \mid i \in \mathbb{N}\} \tag{9}$$

We induce a partial order over values v with the partial order on version vectors c [20].

$$\forall (c,p).\ (c',p') \in V[c \leq c' \iff (c,p) \leq_v (c',p')] \tag{10}$$

We induce a partial order over contexts σ with the lexicographical ordering on scopes s and partial order on values v.

$$\forall (s,v).\ (s',v') \in \sigma[v \leq v' \iff (s,v) \leq_\sigma (s',v')] \tag{11}$$

We define a function α to return a maximal message in a scope s based on that partial order. We will see in Section III-D, that given the way **write** operations are performed, there can be only one maximal element.

$$\alpha(\sigma, s) = \sup(\{(s,v) \mid (s,v) \in \sigma\}) \tag{12}$$

[7] We define π as the standard projection.

D. Semantics

We begin by defining two operations for contexts, **regard** and **disregard** and two operations for the broadcast protocol, **read** and **write**. Regard is used to express interest in a scope, and disregard is used to express disinterest in a scope; therefore, they are duals. Write is used to create a message for a given scope, and read is used to return the latest message for a given scope.

We define the **regard** operation as follows. Given a scope s and a node i, update our local interest set δ accordingly.

$$\begin{aligned} regard_i(s) : \sigma_i' &= \sigma_i \cup \{(s, (\bot, \bot))\} \\ \delta_i' &= \delta_i \cup \{s\} \end{aligned} \tag{13}$$

We define the **disregard** operation as follows. Given a scope s and a node i, update our local interest set δ accordingly. We add the scope s with the bottom value to the context σ, to satisfy the invariant in Equation 8.

$$\begin{aligned} disregard_i(s) : \sigma_i' &= \sigma_i \cup \{(s, (\bot, \bot))\} \\ \delta_i' &= \delta_i \setminus \{s\} \end{aligned} \tag{14}$$

We define the **write** operation as follows. Given a scope s, a payload p, and a broadcasting node i, update our local context σ with the updated message and broadcast.

$$\begin{aligned} write_i(s, p) : m &= \alpha(\sigma, s) \\ v &= \pi_2(m) \\ c' &= incr_i(\pi_1(v)) \\ \sigma_i' &= \sigma_i \cup \{(s, (c', p))\} \\ &broadcast_i(s, (c', p)) \end{aligned} \tag{15}$$

We define the **read** operation as follows. Given a scope s and a reading node i, return the latest message in our local context σ and add the scope to the "interest" set of scopes δ.

$$\begin{aligned} read_i(s) : \sigma_i' &= \sigma_i \cup \{(s, (\bot, \bot))\} \\ \delta_i' &= \delta_i \cup \{s\} \\ &\text{return } \alpha(\sigma, s) \end{aligned} \tag{16}$$

We define a function to **merge** an incoming value v for scope s, consisting of a version vector c and a payload p to node j from node i. The merge function always transmits the most recent version vector; given that objects can be merged, the newest version vector and payload will always subsume previous state[8].

$$\begin{aligned} merge_j^i(s, (c, p)) : m &= \alpha(\sigma, s) \\ v &= \pi_2(m) \\ c' &= c \sqcup \pi_1(v) \\ p' &= p \sqcup \pi_2(v) \\ \sigma_j' &= \sigma_j \cup \{(s, (c', p'))\} \\ &broadcast_j(s, (c', p')) \end{aligned} \tag{17}$$

We define a function to **ignore** an incoming value v for a "uninteresting" scope s to node j from node i.

$$\begin{aligned} ignore_j^i(s, (c, p)) : m &= \alpha(\sigma, s) \\ v &= \pi_2(m) \\ c' &= c \sqcup \pi_1(v) \\ \sigma_j' &= \sigma_j \cup \{(s, (c', \bot))\} \\ &broadcast_i(s, (c', p)) \end{aligned} \tag{18}$$

We define a function to **drop** and incoming value v for a quiesced scope s to node j from node i.

$$drop_j^i(s, (c, p)) : \bot \tag{19}$$

We define a predicate function **seen** for an incoming value v for scope s, consisting of a version vector c and a payload p to node j from node i.

$$seen_j^i(s, (c, p)) \iff (s, (c, _)) \in \sigma_j \tag{20}$$

We define a predicate function **interested** for an incoming value v for scope s, consisting of a version vector c and a payload p to node j from node i.

$$interested_j^i(s) \iff (s, _) \in \delta_j \tag{21}$$

When receiving a broadcast message from node i at node j containing a scope s, version vector c and payload p, a receiving node places the version vector into its context σ, but only accepts the payload if the scope appears in its "interest" set δ.

For the **receive** procedure, we merge the incoming version vector along with the value for the maximum value and broadcast the resulting value after the merge. Only scopes considered "interesting" have their payloads stored; otherwise, we store bottom, to prevent this node from replicating "uninteresting" payloads. Broadcasts are continued as long as we have not previously observed the value, to guarantee events are delivered to all nodes in the broadcast tree via the eager push mechanism.

$$receive_j^i(s, (c, p)) : \begin{cases} drop_j^i(s, (c, p)) & \text{if } seen_j^i(s, (c, p)), \\ merge_j^i(s, (c, p)) & \text{if } interested_j^i(s), \\ ignore_j^i(s, (c, p)) & \text{otherwise.} \end{cases} \tag{22}$$

E. Eager and Lazy Push

We continue to use the normal eager and lazy push algorithms from the Plumtree protocol with one note: the unique message identifier used for each message is the maximum vector for a given scope, and provided explicitly to the broadcast algorithm. This ensures that eventually the greatest message for a given scope is always delivered. Given that messages are always mergable, and partially ordered, this results in eventually delivery of all messages in the system.

[8]It is also important to note, if not already clear, that for a given message once it has already been observed, the version vector merge function will result in the computation of a fixed point.

F. Late Interest

However, a client may express "interest" in a scope after all messages for that scope have been delivered. In this case, the **read** operation, after declaring "interest" will, at a subsequent anti-entropy round (as discussed in Section II-B, ensure that the message is eventually delivered to that node.

G. Optimizations

One potential optimization for the **receive** procedure would be avoiding the merge of both value and version vector if the value is already dominated by the local version vector on the receiving node. The semantics presented here are correct, but not efficient in practice. The previous work by this author on the Selective Hearing [8] contains this optimization.

Another potential optimization is to only store the greatest version vector and payload per scope. In our formal semantics, all received version vectors and payloads are stored for ease of presentation and the planned verification of the protocol.

IV. Loquat: Information Flow Control

We provide an overview followed by the formal semantics of the Loquat information flow control protocol.

A. Overview

To support information flow control, Loquat alters the Plumtree protocol by assuming that there is a single spanning tree computed for each security context and scope pair. This spanning tree begins with an empty "eager" set and a random peer selection or all nodes, in the "lazy" set. The spanning tree for that pair is then computed through the process of nodes responding to the "lazy" broadcast message, if and only if they have a compatible security context. We formally define a security context below.

B. System Model

For our initial formalism of the information flow control extension, we extend the system model presented in Section III-B. We assume a set of unique tokens representing both the level of secrecy and integrity placed on scopes of messages. We assume that messages within a given scope can only move in two directions: messages can decrease in secrecy, or increase in integrity (through additional declassification, or increased authentication.) We assume actors in the system do not act maliciously, by lying about the security contexts they are privileged to (non-Byzantine behaviour). For a discussion on using cryptographic methods from hardware for authentication and authorization of clients in an edge network, we refer the reader to [21]).

Finally, we assume that code executes on a *trusted execution platform* that enforces our system model.

C. Preliminaries

We assume a finite set of **tags**, or immutable tokens.

$$T = \{t_0, \dots, t_n\} \tag{23}$$

We define a function τ at each node i that returns a set of **secrecy** tags, or the **secrecy** label for any entity in the system, including nodes.

$$\forall e \in \Sigma.\ \tau_i(e) \in \mathcal{P}(T) \tag{24}$$

We define a function φ at each node i that returns a set of **integrity** tags, or the **integrity** label for any entity in the system, including nodes.

$$\forall e \in \Sigma.\ \varphi_i(e) \in \mathcal{P}(T) \tag{25}$$

We define the **information exchange** predicate as ω. Information, for a given scope s, is allowed to flow from a entity i to a entity j if the integrity labels and secrecy labels are compatible.

$$\omega^i_j(s) = \tau_i(s) \subseteq \tau_j(s) \wedge \varphi_j(s) \subseteq \varphi_i(s) \tag{26}$$

For each scope s at node i we maintain a **security context** at each node containing the pair of secrecy tags τ and integrity tags φ.

$$\forall s \in S.\ \forall n_i \in N.\ \gamma^s_i = (\tau_i(s), \varphi_i(s)) \tag{27}$$

Given that, we can reformulate Equation 26 in terms of security contexts, γ.

$$\gamma \rightarrow_{flow} \gamma' \iff \pi_1(\gamma) \subseteq \pi_1(\gamma') \wedge \pi_2(\gamma') \subseteq \pi_2(\gamma) \tag{28}$$

D. Protocol

We assume a single spanning tree per unique scope s and security context γ. For instance, the set of nodes that can read a given scope s with the same security context γ will compute their own spanning tree. We discuss optimizations to reduce the number of active spanning trees in Section IV-G.

More formally, each node keeps a map for a given scope and security context to a pair containing the "eager" and "lazy" sets.

$$\textit{trees} : (S, \Gamma) \rightarrow (N, N) \tag{29}$$

We assume that the lazy set for all scopes and security contexts is pre-populated with either the entire set of nodes in the cluster, or the result of a peer sampling services, for larger clusters. We augment each of the four message types (IHave, Graft, Purge, BCast) to carry the scope and security context information for every request.

We enumerate the steps in the initial spanning tree construction for a scope s and a security context γ, which may occur over several rounds of broadcast.

1) At the beginning of the execution of the protocol, no members exist in the "eager set" of the broadcasting node, and a random set of nodes is placed into the "lazy set"[9].
2) Given this, the first message that is broadcast for scope s will be the IHave message, containing the scope s, a unique message identifier derived from the version vector c, and the security context for the scope γ.

[9] Or, this could be all nodes in the cluster, depending on the cluster size and peer sampling protocol used.

3) Receiving nodes will not receive an eager broadcast within the time interval, given these messages were never transmitted with eager broadcast.
4) When the timeout expires, nodes for which the information exchange predicate ω holds, will send the receiving node the GRAFT message to move nodes into the broadcasting node's "eager set".
5) Finally, the message will be delivered by BCAST from the broadcasting node containing the actual message.

This will cause a minimal spanning tree to be computed and maintained that contains nodes that are only allowed to view the confidential information for a given scope s. Figure 5 shows a spanning tree being computed by nodes that share a security context.

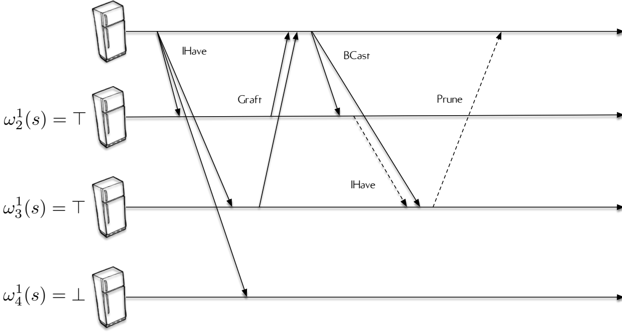


Figure 5: *Computation of a minimal spanning tree for a given security context and scope. Nodes begin with a selection of random nodes in their "lazy set" that messages are delivered to, and pruned accordingly based on message delivery. Nodes without a compatible security context are excluded.*

E. Purging

Spanning trees can be automatically purged for a given scope and security context after a given interval without broadcast messages, to reduce space leaks for contexts that are retired.

F. Anti-Entropy Protocol

When performing anti-entropy, peer selection must enforce the information exchange rule when performing value exchange with a randomly selected peer int he cluster. This is performed by initially exchanging the scopes and security contexts before performing the exchange of values.

G. Protocol Optimizations

For some scopes and security contexts, duplication of spanning trees may not be necessary. One potential optimization is to map between a single spanning tree to many scopes, if the scopes happen to share a security context. This optimization would reduce the number of in-memory spanning trees at any one time, and additionally would reduce the cost of computing the spanning tree upon the broadcast of the scopes first message.

It may also be possible to share spanning trees with less classified information and with stronger integrity across scopes, as long as the information exchange rule is preserved. However, both of these techniques require a mapping structure between scopes and security contexts in a way that is less complex in space for the optimization to be beneficial.

H. Sessions

Each node i can perform many read and write operations. To consider a particular execution safe, any write operations that follow read operations must preserve security; therefore, we must guarantee that the information exchange predicate is satisfied for all pairs of operations in a given execution.

We consider that a number of sessions may exist per node i. Each session is composed of read and write operations issued by that node.

$$Session_i \in Sessions_i \tag{30}$$

Each session contains a totally ordered set of operations executed at that node.

$$Session_i = \{Op_i, Op_{i+1}, \ldots\} \tag{31}$$

We define each operation as a scope s, the security context γ, and a value v.

$$Op_i = (s, \gamma, v) \tag{32}$$

We define a predicate function $write$ that determines whether an operation was a write operation, and a function $read$ that determines whether an operation was a read operation.

We define a function to filter the write operations for a given session.

$$\begin{aligned} writes(Session_i) = \{Op \mid Op \in Session_i \\ \wedge\ write(Op)\} \end{aligned} \tag{33}$$

We define a function to filter the read operations for a given session.

$$\begin{aligned} reads(Session_i) = \{Op \mid Op \in Session_i \\ \wedge\ read(Op)\} \end{aligned} \tag{34}$$

We can now define a function to return all of the read operations that occurred before a given write operation for a single session.

$$\begin{aligned} earlierReads(Op, Session_i) = \\ \{Op' \mid Op' \in Session_i \wedge \\ read(Op') \wedge Op' \leq Op\} \end{aligned} \tag{35}$$

For a given operation, we define a function that returns the joined security context for all previous read operations.

$$\begin{aligned} earlierReadContext(Op, Session_i) = \\ \bigcup \{\pi_2(Op') \mid \\ Op' \in earlierReads(Op, Session_i)\} \end{aligned} \tag{36}$$

We say that an execution is **safe** if the information exchange is allowed, transitively for all operations in the execution. In other words, for each write operation performed, the security

context for the write operation must be compatible with the security context of all read operations that preceded it.

$$Op \in Session_i. \quad earlierReadContext(Op, Session_i) \rightarrow_{flow} \gamma_i^{Op} \tag{37}$$

V. Related Work

Previous academic approaches for the collection of events in large-scale sensor networks, such as both the Tiny AGgregation [22] and directed and digest diffusion [23], [24] systems, either assumed that event delivery would complete for all clients within a time limited window, or involved designing a specific protocol for based on the shape of events. Neither of these approaches provides a general model for guaranteed message delivery that tolerates both churn and network failures.

VI. Future Work and Conclusion

This paper presents a work in progress report on a partially replicated, gossip protocol with information flow control for performing efficient state transmission, in a fault-tolerant manner, for large-scale sensor networks. This design builds on previous work in leveraging gossip protocols for reliable message delivery and information flow control, and ensures that state transmission can be performed transitively through nodes, in a manner where nodes are only exposed to information that is not confidential. We plan to continue the implementation and verification of our design to evaluate the protocol in practice.

Acknowledgment

Thanks to Martin Kleppmann and Jordan West for their feedback.

References

[1] M. Nijdam, private communication, 2015.

[2] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[3] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[4] C. Meiklejohn, "Declarative, secure, convergent edge computation," *CoRR*, vol. abs/1512.04898, 2015. [Online]. Available: http://arxiv.org/abs/1512.04898

[5] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," in *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '87. New York, NY, USA: ACM, 1987, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/41840.41841

[6] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky, "Bimodal multicast," *ACM Transactions on Computer Systems (TOCS)*, vol. 17, no. 2, pp. 41–88, 1999.

[7] J. Leitao, J. Pereira, and L. Rodrigues, "Epidemic broadcast trees," in *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE, 2007, pp. 301–310.

[8] C. Meiklejohn and P. Van Roy, "Selective Hearing: An Approach to Distributed, Eventually Consistent Edge Computation," in *Workshop on Planetary-Scale Distributed Systems collocated with SRDS 2015*. IEEE, 2015.

[9] ——, "Lasp: A language for distributed, coordination-free programming," in *Proceedings of the 17th Symposium on Principles and Practice of Declarative Programming (PPDP 2015)*. ACM, Jul. 2015.

[10] J. Leitao, J. Pereira, and L. Rodrigues, "Hyparview: A membership protocol for reliable gossip-based broadcast," in *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE, 2007, pp. 419–429.

[11] "Controlled Epidemics: Riak's New Gossip Protocol and Metadata Store," https://www.youtube.com/watch?v=s4cCUTPU8GI, accessed: 2016-02-06.

[12] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 9, no. 4, pp. 410–442, 2000.

[13] T. Pasquier, J. Singh, D. Eyers, and J. Bacon, "Camflow: Managed data-sharing for cloud services," 2015.

[14] D. E. Denning, "A lattice model of secure information flow," *Communications of the ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[15] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," DTIC Document, Tech. Rep., 1973.

[16] K. J. Biba, "Integrity considerations for secure computer systems," DTIC Document, Tech. Rep., 1977.

[17] D. S. Parker Jr, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, "Detection of mutual inconsistency in distributed systems," *Software Engineering, IEEE Transactions on*, no. 3, pp. 240–247, 1983.

[18] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, *Managing update conflicts in Bayou, a weakly connected replicated storage system*. ACM, 1995, vol. 29, no. 5.

[19] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," INRIA, Tech. Rep. RR-7506, 01 2011.

[20] F. Mattern, "Virtual time and global states of distributed systems," *Parallel and Distributed Algorithms*, vol. 1, no. 23, pp. 215–226, 1989.

[21] T. F.-M. Pasquier, J. Singh, and J. Bacon, "Clouds of things need information flow control with hardware roots of trust."

[22] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: A tiny aggregation service for ad-hoc sensor networks," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[23] C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: a scalable and robust communication paradigm for sensor networks," in *Proceedings of the 6th annual international conference on Mobile computing and networking*. ACM, 2000, pp. 56–67.

[24] J. Zhao, R. Govindan, and D. Estrin, "Computing aggregates for monitoring wireless sensor networks," in *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*. IEEE, 2003, pp. 139–148.

7.3 Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta state replicated data types. Submitted for publication, 2016.

Delta State Replicated Data Types

Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero

HASLab/INESC TEC and Universidade do Minho, Portugal

Abstract. CRDTs are distributed data types that make eventual consistency of a distributed object possible and non ad-hoc. Specifically, state-based CRDTs ensure convergence through disseminating the entire state, that may be large, and merging it to other replicas; whereas operation-based CRDTs disseminate operations (i.e., small states) assuming an exactly-once reliable dissemination layer. We introduce *Delta State Conflict-Free Replicated Data Types* (δ-CRDT) that can achieve the best of both worlds: small messages with an incremental nature, as in operation-based CRDTs, disseminated over unreliable communication channels, as in traditional state-based CRDTs. This is achieved by defining *δ-mutators* to return a *delta-state*, typically with a much smaller size than the full state, that to be joined with both local and remote states. We introduce the δ-CRDT framework, and we explain it through establishing a correspondence to current state-based CRDTs. In addition, we present an anti-entropy algorithm for eventual convergence, and another one that ensures causal consistency. Finally, we introduce several δ-CRDT specifications of both well-known replicated datatypes and novel datatypes, including a generic map composition.

arXiv:1603.01529v1 [cs.DC] 4 Mar 2016

1 Introduction

Eventual consistency (EC) is a relaxed consistency model that is often adopted by large-scale distributed systems [1,2,3] where availability must be maintained, despite outages and partitioning, whereas delayed consistency is acceptable. A typical approach in EC systems is to allow replicas of a distributed object to temporarily diverge, provided that they can eventually be reconciled into a common state. To avoid application-specific reconciliation methods, costly and error-prone, *Conflict-Free Replicated Data Types* (CRDTs) [4,5] were introduced, allowing the design of self-contained distributed data types that are always available and eventually converge when all operations are reflected at all replicas. Though CRDTs are deployed in practice and support millions of users worldwide [6,7,8], more work is still required to improve their design and performance.

CRDTs support two complementary designs: *operation-based* (or op-based) and *state-based*. In op-based designs [9,5], the execution of an operation is done in two phases: *prepare* and *effect*. The former is performed only on the local replica and looks at the operation and current state to produce a message that aims to represent the operation, which is then shipped to all replicas. Once received, the representation of the operation is applied remotely using *effect*.

On the other hand, in a state-based design [10,5] an operation is only executed on the local replica state. A replica periodically propagates its local changes to other replicas through shipping its entire state. A received state is incorporated with the local state via a *merge* function that deterministically reconciles both states. To maintain convergence, *merge* is defined as a *join*: a least upper bound over a join-semilattice [10,5].

Op-based CRDTs have some advantages as they can allow for simpler implementations, concise replica state, and smaller messages; however, they are subject to some limitations: First, they assume a message dissemination layer that guarantees reliable exactly-once causal broadcast; these guarantees are hard to maintain since large logs must be retained to prevent duplication even if TCP is used [11]. Second, membership management is a hard task in op-based systems especially once the number of nodes gets larger or due to churn problems, since all nodes must be coordinated by the middleware. Third, the op-based approach requires operations to be executed individually (even when batched) on all nodes.

The alternative is to use state-based systems, which are free from these limitations. However, a major drawback in current state-based CRDTs is the communication overhead of shipping the entire state, which can get very large in size. For instance, the state size of a *counter* CRDT (a vector of integer counters, one per replica) increases with the number of replicas; whereas in a *grow-only Set*, the state size depends on the set size, that grows as more operations are invoked. This communication overhead limits the use of state-based CRDTs to data-types with small state size (e.g., counters are reasonable while large sets are not). Recently, there has been a demand for CRDTs with large state sizes (e.g., in RIAK DT Maps [12] that can compose multiple CRDTs and that we formalize in Section 7.4).

In this paper, we rethink the way state-based CRDTs should be designed, having in mind the problematic shipping of the entire state. Our aim is to ship a *representation of the effect* of recent update operations on the state, rather than the whole state, while preserving the idempotent nature of *join*. This ensures convergence over unreliable communication (on the contrary to op-based CRDTs that demand exactly-once delivery and are prone to message duplication). To achieve this, we develop in detail the concept of *Delta State-based CRDTs* (δ-CRDT) that we initially introduced in [13]. In this new (delta) framework, the state is still a join-semilattice that now results from the join of multiple fine-grained states, i.e., *deltas*, generated by what we call *δ-mutators*. *δ-mutators* are new versions of the datatype mutators that return the effect of these mutators on the state. In this way, deltas can be temporarily retained in a buffer to be shipped individually (or joined in groups) instead of shipping the entire object. The changes to the local state are then incorporated at other replicas by joining the shipped deltas with their own states.

The use of "deltas" (i.e., incremental states) may look intuitive in state dissemination; however, this is not the case for state-based CRDTs. The reason is that once a node receives an entire state, merging it locally is simple since there

is no need to care about causality, as both states are self-contained (including meta-data). The challenge in δ-CRDT is that individual deltas are now "state fragments" and usually must be causally merged to maintain the desired semantics. This raises the following questions: is merging deltas semantically equivalent to merging entire states in CRDTs? If not, what are the sufficient conditions to make this true in general? And under what constraints causal consistency is maintained? This paper answers these questions and presents corresponding proofs and examples.

We address the challenge of designing a new δ-CRDT that conserves the correctness properties and semantics of an existing CRDT by establishing a relation between the novel δ-mutators with the original CRDT mutators. We prove that eventual consistency is guaranteed in δ-CRDT as long as all deltas produced by δ-mutators are delivered and joined at other replicas, and we present a corresponding simple anti-entropy algorithm. We then show how to ensure causal consistency using deltas through introducing the concept of *delta-interval* and the *causal delta-merging condition*. Based on these, we then present an anti-entropy algorithm for δ-CRDT, where sending and then joining delta-intervals into another replica state produces the same effect as if the entire state had been shipped and joined.

We illustrate our approach through a simple *counter* CRDT and a corresponding δ-CRDT specification. Later, we present a portfolio of several δ-CRDT*s* that adapt known CRDT designs and also introduce a generic kernel for the definition of CRDTs that keep a causal history of known events and a CRDT map that can compose them. All these δ-CRDT datatypes, and a few more, are available online in a reference C++ library [14]. Our experience shows that a δ-CRDT version can be devised for all CRDTs, but this requires some design effort that varies with the complexity of different CRDTs. This refactoring effort can be avoided for new datatypes by writing all mutations as delta-mutations, and only deriving the standard mutators if needed; these can be trivially obtained from the delta-mutators.

2 System Model

Consider a distributed system with nodes containing local memory, with no shared memory between them. Any node can send messages to any other node. The network is asynchronous; there is no global clock, no bound on the time a message takes to arrive, and no bounds on relative processing speeds. The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted). Some messages will, however, eventually get through: if a node sends infinitely many messages to another node, infinitely many of these will be delivered. In particular, this means that there can be arbitrarily long partitions, but these will eventually heal. Nodes have access to durable storage; they can crash but will eventually recover with the content of the durable storage just before the crash occurred. Durable state is written atomically at each state transition. Each node has access to its globally unique identifier in a set $\mathbb{I}$.

2.1 Notation

We use mostly standard notation for sets and maps, including set comprehension of the forms $\{f(x) \mid x \in S\}$ or $\{x \in S \mid Pred(x)\}$. A map is a set of (k, v) pairs, where each k is associated with a single v. Given a map m, $m(k)$ returns the value associated with key k, while $m\{k \mapsto v\}$ denotes m updated by mapping k to v. The domain and range of a map m is denoted by $\mathsf{dom}\, m$ and $\mathsf{ran}\, m$, respectively, i.e., $\mathsf{dom}\, m = \{k \mid (k, v) \in m\}$ and $\mathsf{ran}\, m = \{v \mid (k, v) \in m\}$. We use $\mathsf{fst}\, p$ and $\mathsf{snd}\, p$ to denote the first and second component of a pair p, respectively. We use $\mathbb{B}$, $\mathbb{N}$, and $\mathbb{Z}$, for the booleans, natural numbers, and integers, respectively; also $\mathbb{I}$ for some unspecified set of node identifiers. Most sets we use are partially ordered and have a least element $\bot$ (the bottom element). We use $A \hookrightarrow B$ for a partial function from A to B; given such a map m, then $\mathsf{dom}\, m \subseteq A$ and $\mathsf{ran}\, m \subseteq B$, and for convenience we use $m(k)$ when $k \notin \mathsf{dom}\, m$ and B has a bottom, to denote $\bot_B$; e.g., for some $m : \mathbb{I} \hookrightarrow \mathbb{N}$, then $m(k)$ denotes 0 for any unmapped key k.

3 A Background of State-based CRDTs

Conflict-Free Replicated Data Types [4,5] (CRDTs) are distributed datatypes that allow different replicas of a distributed CRDT instance to diverge and ensures that, eventually, all replicas converge to the same state. State-based CRDTs achieve this through propagating updates of the local state by disseminating the entire state across replicas. The received states are then merged to remote states, leading to convergence (i.e., consistent states on all replicas).

A state-based CRDT consists of a triple (S, M, Q), where S is a join-semilattice [15], Q is a set of query functions (which return some result without modifying the state), and M is a set of mutators that perform updates; a mutator $m \in M$ takes a state $X \in S$ as input and returns a new state $X' = m(X)$. A join-semilattice is a set with a *partial order* $\sqsubseteq$ and a binary *join* operation $\sqcup$ that returns the *least upper bound* (LUB) of two elements in S; a *join* is designed to be commutative, associative, and idempotent. Mutators are defined in such a way to be *inflations*, i.e., for any mutator m and state X, the following holds:

$$X \sqsubseteq m(X)$$

In this way, for each replica there is a monotonic sequence of states, defined under the lattice partial order, where each subsequent state subsumes the previous state when joined elsewhere.

Both query and mutator operations are always available since they are performed using the local state without requiring inter-replica communication; however, as mutators are concurrently applied at distinct replicas, replica states will likely diverge. Eventual convergence is then obtained using an *anti-entropy* protocol that periodically ships the entire local state to other replicas. Each replica merges the received state with its local state using the *join* operation in S. Given the mathematical properties of *join*, if mutations stop being issued and anti-entropy proceeds, all replicas eventually converge to the same state. i.e. the

$$
\begin{aligned}
\mathsf{GCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\bot &= \{\} \\
\mathsf{inc}_i(m) &= m\{i \mapsto m(i) + 1\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} m(j) \\
m \sqcup m' &= \{j \mapsto \mathsf{max}(m(j), m'(j)) \mid j \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 1: State-based Counter CRDT; replica i.

least upper-bound of all states involved. State-based CRDTs are interesting as they demand little guarantees from the dissemination layer, working under message loss, duplication, reordering, and temporary network partitioning, without impacting availability and eventual convergence.

Fig. 1 represents a state-based increment-only counter. The GCounter CRDT state is a map from replica identifiers to positive integers. Initially, the bottom state $\bot$ is an empty map (unmapped keys implicitly mapping to zero). A single mutator, i.e., inc_i, is defined that increments the value corresponding to the local replica i (returning the updated map). The query operation value returns the counter value by adding the integers in the map entries. The join of two states is the point-wise maximum of the maps. Mutators, like inc_i, are in general parameterized by the replica id i, so that their exact behavior can depend on it, while queries, like value, are typically replica agnostic and only depend on the CRDT state, regardless of in which replica they are invoked.

The main weakness of state-based CRDTs is the cost of dissemination of updates, as the full state is sent. In this simple example of counters, even though increments only update the value corresponding to the local replica i, the whole map will always be sent in messages, even when the other map entries remained unchanged (e.g., if no messages have been received and merged).

It would be interesting to only ship the recent modification incurred on the state, and possibly any received modifications that effectively changed it. This is, however, not possible with the current model of state-based CRDTs as mutators always return a full state. Approaches which simply ship operations (e.g., an "increment n" message), like in operation-based CRDTs, require reliable communication (e.g., because increment is not idempotent). In contrast, the modification that we introduce in the next section allows producing and encoding recent mutations in an incremental way, while keeping the advantages of the state-based approach, namely the idempotent, associative, and commutative properties of join.

4 Delta-state CRDTs

We introduce *Delta-State Conflict-Free Replicated Data Types*, or δ-CRDT for short, as a new kind of state-based CRDTs, in which *delta-mutators* are defined

to return a *delta-state*: a value in the same join-semilattice which represents the updates induced by the mutator on the current state.

Definition 1 (Delta-mutator). *A delta-mutator m^δ is a function, corresponding to an update operation, which takes a state X in a join-semilattice S as parameter and returns a delta-mutation $m^\delta(X)$, also in S.*

Definition 2 (Delta-group). *A delta-group is inductively defined as either a delta-mutation or a join of several delta-groups.*

Definition 3 (δ-CRDT). *A δ-CRDT consists of a triple (S, M^δ, Q), where S is a join-semilattice, M^δ is a set of delta-mutators, and Q a set of query functions, where the state transition at each replica is given by either joining the current state $X \in S$ with a delta-mutation:*

$$X' = X \sqcup m^\delta(X),$$

or joining the current state with some received delta-group D:

$$X' = X \sqcup D.$$

In a δ-CRDT, the effect of applying a mutation, represented by a delta-mutation $\delta = m^\delta(X)$, is decoupled from the resulting state $X' = X \sqcup \delta$, which allows shipping this δ rather than the entire resulting state X'. All state transitions in a δ-CRDT, even upon applying mutations locally, are the result of some join with the current state. Unlike standard CRDT mutators, delta-mutators do not need to be inflations in order to inflate a state; this is however ensured by joining their output, i.e., deltas, into the current state: $X \sqsubseteq X \sqcup m^\delta(X)$.

In principle, a delta could be shipped immediately to remote replicas once applied locally. For efficiency reasons, multiple deltas returned by applying several delta-mutators can be joined locally into a delta-group and retained in a buffer. The delta-group can then be shipped to remote replicas to be joined with their local states. Received delta-groups can optionally be joined into their buffered delta-group, allowing transitive propagation of deltas. A full state can be seen as a special (extreme) case of a delta-group.

If the causal order of operations is not important and the intended aim is merely eventual convergence of states, then delta-groups can be shipped using an unreliable dissemination layer that may drop, reorder, or duplicate messages. Delta-groups can always be re-transmitted and re-joined, possibly out of order, or can simply be subsumed by a less frequent sending of the full state, e.g., for performance reasons or when doing state transfers to new members.

4.1 Delta-state decomposition of standard CRDTs

A δ-CRDT (S, M^δ, Q) is a *delta-state decomposition* of a state-based CRDT (S, M, Q), if for every mutator $m \in M$, we have a corresponding mutator $m^\delta \in M^\delta$ such that, for every state $X \in S$:

$$m(X) = X \sqcup m^\delta(X)$$

This equation states that applying a delta-mutator and joining into the current state should produce the same state transition as applying the corresponding mutator of the standard CRDT.

Given an existing state-based CRDT (which is always a trivial decomposition of itself, i.e., $m(X) = X \sqcup m(X)$, as mutators are inflations), it will be useful to find a non-trivial decomposition such that delta-states returned by delta-mutators in M^δ are smaller than the resulting state:

$$\mathsf{size}(m^\delta(X)) \ll \mathsf{size}(m(X))$$

In general, there are several possible delta-state decompositions, with multiple possible delta-mutators that correspond to each standard mutator. In order to minimize the generated delta-states (which will typically minimize their size) each delta-mutator chosen m^δ should be minimal in following sense: for any other alternative choice of delta-mutator $m^{\delta'}$, for any X, $m^{\delta'}(X) \not\sqsubset m^\delta(X)$. Intuitively, minimal delta-mutators do not leak into the deltas they produce any redundant information that is already present in X. Moreover (although in theory not always necessarily the case) for typical datatypes that we have come across in practice, for each mutator m there exists a corresponding *minimum* delta-mutator $m^{\delta_\bot}$, i.e., with $m^{\delta_\bot} \sqsubseteq m^{\delta'}$ (under the standard pointwise function comparison), for any alternative delta-mutator. As we will see in the concrete examples, typically minimum delta-mutators are found naturally, with no need for some special "search".

4.2 Example: δ-CRDT Counter

Fig. 2 depicts a δ-CRDT specification of a counter datatype that is a delta-state decomposition of the state-based counter in Fig. 1. The state, join and value query operation remain as before. Only the mutator inc^δ is newly defined, which increments the map entry corresponding to the local replica and only returns that entry, instead of the full map as inc in the state-based CRDT counter does. This maintains the original semantics of the counter while allowing the smaller deltas returned by the delta-mutator to be sent, instead of the full map. As before, the received payload (whether one or more deltas) might not include entries for all keys in $\mathbb{I}$, which are assumed to have zero values. The decomposition is easy to understand in this example since the equation $\mathsf{inc}_i(X) = X \sqcup \mathsf{inc}_i^\delta(X)$ holds as $m\{i \mapsto m(i)+1\} = m \sqcup \{i \mapsto m(i)+1\}$. In other words, the single value for key i in the delta, corresponding to the local replica identifier, will overwrite the corresponding one in m since the former maps to a higher value (i.e., using max). Here it can be noticed that: (1) a delta *is* just a state, that can be joined possibly several times without requiring exactly-once delivery, and without being a representation of the "increment" operation (as in operation-based CRDTs), which is itself non-idempotent; (2) joining deltas into a delta-group and disseminating delta-groups at a lower rate than the operation rate

$$
\begin{aligned}
\mathsf{GCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \\
\bot &= \{\} \\
\mathsf{inc}^\delta_i(m) &= \{i \mapsto m(i) + 1\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} m(j) \\
m \sqcup m' &= \{j \mapsto \mathsf{max}(m(j), m'(j)) \mid j \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 2: A δ-CRDT counter; replica i.

reduces data communication overhead, since multiple increments from a given source can be collapsed into a single state counter.

5 State Convergence

In the δ-CRDT execution model, and regardless of the anti-entropy algorithm used, a replica state always evolves by joining the current state with some *delta*: either the result of a delta-mutation, or some arbitrary delta-group (which itself can be expressed as a join of delta-mutations). Without loss of generality, we assume S has a bottom $\bot$ which is also the initial state. (Otherwise, a bottom can always be added, together with a special init delta-mutator, which returns the initial state.) Therefore, all states can be expressed as joins of delta-mutations, which makes state convergence in δ-CRDT easy to achieve: it is enough that all delta-mutations generated in the system reach every replica, as expressed by the following proposition.

Proposition 1. *(δ-CRDT convergence) Consider a set of replicas of a δ-CRDT object, replica i evolving along a sequence of states $X_i^0 = \bot, X_i^1, \ldots$, each replica performing delta-mutations of the form $m^\delta_{i,k}(X_i^k)$ at some subset of its sequence of states, and evolving by joining the current state either with self-generated deltas or with delta-groups received from others. If each delta-mutation $m^\delta_{i,k}(X_i^k)$ produced at each replica is joined (directly or as part of a delta-group) at least once with every other replica, all replica states become equal.*

Proof. Trivial, given the associativity, commutativity, and idempotence of the join operation in any join-semilattice.

This opens up the possibility of having anti-entropy algorithms that are only devoted to enforce convergence, without necessarily providing causal consistency (enforced in standard CRDTs); thus, making a trade-off between performance and consistency guarantees. For instance, in a counter (e.g., for the number of *likes* on a social network), it may not be critical to have causal consistency, but merely not to lose increments and achieve convergence.

```
1  inputs:
2      n_i ∈ P(I), set of neighbors
3      t_i ∈ B, transitive mode
4      choose_i ∈ S × S → S, state/delta
5  durable state:
6      X_i ∈ S, CRDT state, X_i^0 = ⊥
7  volatile state:
8      D_i ∈ S, delta-group, D_i^0 = ⊥
9  on operation_i(m^δ)
10     d = m^δ(X_i)
11     X'_i = X_i ⊔ d
12     D'_i = D_i ⊔ d
13 on receive_{j,i}(d)
14     X'_i = X_i ⊔ d
15     if t_i then
16         D'_i = D_i ⊔ d
17     else
18         D'_i = D_i
19 periodically
20     m = choose_i(X_i, D_i)
21     for j ∈ n_i do
22         send_{i,j}(m)
23     D'_i = ⊥
```

Algorithm 1: Basic anti-entropy algorithm for δ-CRDT.

5.1 Basic Anti-Entropy Algorithm

A basic anti-entropy algorithm that ensures eventual convergence in δ-CRDT is presented in Algorithm 1. For the node corresponding to replica i, the durable state, which persists after a crash, is simply the δ-CRDT state X_i. The volatile state D stores a delta-group that is used to accumulate deltas before eventually sending it to other replicas. The initial value for both X_i and D_i is $\bot$.

When an operation is performed, the corresponding delta-mutator m^δ is applied to the current state of X_i, generating a delta d. This delta is joined both with X_i to produce a new state, and with D. In the same spirit of standard state based CRDTs, a node sends its messages in a periodic fashion, where the message payload is either the delta-group D_i or the full state X_i; this decision is made by the function choose_i which returns one of them. To keep the algorithm simple, a node simply broadcasts its messages without distinguishing between neighbors. After each send, the delta-group is reset to $\bot$.

Once a message is received, the payload d is joined into the current δ-CRDT state. The basic algorithm operates in one of two modes: (1) a *transitive* mode (when t_i is true) in which d is also joined into D, allowing transitive propagation of delta-mutations, where deltas received at node i from some node j can later be sent to some other node k; (2) a *direct* mode where a delta-group is exclusively the join of local delta-mutations (j must send its deltas directly to k). The decisions of whether to send a delta-group versus the full state (typically less periodically), and whether to use the transitive or direct mode are out of the scope of this paper. In general, decisions can be made considering many criteria like delta-group size, state size, message loss distribution assumptions, and network topology.

6 Causal Consistency

Traditional state-based CRDTs converge using joins of the full state, which implicitly ensures per-object causal consistency [16]: each state of some replica of

an object reflects the causal past of operations on the object (either applied locally, or applied at other replicas and transitively joined).

Therefore, it is desirable to have δ-CRDTs offer the same causal-consistency guarantees that standard state-based CRDTs offer. This raises the question about how can delta propagation and merging of δ-CRDT be constrained (and expressed in an anti-entropy algorithm) in such a manner to give the same results as if a standard state-based CRDT was used. Towards this objective, it is useful to define a particular kind of delta-group, which we call a *delta-interval*:

Definition 4 (Delta-interval). *Given a replica i progressing along the states $X_i^0, X_i^1, \ldots,$ by joining delta d_i^k (either local delta-mutation or received delta-group) into X_i^k to obtain X_i^{k+1}, a delta-interval $\Delta_i^{a,b}$ is a delta-group resulting from joining deltas $d_i^a, \ldots, d_i^{b-1}$:*

$$\Delta_i^{a,b} = \bigsqcup \{d_i^k \mid a \leq k < b\}$$

The use of delta-intervals in anti-entropy algorithms will be a key ingredient towards achieving causal consistency. We now define a restricted kind of anti-entropy algorithm for δ-CRDTs.

Definition 5 (Delta-interval-based anti-entropy algorithm). *A given anti-entropy algorithm for δ-CRDTs is delta-interval-based, if all deltas sent to other replicas are delta-intervals.*

Moreover, to achieve causal consistency the next condition must satisfied:

Definition 6 (Causal delta-merging condition). *A delta-interval based anti-entropy algorithm is said to satisfy the causal delta-merging condition if the algorithm only joins $\Delta_j^{a,b}$ from replica j into state X_i of replica i that satisfy:*

$$X_i \sqsupseteq X_j^a.$$

This means that a delta-interval is only joined into states that at least reflect (i.e., subsume) the state into which the first delta in the interval was previously joined. The causal delta-merging condition is important, since any delta-interval based anti-entropy algorithm for a δ-CRDT that satisfies it can be used to obtain the same outcome of a standard CRDT; this is formally stated in Proposition 2.

Proposition 2. *(CRDT and δ-CRDT correspondence) Let (S, M, Q) be a standard state-based CRDT and (S, M^δ, Q) a corresponding delta-state decomposition. Any δ-CRDT state reachable by an execution E^δ over (S, M^δ, Q), by a delta-interval based anti-entropy algorithm A^δ satisfying the causal delta-merging condition, is equal to a state resulting from an execution E over (S, M, Q), having the corresponding data-type operations, by an anti-entropy algorithm A for state-based CRDTs.*

Proof. By simulation, establishing a correspondence between an execution E^δ, and execution E of a standard CRDT of which (S, M^δ, Q) is a decomposition, as

follows: 1) the state $(X_i, D_i, \ldots)$ of each node in E^δ containing CRDT state X_i, information about delta-intervals D_i and possibly other information, corresponds to only X_i component (in the same join-semilattice); 2) for each action which is a delta-mutation m^δ in E^δ, E executes he corresponding mutation m, satisfying $m(X) = X \sqcup m^\delta(X)$; 3) whenever E^δ contains a send action of a delta-interval $\Delta_i^{a,b}$, execution E contains a send action containing the full state X_i^b; 4) whenever E^δ performs a join into some X_i of a delta-interval $\Delta_j^{a,b}$, execution E delivers and joins the corresponding message containing the full CRDT state X_j^b. By induction on the length of the trace, assume that for each replica i, each node state X_i in E is equal to the corresponding component in the node state in E^δ, up to the last action in the global trace. A send action does not change replica state, preserving the correspondence. Replica states only change either by performing data-type update operations or upon message delivery by merging deltas/states respectively. If the next action is an update operation, the correspondence is preserved due to the delta-state decomposition property $m(X) = X \sqcup m^\delta(X)$. If the next action is a message delivery at replica i, with a merging of delta-interval/state from other replica j, because algorithm A^δ satisfies the causal merging-condition, it only joins into state X_i^k a delta-interval $\Delta_j^{a,b}$ if $X_i^k \sqsupseteq X_j^a$. In this case, the outcome will be:

$$
\begin{aligned}
X_i^{k+1} &= X_i^k \sqcup \Delta_j^{a,b} \\
&= X_i^k \sqcup \bigsqcup \{d_j^l \mid a \leq l < b\} \\
&= X_i^k \sqcup X_j^a \sqcup \bigsqcup \{d_j^l \mid a \leq l < b\} \\
&= X_i^k \sqcup X_j^a \sqcup d_j^a \sqcup d_j^{a+1} \sqcup \ldots \sqcup d_j^{b-1} \\
&= X_i^k \sqcup X_j^{a+1} \sqcup d_j^{a+1} \sqcup \ldots \sqcup d_j^{b-1} \\
&= \ldots \\
&= X_i^k \sqcup X_j^{b-1} \sqcup d_j^{b-1} \\
&= X_i^k \sqcup X_j^b
\end{aligned}
$$

The resulting state X_i^{k+1} in E^δ will be, therefore, the same as the corresponding one in E where the full CRDT state from j has been joined, preserving the correspondence between E^δ and E.

Corollary 1. *(δ-CRDT causal consistency) Any δ-CRDT in which states are propagated and joined using a delta-interval-based anti-entropy algorithm satisfying the causal delta-merging condition ensures causal consistency.*

Proof. From Proposition 2 and causal consistency of state-based CRDTs.

6.1 Anti-Entropy Algorithm for Causal Consistency

Algorithm 2 is a delta-interval based anti-entropy algorithm which enforces the causal delta-merging condition. It can be used whenever the causal consistency

1 **inputs:**
2 $\quad n_i \in \mathcal{P}(\mathbb{I})$, set of neighbors
3 **durable state:**
4 $\quad X_i \in S$, CRDT state, $X_i^0 = \bot$
5 $\quad c_i \in \mathbb{N}$, sequence number, $c_i^0 = 0$
6 **volatile state:**
7 $\quad D_i \in \mathbb{N} \hookrightarrow S$, deltas, $D_i^0 = \{\}$
8 $\quad A_i \in \mathbb{I} \hookrightarrow \mathbb{N}$, ack map, $A_i^0 = \{\}$
9 **on** $\mathsf{receive}_{j,i}(\mathsf{delta}, d, n)$
10 $\quad$ **if** $d \not\sqsubseteq X_i$ **then**
11 $\quad\quad X_i' = X_i \sqcup d$
12 $\quad\quad D_i' = D_i\{c_i \mapsto d\}$
13 $\quad\quad c_i' = c_i + 1$
14 $\quad \mathsf{send}_{i,j}(\mathsf{ack}, n)$
15 **on** $\mathsf{receive}_{j,i}(\mathsf{ack}, n)$
16 $\quad A_i' = A_i\{j \mapsto \mathsf{max}(A_i(j), n)\}$
17 **on** $\mathsf{operation}_i(m^\delta)$
18 $\quad d = m^\delta(X_i)$
19 $\quad X_i' = X_i \sqcup d$
20 $\quad D_i' = D_i\{c_i \mapsto d\}$
21 $\quad c_i' = c_i + 1$
22 **periodically** // ship interval or state
23 $\quad j = \mathsf{random}(n_i)$
24 $\quad$ **if** $D_i = \{\} \vee \min \mathsf{dom}\, D_i > A_i(j)$ **then**
25 $\quad\quad d = X_i$
26 $\quad$ **else**
27 $\quad\quad d = \bigsqcup\{D_i(l) \mid A_i(j) \leq l < c_i\}$
28 $\quad$ **if** $A_i(j) < c_i$ **then**
29 $\quad\quad \mathsf{send}_{i,j}(\mathsf{delta}, d, c_i)$
30 **periodically** // garbage collect deltas
31 $\quad l = \min\{n \mid (_, n) \in A_i\}$
32 $\quad D_i' = \{(n, d) \in D_i \mid n \geq l\}$

Algorithm 2: Anti-entropy algorithm ensuring causal consistency of δ-CRDT.

guarantees of standard state-based CRDTs are needed. For simplicity, it excludes some optimizations that are important in practice, but easy to derive. The algorithm distinguishes neighbor nodes, and only sends to each one appropriate delta-intervals that obey the delta-merging condition and can joined at the receiving node.

Each node i keeps a contiguous sequence of deltas $d_i^l, \ldots, d_i^u$ in a map D from integers to deltas, with $l = \min \mathsf{dom}\, D$ and $u = \mathsf{max}\, \mathsf{dom}\, D$. The sequence numbers of deltas are obtained from the counter c_i that is incremented when a delta (whether a delta-mutation or delta-interval received) is joined with the current state. Each node i keeps an acknowledgments map A that stores, for each neighbor j, the largest index b for all delta-intervals $\Delta_i^{a,b}$ acknowledged by j (after j receives $\Delta_i^{a,b}$ from i and joins it into X_j).

Node i sends a delta-interval $d = \Delta_i^{a,b}$ with a (delta, d, b) message; the receiving node j, after joining $\Delta_i^{a,b}$ into its replica state, replies with an acknowledgment message (ack, b); if an ack from j was successfully received by node i, it updates the entry of j in the acknowledgment map, using the max function. This handles possible old duplicates and messages arriving out of order.

Like the δ-CRDT state, the counter c_i is also kept in a durable storage. This is essential to cope with potential crash and recovery incidents. Otherwise, there would be the danger of receiving some delayed ack, for a delta-interval sent before crashing, causing the node to skip sending some deltas generated after recovery, thus violating the delta-merging condition.

The algorithm for node i periodically picks a random neighbor j. In principle, i sends the join of all deltas starting from the latest delta acked by j. Exceptionally, i sends the entire state in two cases: (1) if the sequence of deltas

D_i is empty, or (2) if j is expecting from i a delta that was already removed from D_i (e.g., after a crash and recovery, when both deltas and the ack map, being volatile state, are lost). A delta message is only sent if the counter c_i has advanced past the next delta expected by node j, i.e., if $A_i(j) < c_i$, to avoid sending the full state in local inactivity periods, when no local operations are being issued, all neighbor nodes have acked all deltas, and garbage collection has been applied, making the D_i map empty. To garbage collect old deltas, the algorithm periodically removes the deltas that have been acked by *all* neighbors.

Proposition 3. *Algorithm 2 produces the same reachable states as a standard algorithm over a CRDT for which the δ-CRDT is a decomposition, ensuring causal consistency.*

Proof. From Proposition 2 and Corollary 1, it is enough to prove that the algorithm satisfies the causal delta-merging condition. The algorithm explicitly keeps deltas d_i^k tagged with increasing sequence numbers (even after a crash), according with the definition; node j only sends to i a delta-interval $\Delta_j^{a,b}$ if i has acked a; this ack is sent only if i has already joined some delta-interval (possibly a full state) $\Delta_j^{k,a}$. Either $k = 0$ or, by the same reasoning, this $\Delta_j^{k,a}$ could only have been joined at i if some other interval $\Delta_j^{l,k}$ had already been joined into i. This reasoning can be recursed until a delta-interval starting from zero is reached. Therefore, $X_i \sqsupseteq \bigsqcup\{d_j^k \mid 0 \leq k < a\} = \Delta_j^{0,a} = X_j^a$.

7 Portfolio of δ-CRDTs

Having established the equivalence to classic state based CRDTs we now derive a series of specifications based on delta-mutators. Although we cover a significant number of CRDTs, the goal is not to provide an exhaustive survey, but instead to illustrate more extensively the design of specifications with deltas. In our experience the intellectual effort of designing a delta-based CRDT is not much higher than designing it with standard mutators. Since standard mutators can be obtained from delta-mutators, by composing these with join, having delta-mutators as basic building blocks can only add flexibility to the designs.

First, we will cover simple CRDTs and CRDT compositions that do not require distinguished node identifiers for the mutation. Next, we cover CRDTs that require a unique identifier for each replica that is allowed to mutate the state, and make use of this identifier in one or more of the mutations. Then, we address the important class of what we denote by *Causal CRDTs*, presenting a generic design in which the state lattice is formed by a *dot store* and a *causal context*. We define three such dot stores and corresponding lattices, which are then used to defined several causal CRDTs. We conclude the portfolio with a Map design, a causal CRDT which can correctly embed any causal CRDT, including the map itself.

All of the selected CRDTs have delta implementations available in C++ [14], that complement the specifications. Most of the Causal CRDTs, including the

$$\begin{aligned} \mathsf{Pair}\langle A, B\rangle &= A \times B \\ \bot &= (\bot, \bot) \\ (a, b) \sqcup (a', b') &= (a \sqcup a', b \sqcup b') \end{aligned}$$

Fig. 3: Pair of join-semilattices.

$$\begin{aligned} \mathsf{LexPair}\langle A, B\rangle &= A \boxtimes B \\ \bot &= (\bot, \bot) \\ (a, b) \sqcup (a', b') &= \begin{cases} (a, b) & \text{if } a > a' \\ (a', b') & \text{if } a' > a \\ (a, b \sqcup b') & \text{if } a = a' \\ (a \sqcup a', \bot) & \text{otherwise} \end{cases} \end{aligned}$$

Fig. 4: Lexicographic pair of join-semilattices.

Map, are also available in Erlang and deployed in production as part of Riak DT [17].

7.1 Simple Lattice Compositions

To obtain composite CRDTs, a basic ingredient is being able to obtain states, which are join-semilattices, as composition of join-semilattices. Two common useful cases are the product and lexicographic product. Other examples of lattice composition are presented in [18,15].

Pair In Figure 3 we show the standard pair composition. The bottom is the pair of respective bottoms and the join is the coordinate-wise join of the components. This can be generalized to products of more than two components.

Lexicographic Pair A variation of the *pair* composition is to establish a *lexicographic pair*. In this construction, in Figure 4, the first element takes priority in establishing the outcome of the join, and a join of the second component is only performed on a tie. One important special case in when the first component is a total order; it can be used, e.g., to define an outcome based on the comparison of a time-stamp, as will be shown later.

7.2 Anonymous δ-CRDTs

The simplest CRDTs are anonymous. This occurs when the mutators do not make use of a globally unique replica identifier, having a uniform specification

$$\begin{aligned}
\mathsf{GSet}\langle E\rangle &= \mathcal{P}(E) \\
\bot &= \{\} \\
\mathsf{insert}^{\delta}_{i}(e, s) &= \{e\} \\
\mathsf{elements}(s) &= s \\
s \sqcup s' &= s \cup s'
\end{aligned}$$

Fig. 5: δ-CRDT grow-only set, replica i.

$$\begin{aligned}
\mathsf{2PSet}\langle E\rangle &= \mathcal{P}(E) \times \mathcal{P}(E) \\
\bot &= (\bot, \bot) \\
\mathsf{insert}^{\delta}_{i}(e, (s, t)) &= (\{e\}, \bot) \\
\mathsf{remove}^{\delta}_{i}(e, (s, t)) &= (\bot, \{e\}) \\
\mathsf{elements}((s, t)) &= s \setminus t \\
(s, t) \sqcup (s', t') &= (s \sqcup s', t \sqcup t')
\end{aligned}$$

Fig. 6: δ-CRDT two-phase set, replica i.

for all replicas. (Although for uniformity of notation we will keep parameterizing mutators by replica identifier.)

GSet A simple example is illustrated by a grow-only set, in Figure 5. The single delta mutator $\mathsf{insert}^{\delta}_{i}(e, s)$ does not even need to consider the current state of the replica, available in s, and simply produces a delta with a singleton set containing the element e to be added. This delta $\{e\}$ when joined to s produces the desired result: an inflated set $s \cup \{e\}$ that includes element e. The *join* of grow-only sets is trivially obtained by unioning the sets.

2PSet In case one needs to remove elements, there are multiple ways of addressing it. The simplest way is to include another (grow-only) set that gathers the removed elements. This is done in Figure 6, which shows a *two-phase set*, with state being a pair of sets. The name comes from the fact that elements may go through two phases: the *added* phase and the *removed* phase. The shortcoming of this simple design is that once removed, elements cannot be re-added.

If we look at the query function elements it is clear that the data-type is presenting to the user the set difference between the added elements and the removed elements (those stored in the tombstone set t). Removing an element simply adds it to the *removed* set. (A variant of 2PSet with *guarded removes* [19] only does so if the element is already present in the *added* set.) The join is simply a pairwise join.

$$\begin{aligned}
\mathsf{AWLWWSet}\langle E\rangle &= E \hookrightarrow \mathbb{N} \boxtimes \mathbb{B}\\
\bot &= \{\}\\
\mathsf{insert}^{\delta}_{i}(e,t,m) &= \{e \mapsto (t, \mathsf{True})\}\\
\mathsf{remove}^{\delta}_{i}(e,t,m) &= \{e \mapsto (t, \mathsf{False})\}\\
\mathsf{elements}(m) &= \{e \mid (e,(_,\mathsf{True})) \in m\}\\
m \sqcup m' &= \{e \mapsto m(e) \sqcup m'(e) \mid e \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}$$

Fig. 7: δ-CRDT Add-Wins LWW Set, replica i.

Add-Wins Last-Writer-Wins Set This construction, depicted in Figure 7, manages a set of elements of type E by tagging them with timestamps from some total order – here we use natural numbers. Each time an elements is added, it is tagged with a client supplied timestamp and the boolean True. Removed elements are similarly tagged, but with the boolean False. Elements marked with True are considered to be in the set. When joining two such sets, those elements in common will have to compete to define if they are in the set. By using lexicographic pairs, we obtain the behaviour that elements with higher (more recent) time-stamps will win, defining the presence according to the boolean tag; if there is a tie in the time-stamp, adds will win, since we order $\mathsf{False} < \mathsf{True}$.

Notice that is is up to the client to ensure that supplied timestamps always grow monotonically. Failure to do so is a common source of errors in timestamp based systems [20]. A dual construction to the *Add-Wins LWW Set* is a *Remove-Wins LWW Set*, where remove operations take priority on the event of a time-stamp tie. This construction has been widely deployed in production as part the SoundCloud system [6].

7.3 Named δ-CRDTs

Another design strategy for conflict-free data-types is to ensure that each replica only changes a specific part of the state. In Section 4, we defined a GCounter that, using a map from globally unique replica identifiers to natural numbers, keeps track of how many increment operations each replica did. This was the first example of a *named CRDT*, the construction covered in this section. The distinction from anonymous CRDTs is that mutations make use of the replica identifier i.

PNCounter By composing, in a pair, two grow-only counters we obtain a *positive-negative counter* that can track both increments and decrements. Shown in Figure 8, the increment and decrement operations will update the first and second components of the pair, respectively. As expected, the value is obtained by subtracting the decrements from the increments.

$$
\begin{aligned}
\mathsf{PNCounter} &= \mathsf{GCounter} \times \mathsf{GCounter} \\
\bot &= (\bot, \bot) \\
\mathsf{inc}^{\delta}_{i}((p, n)) &= (\mathsf{inc}^{\delta}_{i}(p), \bot) \\
\mathsf{dec}^{\delta}_{i}((p, n)) &= (\bot, \mathsf{inc}^{\delta}_{i}(n)) \\
\mathsf{value}((p, n)) &= \mathsf{value}(p) - \mathsf{value}(n) \\
(p, n) \sqcup (p', n') &= (p \sqcup p', n \sqcup n')
\end{aligned}
$$

Fig. 8: δ-CRDT positive-negative counter, replica i.

$$
\begin{aligned}
\mathsf{LexCounter} &= \mathbb{I} \hookrightarrow \mathbb{N} \boxtimes \mathbb{Z} \\
\bot &= \{\} \\
\mathsf{inc}^{\delta}_{i}(m) &= \{i \mapsto m(i) + (0, 1)\} \\
\mathsf{dec}^{\delta}_{i}(m) &= \{i \mapsto m(i) + (1, -1)\} \\
\mathsf{value}(m) &= \sum_{j \in \mathbb{I}} \mathsf{snd}\, m(j) \\
m \sqcup m' &= \{j \mapsto m(j) \sqcup m'(j) \mid j \in \mathsf{dom}\, m \cup \mathsf{dom}\, m'\}
\end{aligned}
$$

Fig. 9: δ-CRDT Lexicographic Counter, replica i.

Lexicographic Counter While the PNCounter was one of the first CRDTs to be added to a production database, in Riak 1.4 [21], the competing Cassandra database had its own counter implementations based on the LWW strategy. Interestingly it proved to be difficult to avoid semantic anomalies in the behaviour of those early counters, and since Cassandra 2.1, a new counter was introduced [22]. We capture its main properties in the Figure 9 specification of a LexCounter.

This counter is updated by either incrementing or decrementing the second component of the lexicographic pair corresponding to the replica issuing the mutation. Decrements also increment the first component, to ensure that the pair will be inflated, making it (and therefore, the just updated second component) win upon a lexicographic join.

7.4 Causal δ-CRDTs

We now introduce a specific class of CRDTs, that we will refer to as *causal CRDTs*. Initial designs [5] introduced data types such as *observed-remove sets* and *multi-value registers*. While these made possible sets which allow adding and removing elements multiple times, and to model the design of the eventually consistent shopping cart, in Amazon Dynamo [3], they had sub-optimal scalability properties [16]. Later designs, such as in *observed-remove sets without*

$$\begin{aligned}
\mathsf{CausalContext} &= \mathcal{P}(\mathbb{I} \times \mathbb{N}) \\
\mathsf{max}_i(c) &= \mathsf{max}(\{n \mid (i,n) \in c\} \cup \{0\}) \\
\mathsf{next}_i(c) &= (i, \mathsf{max}_i(c) + 1)
\end{aligned}$$

Fig. 10: Causal Context.

tombstones [23], allow an efficient management of meta-data state and can be applied to a broad class of data-types.

We introduce the concept of *dot store* to be used together with a *causal context* to form the state (a join-semilattice) of a causal CRDT, presenting three such dot stores and lattices. These are then used to obtain several related data-types, including flags, registers, sets, and maps.

Causal Context A common property to causal CRDTs is that events can be assigned unique identifiers. A simple mechanism is to create these identifiers by appending to a globally unique replica identifier a replica-unique integer. For instance, in replica $i \in \mathbb{I}$ we can create the sequence $(i, 1), (i, 2), \ldots$. Each of these pairs can be used to tag a specific event, or client action, and if we collect these pairs in a grow-only set, we can remember which events are known to each replica. The pair is called a *dot* and the grow-only set of pairs can be called a *causal history*, or alternatively a *causal context*, as we do here.

As seen in Figure 10, a causal context is a set of dots. We define two functions over causal contexts: $\mathsf{max}_i(c)$ gives the maximum sequence number for pairs in c from replica i, or 0 if there is no such dot; $\mathsf{next}_i(c)$ produces the next available sequence number for replica i given set of events in c.

Causal Context Compression In practice, a causal context can be efficiently compressed without any loss of information. When using an anti-entropy algorithm that provides causal consistency, e.g., Algorithm 2, then for each replica state X_i that includes a causal context c_i, and for any replica identifier $j \in \mathbb{I}$, we have a contiguous sequence:

$$1 \leq n \leq \mathsf{max}_j(c_i) \Rightarrow (j, n) \in c_i.$$

Thus, under causal consistency the causal context can always be encoded as a compact *version vector* [24] $\mathbb{I} \hookrightarrow \mathbb{N}$ that keeps the maximum sequence number for each replica.

Even under non-causal anti-entropy, such as in Algorithm 1, compression is still possible by keeping a version vector that encodes the initial contiguous sequence of dots from each replica, together with a set for the non-contiguous dots. As anti-entropy proceeds, each dot is eventually encoded in the vector, and thus the set remains typically small. Compression is less likely for the causal context of delta-groups in transit or buffered to be sent, but those contexts are

$$
\begin{aligned}
&\mathsf{DotStore}\\
\mathsf{dots}\langle S : \mathsf{DotStore}\rangle \;&:\; S \to \mathcal{P}(\mathbb{I} \times \mathbb{N})\\[1em]
\mathsf{DotSet} : \mathsf{DotStore} &= \mathcal{P}(\mathbb{I} \times \mathbb{N})\\
\mathsf{dots}(s) &= s\\[1em]
\mathsf{DotFun}\langle V : \mathsf{Lattice}\rangle : \mathsf{DotStore} &= \mathbb{I} \times \mathbb{N} \hookrightarrow V\\
\mathsf{dots}(s) &= \mathsf{dom}\, s\\[1em]
\mathsf{DotMap}\langle K,\ V : \mathsf{DotStore}\rangle : \mathsf{DotStore} &= K \hookrightarrow V\\
\mathsf{dots}(m) &= \bigcup\{\mathsf{dots}(v) \mid (_, v) \in m\}
\end{aligned}
$$

Fig. 11: Dot Stores.

only transient and smaller than those in the actual replica states. Moreover, the same techniques that encodes contiguous sequences of dots can also be used for transient context compression [25].

Dot Stores Together with a causal context, the state of a causal CRDT will use some kind of dot store, which acts as a container for data-type specific information. A dot store can be queried about the set of event identifiers (dots) corresponding to the relevant operations in the container, by function dots, which takes a dot store and returns a set of dots. In Figure 11 we define three kinds of dot stores: a DotSet is simply a set of dots; the generic $\mathsf{DotFun}\langle V\rangle$ is a map from dots to some lattice V; the generic $\mathsf{DotMap}\langle K, V\rangle$ is a map from some set K into some dot store V.

Causal δ-CRDTs In figure 12 we define the join-semilattice which serves as state for Causal δ-CRDTs, where an element is a pair of dot store and causal context. We define the join operation for each of the three kinds of dot stores. These lattices are a generalization of techniques introduced in [23,26]. To understand the meaning of a state (and the way join must behave), a dot present in a causal context but not in the corresponding dot store, means that the dot was present in the dot store, some time the past, but has been removed meanwhile. Therefore, the causal context can track operations with remove semantics, while avoiding the need for individual tombstones.

When joining two replicas, a dot present in only one dot store, but included in the causal context of the other, will be discarded. This is clear for the simpler case of a DotSet, where the join preserves all dots in common, together with those not present in the other causal context. The $\mathsf{DotFun}\langle V\rangle$ case is analogous,

$$\mathsf{Causal}\langle T : \mathsf{DotStore}\rangle = T \times \mathsf{CausalContext}$$

$$\sqcup \ : \ \mathsf{Causal}\langle T\rangle \times \mathsf{Causal}\langle T\rangle \to \mathsf{Causal}\langle T\rangle$$

$$\begin{aligned}
\textbf{when} \quad & T : \mathsf{DotSet} \\
(s, c) \sqcup (s', c') = {} & ((s \cap s') \cup (s \setminus c') \cup (s' \setminus c), c \cup c')
\end{aligned}$$

$$\begin{aligned}
\textbf{when} \quad & T : \mathsf{DotFun}\langle _ \rangle \\
(m, c) \sqcup (m', c') = {} & (\{k \mapsto m(k) \sqcup m'(k) \mid k \in \mathsf{dom}\, m \cap \mathsf{dom}\, m'\} \cup \\
& \ \{(d, v) \in m \mid d \notin c'\} \cup \{(d, v) \in m' \mid d \notin c\}, c \cup c')
\end{aligned}$$

$$\begin{aligned}
\textbf{when} \quad & T : \mathsf{DotMap}\langle _, _ \rangle \\
(m, c) \sqcup (m', c') = {} & (\{k \mapsto \mathsf{v}(k) \mid k \in \mathsf{dom}\, m \cup \mathsf{dom}\, m' \wedge \mathsf{v}(k) \neq \bot\}, c \cup c') \\
& \textbf{where } \mathsf{v}(k) = \mathsf{fst}\,((m(k), c) \sqcup (m'(k), c'))
\end{aligned}$$

Fig. 12: Lattice for Causal δ-CRDTs.

but the container is now a map from dots to some value, allowing the value for a given dot to evolve with time, independently at each replica. It assumes the value set is a join-semilattice, and applies the corresponding join of values for each dot in common.

In the more complex case of $\mathsf{DotMap}\langle K, V\rangle$, a map from some K to some dot store V, the join, for each key present in either replica, performs a join in the lattice $\mathsf{Causal}\langle V\rangle$, by pairing the per-key value with the replica-wide causal context, and storing the resulting value (first component of the result) for that key, but only when it is not $\bot_V$. This allows the disassociation of a composite embedded value from a key, with no need for a per-key tombstone, by remembering in the causal context all dots from the composite value. Matching our notation, in a $\mathsf{DotMap}\langle K, V\rangle$, any unmapped key corresponds effectively to the bottom $\bot_V$.

Enable-Wins Flag The flags are simple, yet useful, data-types that were first introduced in Riak 2.0 [17]. Figure 13 presents an *enable-wins flag*. Enabling the flag simply replaces all dots in the store by a new dot; this is achieved by obtaining the dot through $\mathsf{next}_i(c)$, and making the delta mutator return a store containing just the new dot, together with a causal context containing both the new dot and all current dots in the store; this will make all current dots to be removed from the store upon a join (as previously defined), while the new dot is added. Concurrent enabling can lead to the store containing several dots. Reads will consider the flag enabled if the store is not an empty set. Disabling is similar to enabling, in that all current dots are removed from the store, but no new dot

$$\begin{aligned}
\mathsf{EWFlag} &= \mathsf{Causal}\langle\mathsf{DotSet}\rangle \\
\mathsf{enable}^{\delta}_i((s,c)) &= (d, d \cup s) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{disable}^{\delta}_i((s,c)) &= (\{\}, s) \\
\mathsf{read}((s,c)) &= s \neq \{\}
\end{aligned}$$

Fig. 13: δ-CRDT Enable-wins Flag, replica i.

$$\begin{aligned}
\mathsf{MVRegister}\langle V\rangle &= \mathsf{Causal}\langle\mathsf{DotFun}\langle V\rangle\rangle \\
\mathsf{write}^{\delta}_i(v, (m,c)) &= (\{d \mapsto v\}, \{d\} \cup \mathsf{dom}\, m) \quad \textbf{where } d = \mathsf{next}_i(c) \\
\mathsf{clear}^{\delta}_i((m,c)) &= (\{\}, \mathsf{dom}\, m) \\
\mathsf{read}((m,c)) &= \mathsf{ran}\, m
\end{aligned}$$

Fig. 14: δ-CRDT Multi-value register, replica i.

is added. It is possible to construct a dual data-type with *disable-wins* semantics and its code is also available [14].

Multi-Value Register A *multi-value register* supports read and write operations, with traditional sequential semantics. Under concurrent writes, a join makes a subsequent read return all concurrently written values, and a subsequent write will overwrite all those values. This data-type captures the semantics of the Amazon shopping cart [3], and the usual operation of Riak (when not using CRDT data-types). Initial implementations of these registers tagged each value with a full version vector [5]; here we introduce an optimized implementation that tags each value with a single dot, by using a $\mathsf{DotFun}\langle V\rangle$ as dot store. In Figure 14 we can see that the write delta mutator returns a causal context with all dots in the store, so that they are removed upon join, together with a single mapping from a new dot to the value written; as usual, the new dot is also put in the context. A clear operation simply removes current dots, leaving the register in the initial empty state. Reading simply returns all values mapped in the store.

Add-Wins Set In an *add-wins set* removals do not affect elements that have been concurrently added. In this sense, under concurrent updates, an add will win over a remove of the same element. The implementation, in Figure 15, uses a map from elements to sets of dots as dot store. This data-type can be seen as a map from elements to enable-wins flags, but with a single common causal context, and keeping only elements mapped to an enabled flag.

When an element is added, all dots in the corresponding entry will be replaced by a singleton set containing a new dot. If a DotSet for some element were to become empty, such as when removing the element, the join for $\mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle$

$$
\begin{aligned}
\mathsf{AWSet}\langle E\rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotSet}\rangle\rangle \\
\mathsf{add}^{\delta}_i(e,(m,c)) &= (\{e \mapsto d\}, d \cup m(e)) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{remove}^{\delta}_i(e,(m,c)) &= (\{\}, m(e)) \\
\mathsf{clear}^{\delta}_i((m,c)) &= (\{\}, \mathsf{dots}(m)) \\
\mathsf{elements}((m,c)) &= \mathsf{dom}\, m
\end{aligned}
$$

Fig. 15: δ-CRDT Add-wins set, replica i.

$$
\begin{aligned}
\mathsf{RWSet}\langle E\rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle E, \mathsf{DotMap}\langle \mathbb{B}, \mathsf{DotSet}\rangle\rangle\rangle \\
\mathsf{add}^{\delta}_i(e,(m,c)) &= (\{e \mapsto \{\mathsf{True} \mapsto d\}\}, d \cup \mathsf{dots}(m(e))) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{remove}^{\delta}_i(e,(m,c)) &= (\{e \mapsto \{\mathsf{False} \mapsto d\}\}, d \cup \mathsf{dots}(m(e))) \quad \textbf{where } d = \{\mathsf{next}_i(c)\} \\
\mathsf{clear}^{\delta}_i((m,c)) &= (\{\}, \mathsf{dots}(m)) \\
\mathsf{elements}((m,c)) &= \{e \in \mathsf{dom}\, m \mid \mathsf{False} \notin \mathsf{dom}\, m(e)\}
\end{aligned}
$$

Fig. 16: δ-CRDT Remove-wins set, replica i.

will remove the entry from the resulting map. Concurrently created dots are preserved when joining. The clear delta mutator will put all dots from the dot store in the causal context, to be removed when joining. As only non-empty entries are kept in the map, the set of elements corresponds to the map domain.

Remove-Wins Set Under concurrent adds and removes of the same element, a *remove-wins set* will make removes win. To obtain this behaviour, the implementation in Figure 16 uses a map from elements to a nested map from booleans to sets of dots. For both adding and removing of a given entry, the corresponding nested map is cleared (by the delta mutator inserting all corresponding dots into the causal context), and a new mapping from either True or False, respectively, to a singleton new dot is added.

When joining replicas, the nested map will collect the union of the respective sets in the corresponding entry (for dots not seen by the other causal context). As before, only non-bottom entries are kept, for both outer map (non-empty maps) and nested map (non-empty DotSets). Therefore, an element is considered to be in the set if it belongs to the outer map domain, and the corresponding nested map does not contain a False entry; thus, concurrent removes will win over adds.

A Map Embedding Causal δ-CRDTs. Maps are important composition tools for the construction of complex CRDTs. Although grow-only maps are simple to conceive and have been used in early state based designs [10], the creation of a map that allows removal of entries and supports recursive composition is not trivial. Riak 2.0 introduced a map design that provides a clear

$$
\begin{aligned}
\mathsf{ORMap}\langle K, \mathsf{Causal}\langle V\rangle\rangle &= \mathsf{Causal}\langle \mathsf{DotMap}\langle K, V\rangle\rangle \\
\mathsf{apply}^\delta_i(o^\delta_i, k, (m, c)) &= (\{k \mapsto v\}, c') \quad \textbf{where } (v, c') = o^\delta_i((m(k), c)) \\
\mathsf{remove}^\delta_i(k, (m, c)) &= (\{\}, \mathsf{dots}(m(k))) \\
\mathsf{clear}^\delta_i((m, c)) &= (\{\}, \mathsf{dots}(m))
\end{aligned}
$$

Fig. 17: δ-CRDT Map embedding Causal δ-CRDTs, with observed removes, replica i.

observed-remove semantics: a remove can be seen as an "undo" of all operations leading to the embedded value, putting it in the bottom state, but remembering those operations, to undo them in other replicas which observe it by a join. Key to the design is to enable removal of keys to affect (and remember) the dots in the associated nested CRDT, to allow joining with replicas that have concurrently evolved from the before-removal point, or to ensure that re-creating entries previously removed does not introduce anomalies.

In order to obtain the desired semantics it is not possible to simply map keys to causal CRDTs having their own causal contexts. Doing so would introduce anomalies when recreating keys, since old versions of the mappings in other replicas could be considered more recent than newer mappings, since the causal contexts of the re-created entries would start again at their bottom state. The solution is to have a common causal context to the whole map, to be used for all nested components, and never reset that single context.

For an arbitrary set of keys K and a causal δ-CRDT $\mathsf{Causal}\langle V\rangle$ that we want to embed (including, recursively, the map we are defining), the desired map can be achieved through $\mathsf{Causal}\langle \mathsf{DotMap}\langle K, V\rangle\rangle$, where a single causal context is shared by all keys and corresponding nested CRDTs, as presented in Figure 17. This map can embed any causal CRDT as values. For instance we can define a map of type $\mathsf{ORMap}\langle S, \mathsf{AWSet}\langle E\rangle\rangle$, mapping strings S to add-wins sets of elements E; or define a more complex recursive structure that uses a map within a map $\mathsf{ORMap}\langle \mathbb{N}, \mathsf{ORMap}\langle S, \mathsf{MVReg}\langle E\rangle\rangle\rangle$.

The map does not support a specific operation to add new entries: it starts as an empty map, which corresponds to any key implicitly mapped to bottom; then, any operation from the embedded type can be applied, through a higher-order apply, which takes a delta mutator o^δ_i to be applied, the key k, and the map (m, c). This mutator fetches the value at key k from m, pairs it with the shared causal context c, obtaining a value from the embedded type, and invokes the operation over the pair; from the resulting pair, it extracts the value to create a new mapping for that key, which it pairs with the resulting causal context. Removing a key will recursively remove the dots in the corresponding embedded value, while the clear operation will remove all dots from the store. This simplicity was achieved by encapsulating most complexity in the join (and also the dots function) of the embedded type.

8 Related Work

8.1 Eventually convergent data types.

The design of replicated systems that are always available and eventually converge can be traced back to historical designs in [27,28], among others. More recently, replicated data types that always eventually converge, both by reliably broadcasting operations (called operation-based) or gossiping and merging states (called state-based), have been formalized as CRDTs [9,10,4,5]. These are also closely related to Bloom^L [29] and Cloud Types [30]. State join-semilattices were used for deterministic parallel programming in LVars [31], where variables progress in the lattice order by joining other values, and are only accessible by special threshold reads.

8.2 Message size.

A key feature of δ-CRDT is message size reduction and coalescing, using small-sized deltas. The general old idea of using differences between things, called "deltas" in many contexts, can lead to many designs, depending on how exactly a delta is defined. The state-based deltas introduced for Computational CRDTs [32] require an extra delta-specific merge (in addition to the standard join) which does not ensure idempotence. In [33], an improved synchronization method for non-optimized OR-set CRDT [4] is presented, where delta information is propagated; in that paper deltas are a collection of items (related to update events between synchronizations), manipulated and merged through a protocol, as opposed to normal states in the semilattice. No generic framework is defined (that could encompass other data types) and the protocol requires several communication steps to compute the information to exchange. Operation-based CRDTs [4,5,34] also support small message sizes, and in particular, *pure* flavors [34] that restrict messages to the operation name, and possible arguments. Though pure operation-based CRDTs allow for compact states and are very fast at the source (since operations are broadcast without consulting the local state), the model requires more systems guarantees than δ-CRDT do, e.g., exactly-once reliable delivery and membership information, and impose more complex integration of new replicas. The work in [35] shows a different trade-off among state deltas and pure operations, by tagging operations and creating a globally stable log of operations while allowing local transient logs to preserve availability. While having other advantages, the creation of this global log requires more coordination than our gossip approach for causally consistent delta dissemination, and can stall dissemination.

8.3 Encoding causal histories.

State-based CRDT are always designed to be causally consistent [10,5]. Optimized implementations of sets, maps, and multi-value registers can build on this assumption to keep the meta-data small [16]. In δ-CRDT, however, deltas and

delta-groups are normally not causally consistent, and thus the design of *join*, the meta-data state, as well as the anti-entropy algorithm used must ensure this. Without causal consistency, the causal context in δ-CRDT can not always be summarized with version vectors, and consequently, techniques that allow for gaps are often used. A well known mechanism that allows for encoding of gaps is found in Concise Version Vectors [36]. Interval Version Vectors [25], later on, introduced an encoding that optimizes sequences and allows gaps, while preserving efficiency when gaps are absent.

9 Conclusion

We introduced the new concept of δ-CRDTs and devised *delta-mutators* over state-based datatypes which can detach the changes that an operation induces on the state. This brings a significant performance gain as it allows only shipping small states, i.e., *deltas*, instead of the entire state. The significant property in δ-CRDT is that it preserves the crucial properties (idempotence, associativity and commutativity) of standard state-based CRDT. In addition, we have shown how δ-CRDT can achieve causal consistency; and we presented an anti-entropy algorithm that allows replacing classical state-based CRDTs by more efficient ones, while preserving their properties. As an application of our approach we designed several novel δ-CRDT specifications, including a general framework for causal CRDTs and composition in maps.

Our approach is more relaxed than classical state-based CRDTs, and thus, can replace them without losing their power since δ-CRDT allows shipping delta-states as well as the entire state. Another interesting observation is that δ-CRDT can mimic the behavior of operation-based CRDTs, by shipping individual deltas on the fly but with weaker guarantees from the dissemination layer.

References

1. Cribbs, S., Brown, R.: Data structures in Riak. In: Riak Conference (RICON), San Francisco, CA, USA (oct 2012)
2. Terry, D.B., Theimer, M.M., Petersen, K., Demers, A.J., Spreitzer, M.J., Hauser, C.H.: Managing update conflicts in Bayou, a weakly connected replicated storage system. In: Symp. on Op. Sys. Principles (SOSP), Copper Mountain, CO, USA, ACM SIGOPS, ACM Press (December 1995) 172–182
3. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Symp. on Op. Sys. Principles (SOSP). Volume 41 of Operating Systems Review., Stevenson, Washington, USA, Assoc. for Computing Machinery (October 2007) 205–220
4. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: A comprehensive study of Convergent and Commutative Replicated Data Types. Rapp. Rech. 7506, INRIA, Rocquencourt, France (January 2011)
5. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M.: Conflict-free replicated data types. In Défago, X., Petit, F., Villain, V., eds.: Int. Symp. on Stabilization, Safety,

and Security of Distributed Systems (SSS). Volume 6976 of Lecture Notes in Comp. Sc., Grenoble, France, Springer-Verlag (October 2011) 386–400

6. Peter Bourgon: Consistency without Consensus: CRDTs in Production at SoundCloud. URL http://www.infoq.com/presentations/crdt-soundcloud (Retrieved 22-dec-2015)
7. Todd Hoff: How League of Legends Scaled Chat to 70 Million Players - It akes a lot of Minions. URL http://highscalability.com/blog/2014/10/13 (Retrieved 22-dec-2015)
8. Michael Owen: Using Erlan, Riak and the ORSWOT CRDT at bet365. URL http://www.erlang-factory.com/euc2015/michael-owen
9. Letia, M., Preguiça, N., Shapiro, M.: CRDTs: Consistency without concurrency control. Rapp. Rech. RR-6956, INRIA, Rocquencourt, France (June 2009)
10. Baquero, C., Moura, F.: Using structural characteristics for autonomous operation. Operating Systems Review **33**(4) (1999) 90–96
11. Helland, P.: Idempotence is not a medical condition. Queue **10**(4) (April 2012) 30:30–30:46
12. Brown, R., Cribbs, S., Meiklejohn, C., Elliott, S.: Riak dt map: A composable, convergent replicated dictionary. In: Proceedings of the First Workshop on Principles and Practice of Eventual Consistency. PaPEC '14, New York, NY, USA, ACM (2014) 1:1–1:1
13. Almeida, P.S., Shoker, A., Baquero, C.: Efficient state-based crdts by delta-mutation. In: Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015. (2015)
14. Baquero, C.: Delta-enabled-crdts. URL http://github.com/CBaquero/delta-enabled-crdts (Retrieved 22-dec-2015)
15. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order (2. ed.). Cambridge University Press (2002)
16. Burckhardt, S., Gotsman, A., Yang, H., Zawirski, M.: Replicated data types: specification, verification, optimality. In Jagannathan, S., Sewell, P., eds.: POPL, ACM (2014) 271–284
17. Basho: Riak datatypes. URL http://github.com/basho (Retrieved 22-dec-2015)
18. Kemme, B., Schiper, A., Ramalingam, G., Shapiro, M.: Dagstuhl seminar review: Consistency in distributed systems. SIGACT News **45**(1) (March 2014) 67–89
19. Zeller, P., Bieniusa, A., Poetzsch-Heffter, A.: Formal specification and verification of crdts. In Ábrahám, E., Palamidessi, C., eds.: Formal Techniques for Distributed Objects, Components, and Systems - 34th IFIP WG 6.1 International Conference, FORTE 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014. Proceedings. Volume 8461 of Lecture Notes in Computer Science., Springer (2014) 33–48
20. Kyle Kingsbury: The trouble with timestamps. URL https://aphyr.com/posts/299-the-trouble-with-timestamps
21. Basho: Riak 1.4. URL https://github.com/basho/riak/blob/1.4/RELEASE-NOTES.md (Retrieved 4-jan-2016)
22. Datatax: Whats New in Cassandra 2.1: Better Implementation of Counters. URL http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counte (Retrieved 4-jan-2016)
23. Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., Duarte, S.: An optimized conflict-free replicated set. Rapp. Rech. RR-8083, INRIA, Rocquencourt, France (October 2012)

24. Parker, D.S., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C.: Detection of mutual inconsistency in distributed systems. IEEE Trans. Softw. Eng. **9**(3) (May 1983) 240–247
25. Mukund, M., R., G.S., Suresh, S.P.: Optimized or-sets without ordering constraints. In: Proceedings ot the International Conference on Distributed Computing and Networking, New York, NY, USA, ACM (2014) 227241
26. Almeida, P.S., Baquero, C., Gonçalves, R., Preguiça, N.M., Fonte, V.: Scalable and accurate causality tracking for eventually consistent stores. In Magoutis, K., Pietzuch, P., eds.: Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings. Volume 8460 of Lecture Notes in Computer Science., Springer (2014) 67–81
27. Wuu, G.T.J., Bernstein, A.J.: Efficient solutions to the replicated log and dictionary problems. In: Symp. on Principles of Dist. Comp. (PODC), Vancouver, BC, Canada (August 1984) 233–242
28. Johnson, P.R., Thomas, R.H.: The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute (January 1976)
29. Conway, N., Marczak, W.R., Alvaro, P., Hellerstein, J.M., Maier, D.: Logic and lattices for distributed programming. In: Proceedings of the Third ACM Symposium on Cloud Computing, ACM (2012) 1
30. Burckhardt, S., Fähndrich, M., Leijen, D., Wood, B.P.: Cloud types for eventual consistency. In: ECOOP 2012–Object-Oriented Programming. Springer (2012) 283–307
31. Kuper, L., Newton, R.R.: Lvars: lattice-based data structures for deterministic parallelism. In: Proceedings of the 2nd ACM SIGPLAN workshop on Functional high-performance computing, ACM (2013) 71–84
32. Navalho, D., Duarte, S., Preguiça, N., Shapiro, M.: Incremental stream processing using computational conflict-free replicated data types. In: Proceedings of the 3rd International Workshop on Cloud Data and Platforms, ACM (2013) 31–36
33. Deftu, A., Griebsch, J.: A scalable conflict-free replicated set data type. In: Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems. ICDCS '13, Washington, DC, USA, IEEE Computer Society (2013) 186–195
34. Baquero, C., Almeida, P.S., Shoker, A.: Making operation-based CRDTs operation-based. In: Proceedings of Distributed Applications and Interoperable Systems: 14th IFIP WG 6.1 International Conference, Springer (2014)
35. Burckhardt, S., Leijen, D., Fahndrich, M.: Cloud types: Robust abstractions for replicated shared state. Technical Report MSR-TR-2014-43 (March 2014)
36. Malkhi, D., Terry, D.: Concise version vectors in winfs. Distributed Computing **20**(3) (2007) 209–219

7.4 Seyed H. Haeri (Hossein), Peter Van Roy, Carlos Baquero, and Christopher Meiklejohn. Deduction with partial knowledge about causality. Submitted for publication, 2016.

Worlds of Events
Deduction with Partial Knowledge about Causality

Seyed H. Haeri (Hossein)[1], Peter Van Roy[1], Carlos Baquero[2], and Christopher Meiklejohn[1]

[1] Université catholique de Louvain, Belgium
[2] Universidade do Minho, Portugal

Abstract. Interactions between internet users are mediated by their devices and the common support infrastructure in data centres. Keeping track of causality amongst actions, that take place in this distributed system, is key to provide a seamless interaction where effects follow causes. Tracking causality in large scale interactions is difficult due to the cost of keeping large quantities of metadata; even more challenging when dealing with resource-limited devices. In this paper, we focus on keeping partial knowledge on causality and address deduction from that knowledge.
We provide the first proof-theoretic causality modelling for distributed partial knowledge. We prove computability and consistency results. We also prove that the partial knowledge gives rise to a weaker model than classical causality. We provide rules for offline deduction about causality and refute some related folklore. We define two notions of bisimilarity between devices, using which we prove two important results. Namely, no matter the order of addition/removal, two devices deduce similarly about causality so long as: (1) the same causal information is fed to both. (2) they start bisimilar and erase the same causal information. Thanks to our establishment of bisimilarity, proofs of the latter two results work by simple induction on length.

1 Introduction

Causality [15,19] is an essential for our perception of the physical world, and of our relations to other entities. If one puts a cup on a table, and looks back at it, one expects it to be there. One also expects to get a reply to one's postcards, **after** they were sent, and not before.

Given the fault-tolerance and high availability expected of internet-based services today, distributed algorithms have become ubiquitous. One duty of these algorithms is to order events totally across multiple replicas of a service. This total order is required to ensure computation determinism; given the requirement of having multiple replicas appear as a single system, each replica must implement a state machine which observes the same events in the same order [18]. However, because of the amount of coordination required, a total order in the entire distributed system is not always feasible while maintaining availability [12].

Given the intractability of a total order, techniques that favour a partial order based on causality are explored for they express user's **intent**. For a key-value

store, one's writes may, e.g., be directed to one replica, and, subsequent reads served from a replica which has not yet received those writes. If we consider the canonical example of an access control list for viewing photos [16], one would expect that a write operation removing Eve from having access to Alice's photos prior to Alice uploading a photo she did not want Eve to see, would be observed in this order by Eve when performing read operations.

However, tracking causality can be very expensive in terms of metadata size; more so when interactions amongst many distinct entities is targeted [7]. Devising scalable solutions to causality tracking is a demanding problem [16,20] to the extent that some solutions even accept to lose causal information by pruning metadata [8]. Be it because limited resource is available (say to an edge device) or because a replica (say in a data centre) is temporarily out-of-sync, only a partial view of the system causality might be available. There is not much study on dealing with that partiality of knowledge, however. In this paper, we address that problem via a proof-theoretic modelling for partial causality knowledge of distributed systems. Partiality is not a loss in our model: What is not stored might be deducible – acting in favour of metadata size reduction.

Contributions of this paper are as follows:

1. We model distributed causality such that the holistic system and the partial causal knowledge of a device are categorically distinct (Definitions 1 and 2).
2. We offer rules for deducing causality when a device is online (Definition 4) and prove its computability (Theorem 1) and consistency (Theorem 2).
3. We show that deduction of causality with partial knowledge is strictly less accurate than the holistic causal knowledge (Lemma 2) and that the deductions of different devices do not conflict (Corollary 1).
4. We offer rules for a device to deduce causal information independent of new causal data from outside, e.g., when offline (Definition 6) and prove its consistency with the online rules (Lemmata 3 and 4). We also prove a related folklore wrong (Lemma 5) using the latter machinery.
5. We craft a notion of bisimilarity (Definition 7) and prove that the order of arrival of new causal data is insignificant for bisimilar devices (Theorem 4).
6. We craft another notion of bisimilarity (Definition 11) to prove that the order of removal of causal data is insignificant for bisimilar devices (Theorem 7).

Unlike traditional approaches to causality modelling that store a partial order of known causally related events and consider non related events as concurrent events, we explicitly model concurrency information and provide a broader spectrum of relations amongst events.

2 Worlds of Events and Microcosms

Call a binary relation R a strict partial order on a set P when R is irreflexive, asymmetric, and transitive. In such a case, we say that (P, R) is a strict poset. For a strict poset (T, R), when R is also total, call (T, R) a strict chain. Let R be a relation on a set S. For a subset U of S, the symbol $R_{|U}$ denotes R restricted

to U. We use "$\veebar$" for the exclusive or of mathematical logic. Throughout this paper, "_" is our wild card; its usage expresses our lack of interest in the exact details of what "_" has replaced.

Definition 1. *Call $W(<, \|)$ a world of events when:*

- ***(W1)*** *W is an infinitely countable set (i.e., $|W| = \aleph_0$) of events that are ranged over by $e_1, e_2, \ldots, e, e', \ldots,$*
- ***(W2)*** *$<$ and $\|$ are binary relations defined on W that are ranged over by $r_1, r_2, \ldots, r, r', \ldots,$*
- ***(W3)*** *$(W, <)$ is a strict poset,*
- ***(W4)*** *$\|$ is anti-reflexive and symmetric, and*
- ***(W5)*** *$e_1 \neq e_2$ iff $e_1 \| e_2 \veebar e_1 < e_2 \veebar e_2 < e_1$.*

When $e_1\ r\ e_2$ is known to hold for W, we write $W \vDash e_1\ r\ e_2$.

The relations $<$ and $\|$ denote the familiar happens-before and is-concurrent-with, respectively [15]. For a world of events, we take the relation $<>$ (read is-causally-related-to) as a syntactic sugar for $< \cup <^{-1}$, namely, $e_1 <> e_2 \stackrel{\text{def}}{=} e_1 < e_2 \vee e_2 < e_1$. Hence, $<>$ is symmetric.

Fix the set of **accurate** relations $R = \{<, \|\}$. The relation $<>$ is an **inaccurate** relation in that it does not expose the exact known direction of $<$. We now extend $\vDash$ to $\vDash^*$ for when the inaccurate relation $<>$ is also needed to be taken into consideration. Write $W \vDash^* e_1\ r\ e_2$ iff: $W \vDash e_1\ r\ e_2$; or, $r = <>$ and either $W \vDash e_1 < e_2$ or $W \vDash e_2 < e_1$. Note that, unlike $\vDash$, not every distinct pair of events are attributed to a unique relation by $\vDash^*$. In particular, for every e_1 and e_2 such that $W \vDash e_1 < e_2$, by definition, it is both the case that $W \vDash^* e_1 < e_2$ and $W \vDash^* e_1 <> e_2$. We call $e_1\ r\ e_2$ a correspondence, ranged over by $c_1, c_2, \ldots, c, c', \ldots$ For a world of events W, we also fix $\mathcal{C}^*_W = \{c \mid W \vDash^* c\}$. For $c = e_1\ r\ e_2$, we say c is an accurate correspondence when r is accurate. We call c inaccurate otherwise.

Proposition 1. *Every world of events W is consistent: $W \vDash e_1\ r\ e_2$ and $W \vDash e_1\ r'\ e_2$ imply $r = r'$.*

Definition 2. *Let $W(<, \|)$ be a world of events. Call $M(I, E)$ a microcosm of W (write $M \lhd W$) when:*

- ***(M1)*** *$I \subset W$ and $|I| < \aleph_0$,*
- ***(M2)*** *$(I, <_{|I})$ is a strict chain,*
- ***(M3)*** *$E \subset \mathcal{C}^*_W$ and $|E| < \aleph_0$,*
- ***(M4)*** *$e_1\ r\ e_2 \in E$ implies that there is no chain of events $e'_1, \ldots, e'_n$ in M such that $e_1 < e'_1 < \cdots < e'_n < e_2$.*

Accordingly, call W the enclosing world of M and let $\mathcal{M}_W = \{M \mid M \lhd W\}$.

For an $M(I, E)$, we refer to I as the **internal** events of M, and, to E as the set of **external** correspondences known to it. When appropriate, we use the alternative notions $I(M)$ and $E(M)$, respectively. Write $e_1 < e_2 \in I$ when

$e_1, e_2 \in I$ and $W \vDash e_1 < e_2$. Besides, write $e_1\ r\ e_2 \in M$, when $e_1\ r\ e_2 \in I$ or $e_1\ r\ e_2 \in E$. Write $e \in E$ when $\exists e' \in W.\ e _ e' \in E \vee e' _ e \in E$. Finally, write $e \in M$ when $e \in I$ or $e \in E$. That is how we formalise the notion of microcosm membership informally used in (M4).

Microcosms are our abstraction for a generic device. The enclosing world of events of a microcosm is the abstraction we use for the ecosystem in which a device lives. Certain events can take place locally for a device; in which case, they are stored in the internal events of the respective microcosm. The correspondence between certain events can also be disclosed to a device by the ecosystem; in which case, they are stored in the external correspondences of the respective microcosm. In our model, devices do not get to communicate directly with one another. The ecosystem sits between devices in that news from other devices in the same ecosystem arrives via the ecosystem (as opposed to the other devices themselves).

$$\boxed{M \vdash^{\circ} e_1\ r\ e_2} \qquad \text{where } r \in R \cup \{<>\}$$

$$\frac{e_1\ r\ e_2 \in M}{M \vdash^{\circ} e_1\ r\ e_2}\ (\textsc{Init})$$

$$\frac{M \vdash^{\circ} e_1 < e_2 \quad M \vdash^{\circ} e_2 < e_3}{M \vdash^{\circ} e_1 < e_3}\ (\textsc{In-Tr})$$

Fig. 1: Microcosm Initial Judgements

Note that, unlike a world of events, for a microcosm, the relation $<>$ is not a syntactic sugar. To the latter, an $<>$ instance is all the information that is given for the respective pair of events. In that case, whilst no stronger information about the given pair is provided to the microcosm, the enclosing world of events is aware of the exact $<$ direction between the pair. It, nevertheless, follows from anti-reflexiveness of $<$ that $<>$ is anti-reflexive too – both for worlds of events and their microcosms.

We now introduce the first sort of deduction for microcosms (Definition 3). The idea is that such a deduction is for a microcosm to decree on the correspondences it knows of. Later in Section 3, we will generalise deduction for a microcosm to also conclude that it does not know the correspondence between a given pair of events.

Definition 3. *Let M be a microcosm. Judgements of the form $M \vdash^{\circ} e_1\ r\ e_2$ are called the initial judgements of M when they are derived using the rules (*Init*) and (*In-Tr*) in Figure 1. Write $M \nvdash^{\circ} e_1\ r\ e_2$ when $M \vdash^{\circ} e_1\ r\ e_2$ is not true.*

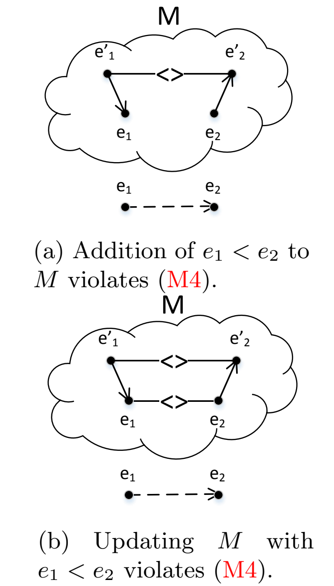
Note that with "$_$" being existential in nature, the negation acts universally. In particular, $M \nvdash^{\circ} e_1 _ e_2$ stipulates the lack of *any* initial correspondence between e_1 and e_2 in M.

Here is an informal account of the rules in Fig. 1: (Init) states that every piece of information that is initially provided to a microcosm is reliable in the initial judgements made inside that microcosm. (In-Tr) legislates transitivity of $<$ for initial judgements (regardless of whether the premises come from internal or external knowledge of a microcosm or a combination of those).

In this paper, we let $\Pi, \Pi', \ldots, \Pi_1, \Pi_2, \ldots$ range over derivation trees. For a derivation tree Π, we write $\mathsf{lr}(\Pi)$ for the last rule used in Π.

There are two possible ways a microcosm can evolve upon receipt of new information: (Section 3 gives more details about the intuition and the semantics of the two evolution mechanisms.)

For a microcosm M, when $M \not\vdash^{\circ} e_1 _ e_2$, write $(M + e_1\ r\ e_2)$ for the microcosm M with the additional information $e_1\ r\ e_2$. We assume that $e_1\ r\ e_2$ is known to be internal or external to the resulting microcosm. Likewise, when $M \vdash^{\circ} e_1 <> e_2$ (or $M \vdash^{\circ} e_2 <> e_1$), define $M[e_1 < e_2]$ for the microcosm M in which $e_1 < e_2$ replaces $e_1 <> e_2$ (or $e_2 <> e_1$). For both addition – namely, $(M + e_1\ r\ e_2)$ – and update – namely, $M[e_1 < e_2]$ – we assume that the change will not violate (M4). See Fig. 2a and 2b for when careless addition and update violate (M4).



(a) Addition of $e_1 < e_2$ to M violates (M4).

(b) Updating M with $e_1 < e_2$ violates (M4).

Fig. 2: Illegal for $e'_1 < e_1 < e_2 < e'_2$ whilst $e'_1 <> e'_2 \in M$

3 Online Decision Making

This section provides an algorithm for a microcosm to issue its verdict on the relation it can deduce, to the best of its knowledge, to hold between a queried pair of (distinct) events. This algorithm (manifested in Fig. 3) is called the **online** decision making procedure. The idea is that the decision accuracy keeps improving using this procedure upon the inflow of the new or updated correspondences. In crude terms, this is the situation where the device is connected and thus online. Contrast this with what comes in Section 4. We prove computability (Theorem 1) and consistency (Theorem 2) of the online decision making. We show that the causal knowledge of a microcosm is strictly less than its enclosing world of events (Lemma 2), there is no conflict between the verdict of two microcosms of the same world of events – even when they do not issue the exact same correspondence (Corollary 1).

Definition 4. *Define the online decision making procedure of a microcosm using the rules of the scheme* $M \vdash e_1\ r\ e_2$ *shown in Fig. 3, where* $r \in R \cup \{<>, ?\}$.

In Fig. 3, a judgement $M \vdash e_1\ ?\ e_2$ stipulates the lack of knowledge "in M" about the correspondence between the pair of events e_1 and e_2. As such, ? (read is-unknown-to) is another inaccurate relation. Note that, unlike $<>$, the relation ? is only available for microcosms. Recall that, as axiomatised by (W5), the correspondence between every two distinct pair of events is known to their enclosing world of events.

Here is an informal account of the rules in Fig. 3:

(In-OK) says online decision making approves of every initial judgement. The rest of the first row as well as the next two rows concern when a microcosm

$$\boxed{M \vdash e_1\ r\ e_2} \qquad \text{where } r \in R \cup \{<>, ?\}$$

$$\frac{M \vdash^{\circ} e_1\ r\ e_2}{M \vdash e_1\ r\ e_2}\ (\textsc{In-OK}) \qquad \frac{M \vdash e_1\ r\ e_2 \quad M \vdash e_2\ r\ e_3 \quad r \in \{\|, <>, ?\} \quad M \nvdash^{\circ} e_1\ _\ e_3}{M \vdash e_1\ ?\ e_3}\ (\textsc{Un-1})$$

$$\frac{M \nvdash^{\circ} e_1\ _\ e_2 \quad \nexists e' \in M.\ (M \vdash e_1\ _\ e') \wedge (M \vdash e'\ _\ e_2)}{M \vdash e_1\ ?\ e_2}\ (\textsc{Un-3})$$

$$\frac{M \vdash e_1\ r\ e_2 \quad M \vdash e_2\ ?\ e_3 \quad r \in R \cup \{<>\} \quad M \nvdash^{\circ} e_1\ _\ e_3}{M \vdash e_1\ ?\ e_3}\ (\textsc{Un-2}) \qquad \frac{e \notin M \quad e \neq e'}{M \vdash e\ ?\ e'}\ (\textsc{Un-4})$$

$$\frac{M \vdash e_1 \parallel e_2}{M \vdash e_2 \parallel e_1}\ (\textsc{Co-Sym}) \qquad \frac{M \vdash e_1 <> e_2}{M \vdash e_2 <> e_1}\ (\text{CR-}\textsc{Sym}) \qquad \frac{M \vdash e_1\ ?\ e_2}{M \vdash e_2\ ?\ e_1}\ (\textsc{Un-Sym})$$

$$\frac{M \vdash e_1\ ?\ e_2 \quad M \lhd W \quad W \vDash^{*} e_1\ r\ e_2}{(M + e_1\ r\ e_2) \vdash e_1\ r\ e_2}\ (\textsc{Strng})$$

$$\frac{M \vdash e_1\ r\ e_2 \quad r \neq ? \quad M \lhd W \quad W \vDash^{*} e'_1\ r'\ e'_2}{(M + e'_1\ r'\ e'_2) \vdash e_1\ r\ e_2}\ (\textsc{Weak})$$

$$\frac{M \lhd W \quad M \vdash e_1 <> e_2 \quad W \vDash e_1 < e_2}{M[e_1 < e_2] \vdash e_1 < e_2}\ (\textsc{Up-S})$$

$$\frac{M \lhd W \quad W \vDash e_1 < e_2 \quad M \vdash e_1 <> e_2 \quad M \vdash e'_1\ r'\ e'_2 \quad r' \neq ?}{M[e_1 < e_2] \vdash e'_1\ r'\ e'_2}\ (\textsc{Up-W})$$

Fig. 3: Online Decision Making

judges two events as unknown to one another. (Un-1) decrees so when there is an intermediate event e_2 that has the same correspondence r with both e_1 and e_3 (in different orders albeit). Of course, given the transitivity of $<$, in the case of (Un-1), r cannot be $<$. (Un-2) is similar except that, in the microcosm of discourse, the intermediate event e_2 is unknown to e_3. Then, (Un-3) decrees for e_1 and e_2 to be unknown to one another when there is no intermediate event in the microcosm that is in correspondence with both e_1 and e_2. The last rule of the group, i.e., (Un-4) declares the correspondence between an event that is not in a microcosm to be unknown with any other event. Note that all the (Un-*) rules except (Un-4) assume that the microcosm has no initial judgements between the two events. It can be shown that such a premise is not required for (Un-4).

The rules in the fourth row are routine and legislate the symmetry of $\|$, $<>$, and ? for microcosms.

Next two rules concern when a microcosm is supplied with new event information. With such a supply, the microcosm of discourse evolves into a new one. To this latter microcosm, one (and only one) more initial correspondence is available than the old microcosm. (Strng) states that, when two events are judged to be unknown to one another by a microcosm, the judgement will be changed accordingly when the respective information from the enclosing world of event evolves the microcosm. (Weak) says the supply of new event infor-

mation from the enclosing world of events preserves every event correspondence decreed earlier not to be unknown. Note that the supply of new information is only possible through the enclosing world of event.

Finally, the last two rules are on update of $<>$ instances. (UP-S) (for strengthening) and (UP-W) (for weakening) are the update counterparts (STRNG) and (WEAK). The difference is that, for the former pair of rules, the total number of correspondences initially known to the old microcosm and the new one are equal. Yet, in (UP-S) and (UP-W), one and only one $<>$ in the old microcosm is replaced by exactly one $<$ in the new microcosm. The respective microcosm judgements are updated consequently.

The following lemma will later be used in Lemma 5.

Lemma 1. *Suppose that* $M \vdash e_1\ r\ e_2$*, where* $r \in R \cup \{<>\}$*. Then,* $M \overset{\circ}{\vdash} e_1\ r\ e_2$*.*

For a derivation Π of the form

$$\frac{\Pi_1 \quad \Pi_2 \quad \Pi_n}{M _\ c'}$$

we write $c \notin \Pi$ when $c \neq c'$ and $c \notin \Pi_1, c \notin \Pi_2, \ldots, c \notin \Pi_n$. (The "$_$" in "$M _\ c'$" above can be "$\overset{\circ}{\vdash}$", "$\vdash$", and "$\vdash^*$." See Definition 6 for the latter.)

Fundamental results about online decision making follow. Theorem 1 is on its computability. Then, Theorem 2 proves consistency. At last, Lemma 2 and Corollary 1 focus on the relative accuracy of online decision making.

Theorem 1. *The online decision making algorithm is computable: For every distinct pair of events* e_1 *and* e_2 *and microcosm* M*, in finite number of steps, the relation* r *for which* $M \vdash e_1\ r\ e_2$ *can be found, if any.*

Theorem 2. *Online decision making is consistent:* $M \vdash e_1\ r\ e_2$ *and* $M \vdash e_1\ r'\ e_2$ *imply* $r = r'$*.*

Proof. Let $\Pi = M \vdash e_1\ r\ e_2$ and $\Pi' = M \vdash e_1\ r'\ e_2$. The proof is by rule-based induction on Π, namely, by case analysis of $\mathsf{lr}(\Pi)$:

- (UN-n) for $n \in \{1, 2, 3\}$. In all those cases, as a part of the hypotheses, $M \overset{\circ}{\nvdash} e_1\ _\ e_2$. Hence, $\mathsf{lr}(\Pi') \neq$ (IN-OK). Furthermore, $\mathsf{lr}(\Pi') \neq$ (UP-S) (because, then, $r' = <$ and $M \overset{\circ}{\vdash} e_1 < e_2$) and $\mathsf{lr}(\Pi') \neq$ (UP-W) (because, then, $r' \neq ?$, and, by Lemma 1, $M \overset{\circ}{\vdash} e_1\ r'\ e_2$). Likewise, $\mathsf{lr}(\Pi') \neq$ (STRNG) either because, then, $M \overset{\circ}{\vdash} e_1\ r'\ e_2$ for $M = (_ + e_1\ r'\ e_2)$. We claim that the last rule in $M \vdash e_1\ r'\ e_2$ cannot be (WEAK) either, and, the result follows because all the remaining rules imply that $r' = ?$.
  We now prove our last claim. If the last rule in $M \vdash e_1\ r'\ e_2$ is to be (WEAK), there exists a microcosm M' such that $M = (M' + e'_1\ _\ e'_2)$ and $M' \vdash e_1\ r'\ e_2$. Besides, $r' \neq ?$, which, by Lemma 1, implies $M' \overset{\circ}{\vdash} e_1\ r'\ e_2$. This is, however, a contradiction because, then $M \overset{\circ}{\vdash} e_1\ r'\ e_2$ as well.

– (Un-4). When $e \notin M$ and $e \neq e'$, there essentially is no other rule that can apply than (Un-4). That is, the last rule for $M \vdash e_1\ r'\ e_2$ too needs to be (Un-4) and $r' = ?$.

We drop the remaining cases due to space restrictions. □

We now introduce our measure for when a correspondence carries more accurate information about a pair of events than another. The measurement is based on a comparison between the accuracy of the respective relations the two correspondences attribute to a given pair of events.

Definition 5. *For a pair of relations $r, r' \in R \cup \{<>, ?\}$, write $r \sqsubset r'$ – for r is less accurate than r' – when: (i) $r' \neq ?$ and $r = ?$, (ii) $r' = <$ and $r = <>$, (iii) $r' = <^{-1}$ and $r = <>$, (iv) $r = r' = <$, and (v) $r = r' = \|$. Write $\sqsubseteq$ for the reflexive closure of $\sqsubset$.*

The following result states that a microcosm always **approximate**s its enclosing world of event: For every pair of events, when the relation a microcosm attributes to the pair does not exactly coincide with that of its enclosing world of events, the microcosm is only less accurate. This is the essence of our model being weaker than the mainstream practice where every device is exactly as accurate as its enclosing ecosystem.

Lemma 2. *Let $M \lhd W$. Suppose also that $W \vDash e_1\ r_W\ e_2$ and $M \vdash e_1\ r_M\ e_2$. Then, $r_M \sqsubset r_W$.*

Here is what the following result stipulates: When two microcosms of the same world of events do not agree on a given pair of events, it only is that one of the two is more accurate than the other. In other words, two microcosms of the same world of events will never attribute conflicting relations to any given pair of events.

Corollary 1. *Let $M \lhd W$ and $M' \lhd W$. Suppose also that $M \vdash e_1\ r\ e_2$ and $M' \vdash e_1\ r'\ e_2$. Then, $r \sqsubseteq r'$ or $r' \sqsubseteq r$.*

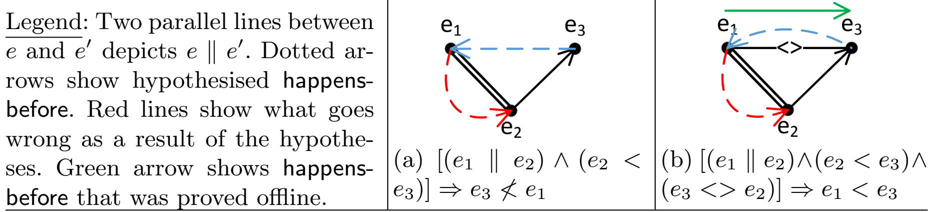
4 Offline Decision Making

The algorithm presented in this section enables a microcosm to make new decisions without depending on new supply from the enclosing world of events. As such, it suits a device required to perform offline computation. Hence, the naming "offline." Unlike our online algorithm that exclusively proves correspondences, our offline algorithm is based on cancelling possibilities. That is, deducing it that certain correspondences cannot possibly hold between the given pair of events. We say that the online decision making *confirm*s, whereas the offline one (mostly) *refute*s.

Sometimes, cancelling enough possibilities out will prove the only remaining correspondence (e.g., Fig. 4b). But, even if that is not quite the case, cancelling

one or more correspondences out is still useful (e.g., Fig. 4a): It conveys the information that the given pair of events are **not** unknown to one another. (See Lemma 5.) Most particularly, in such a scenario, it would be wrong to consider the pair parallel. That is in exact contrast with the common causality folklore that: 'when one cannot confirm any correspondence between two events, one can safely [sic] consider them parallel.'

In Fig. 4a, given that $e_1 \parallel e_2$ and $e_2 < e_3$, it cannot be that $e_3 < e_1$. This is because, then, by transitivity of happens-before, $e_2 < e_3$ and $e_3 < e_1$, imply $e_2 < e_1$, contradicting $e_1 \parallel e_2$. Fig. 4b rules $e_3 < e_1$ out similarly. But, then, given that $e_1 <> e_3$, the implication is $e_1 < e_3$. Note that the only correspondences that were available prior to concluding $e_3 \not< e_1$ (in Fig. 4a) and $e_1 < e_3$ (in Fig. 4b) were the black lines between e_1, e_2, and e_3. No new correspondence was supplied over the arguments either. The important observation to make, hence, is that such arguments do not depend on new supply from the enclosing world of events. Offline decision making (Definition 6) enables such arguments.



Legend: Two parallel lines between e and e' depicts $e \parallel e'$. Dotted arrows show hypothesised happens-before. Red lines show what goes wrong as a result of the hypotheses. Green arrow shows happens-before that was proved offline.

(a) $[(e_1 \parallel e_2) \wedge (e_2 < e_3)] \Rightarrow e_3 \not< e_1$

(b) $[(e_1 \parallel e_2) \wedge (e_2 < e_3) \wedge (e_3 <> e_2)] \Rightarrow e_1 < e_3$

Fig. 4: Two Useful Offline Deductions

Before we can delve into offline decision making itself, we need to introduce a couple of notations. For a microcosm M and a pair of distinct events e_1 and e_2 such that $M \nvdash e_1 _ e_2$, write $(M +_? e_1\ r\ e_2)$ for a microcosm that is *structurally* the same as $(M + e_1\ r\ e_2)$. Despite their same structure, the former is meant to be used only when $e_1\ r\ e_2$ is not supplied by the enclosing world of events; it rather is M with the *hypothesis* that $e_1\ r\ e_2$ was also known by M. That is, "$+_?$" is like the blue arrow in Fig. 4a. Note the additional requirement of the former over the latter. The latter only requires that $M \overset{?}{\nvdash} e_1 _ e_2$. In contrast, the former requires that $M \nvdash e_1 _ e_2$. (By definition, the requirement for $(M +_? e_1\ r\ e_2)$ implies the requirement of $(M + e_1\ r\ e_2)$ too. Hence, $(M +_? e_1\ r\ e_2)$ is well-defined.) Note also that, by Theorem 1, satisfiability of $M \nvdash e_1 _ e_2$ is computable. Define $M[e_1\ r\ e_2]_?$ similarly for a microcosm that is structurally like $M[e_1\ r\ e_2]$; yet $e_1\ r\ e_2$ is not supplied by the enclosing world of events but is only hypothesised. That is, "$[.]_?$" is like the blue arrow in Fig. 4b.

Definition 6. *Define the offline decision making process of a microcosm using the rules in Fig. 5, where the judgements take the form* $M \vdash^* e_1\ \tilde{r}\ e_2$*, and* $\tilde{r} ::= r \mid \not r$.

The rules in Fig. 5 are fairly self-explanatory and we drop explanation to save space, except for the two key rules: (Cntrd) and (Up-Cntrd). If a hypothetical correspondence between a pair of events leads to two different conclusions about a single (possibly different) pair of events, we have come to a contradiction, and, conclude the hypothesis to be false.

The online and offline decisions on the same pair of events will not conflict. That is, online and offline decision making are consistent:

Lemma 3. *Let M be a microcosm and e_1 and e_2 a pair of distinct events. Then: (i) $M \vdash e_1\ r\ e_2$ implies $M \nvdash^* e_1 \not r\ e_2$, and (ii) $M \vdash^* e_1 \not r\ e_2$ implies $M \nvdash e_1 \mathbin{_} e_2$, in particular, $M \nvdash e_1\ r\ e_2$.*

$\boxed{M \vdash^* e_1\ \tilde{r}\ e_2}$ where $\tilde{r} ::= r \mid \not r$

$$\frac{M \vdash e_1\ r\ e_2}{M \vdash^* e_1\ r\ e_2}\ (\textsc{Onl-OK}) \qquad \frac{M \vdash^* e_1 \not r\ e_2 \quad r \in R}{M \vdash^* e_1 \not ?\ e_2}\ (\textsc{Not-R})$$

$$\frac{M \vdash^* e_1 \not\lessgtr e_2}{M \vdash^* e_1 \parallel e_2}\ (\textsc{Not-CR}) \qquad \frac{M \vdash^* e_1 \nparallel e_2}{M \vdash^* e_1\ <>\ e_2}\ (\textsc{Not-Co})$$

$$\frac{(M +_? e_1\ r\ e_2) \vdash^* e_1'\ r_1'\ e_2' \quad (M +_? e_1\ r\ e_2) \vdash^* e_1'\ r_2'\ e_2' \quad r_1 \neq r_2}{M \vdash^* e_1 \not r\ e_2}\ (\textsc{Cntrd})$$

$$\frac{M[e_1\ r\ e_2]_? \vdash^* e_1'\ r_1'\ e_2' \quad M[e_1\ r\ e_2]_? \vdash^* e_1'\ r_2'\ e_2' \quad r_1 \neq r_2}{M \vdash^* e_1 \not r\ e_2}\ (\textsc{Up-Cntrd})$$

$$\frac{M \vdash^* e_1\ <>\ e_2 \quad M \vdash^* e_1 \not< e_2}{M \vdash^* e_2 < e_1}\ (\textsc{Not-HB}) \qquad \frac{M \vdash^* e_1 \not< e_2 \quad M \vdash^* e_2 \not< e_1}{M \vdash^* e_1 \not\lessgtr e_2}\ (\textsc{No-HBs})$$

$$\frac{M \vdash^* e_1 \not ?\ e_2}{M \vdash^* e_2 \not ?\ e_1}\ (\textsc{NU-Sym}) \qquad \frac{M \vdash^* e_1 \not\lessgtr e_2}{M \vdash^* e_2 \not\lessgtr e_1}\ (\textsc{NCR-Sym}) \qquad \frac{M \vdash^* e_1 \nparallel e_2}{M \vdash^* e_2 \nparallel e_1}\ (\textsc{NCo-Sym})$$

Fig. 5: Offline Decision Making

Lemma 4. *Let M be a microcosm and e_1 and e_2 a pair of distinct events. If $M \vdash e_1\ r\ e_2$ and $M \vdash^* e_1\ r'\ e_2$, then $r = r'$.*

The offline decision making can be used, for example, to mechanically conclude in the case of Fig. 4a that $M \vdash^* e_3 \not ?\ e_1$:

Lemma 5. *Let $M \vdash e_1 \parallel e_2$ and $M \vdash e_2 < e_3$ but $M \nvdash e_3 \mathbin{_} e_1$. Then, $M \vdash^* e_3 \not ?\ e_1$.*

Proof. Here is the mechanical proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{M \vdash e_1 \parallel e_2}{(M +_? e_3 < e_1) \vdash e_1 \parallel e_2}\,(\textsc{Weak})}{(M +_? e_3 < e_1) \vdash e_2 \parallel e_1}\,(\textsc{Co-Sym})}{(M +_? e_3 < e_1) \vdash^* e_2 \parallel e_1}\,(\textsc{Onl-OK}) \quad \dfrac{\dfrac{\dfrac{\dfrac{M \vdash e_2 < e_3}{M \vdash^{\circ} e_2 < e_3}\,(*) \quad \dfrac{e_3 < e_1 \in (M +_? e_3 < e_1)}{(M +_? e_3 < e_1) \vdash^{\circ} e_3 < e_1}\,(\textsc{Init})}{(M +_? e_3 < e_1) \vdash^{\circ} e_2 < e_1}\,(\textsc{In-Tr})}{(M +_? e_3 < e_1) \vdash e_2 < e_1}\,(\textsc{In-OK})}{(M +_? e_3 < e_1) \vdash^* e_2 < e_1}\,(\textsc{Onl-OK})}{\dfrac{M \vdash^* e_3 \not< e_1}{M \vdash^* e_3 \not ?\ e_1}\,(\textsc{Not-R})}\,(\textsc{Cntrd})$$

where the derivation labelled $(*)$ is Lemma 1. □

5 Forward Bisimilarity

In this section, we present our first notion of microcosm bisimilarity. We start by defining microcosm analogy (Definition 7), namely, what exactly we mean when we say two microcosms agree on every correspondence. Then, we show that such microcosms will evolve likewise when supplied with the exact same new single correspondence (Theorem 3), i.e., they are forward bisimilar (Definition 9). The most important result of this section is Theorem 4, which proves it that the order of arrival of causal information is irrelevant so long as the same correspondences are available to a pair of bisimilar microcosms. Finally, Theorem 5 establishes the bisimilarity of analogy. We call the bisimilarity of this section forward to contrast it with that of next section (Definition 11), which we call backward.

Definition 7. *Call microcosms M and M' analogous – write $M \approx M'$ – when: $\forall e_1, e_2.\ M \vdash e_1\ r\ e_2 \Leftrightarrow M' \vdash e_1\ r\ e_2$.*

In words, two microcosms are analogous when they 'agree on the correspondence between every pair of events.' That can, for instance, be two replicas of a single data centre that are in the same state. As another example consider a copy taken from a device before it temporarily dies. As soon as the original device comes back to life, the original device and the copy would be analogous. Interestingly enough, the order of arrival of the causal information to the original device is completely sporadic to the copy. Note that Definition 7 has even no explicit mention of the enclosing worlds of events of the two microcosms.

Definition 8. *Define $\xrightarrow{c}$ for the transition system $\mathscr{T}_F(W) = (\mathcal{M}_W, \mathcal{C}^*_W, \dashrightarrow)$ such that $M \xrightarrow{c} M'$ when $M' = (M + c)$ for some $c \in \mathcal{C}^*_W$. Call $\mathscr{T}_F(W)$ the forward transition system of W.*

The above definition formalises our understanding of a microcosm evolving *forward* with the arrival of new supply to it. The following two lemmata explore two different scenarios for forward evolution: when the new supply is not used for deriving the correspondence between a given pair of events (Lemma 6) and when it is (Lemma 7). Those two pave the road for Theorem 3.

Lemma 6. *Suppose that $M \approx M'$ and $M \xrightarrow{c} (M + c)$. Then, $\Pi = (M + c) \vdash e_1\ r\ e_2$ implies $(M' + c) \vdash e_1\ r\ e_2$ when $c \notin \Pi$.*

Lemma 7. *Suppose that $M \approx M'$ and $M \xrightarrow{c} (M + c)$. Then, $\Pi = (M + c) \vdash e_1\ r\ e_2$ implies $(M' + c) \vdash e_1\ r\ e_2$ when $c \in \Pi$.*

Proof. Induction on the size of Π by case distinction on $\mathsf{lr}(\Pi)$.

We next define our notion of forward bisimulation (and bisimilarity) and prove that analogy is a bisimulation.

Definition 9. *Call a binary relation $\mathcal{R}$ on $\mathcal{M}_W$ a bisimulation for $\mathscr{T}_F(W)$ when for every microcosms M_1 and M_2 of W such that $M_1\ \mathcal{R}\ M_2$, the following hold:*

– $M_1 \xrightarrow{c} M_1' \Rightarrow \exists M_2' \lhd W.\ (M_2 \xrightarrow{c} M_2') \wedge (M_1' \mathrel{\mathcal{R}} M_2')$, *and*
– $M_2 \xrightarrow{c} M_2' \Rightarrow \exists M_1' \lhd W.\ (M_1 \xrightarrow{c} M_1') \wedge (M_1' \mathrel{\mathcal{R}} M_2')$.

Write $\sim_F$ *for the bisimilarity of* $\mathscr{T}_F(W)$*, i.e., the largest bisimulation for* $\mathscr{T}_F(W)$*.*

Theorem 3. *For every* W*, the relation* $\approx$ *is a bisimulation for* $\mathscr{T}_F(W)$*.*

Proof. Let $M, M' \lhd W$ and $M \approx M'$. Suppose that $M \xrightarrow{c} (M + c)$ and $\Pi = (M + c) \vdash e_1\ r\ e_2$. When $c \notin \Pi$, by Lemma 6, $(M + c) \vdash e_1\ r\ e_2$. When $c \in \Pi$, by Lemma 7, $(M + c) \vdash e_1\ r\ e_2$. The result follows by symmetry. □

Now that we are armed with Theorem 3, it is easy to prove Theorem 4. We would like to draw the reader's attention to the small length of the proof and the simple technique used for it. Such a comfort is a consequence of bisimulation being such a strong concept.

For a given n, write $\overline{c}$ for $c_1, c_2, \ldots, c_n$ and $n = |\overline{c}|$. Extend $\dot{\rightarrow}$, accordingly, to $\dot{\twoheadrightarrow}$ where $\overset{\overline{c}}{\twoheadrightarrow}$ abbreviates $\xrightarrow{c_1} \circ \xrightarrow{c_2} \circ \cdots \circ \xrightarrow{c_n}$. Furthermore, write $\overline{c}' = p(\overline{c})$ when $\overline{c}'$ is a permutation of $\overline{c}$.

Theorem 4. *Suppose that* $M_0 \approx M_0'$*. Suppose also that* $M_0 \overset{\overline{c}}{\twoheadrightarrow} M$ *and* $M_0' \overset{\overline{c}'}{\twoheadrightarrow} M'$*, where* $\overline{c}' = p(\overline{c})$*. Then,* $M \approx M'$*.*

Proof. We proceed by strong induction on n, where $n = |\overline{c}|$:

– $n = 1$. By Theorem 3.
– $n = k$. Suppose that the theorem is correct for every $n < k$. The case when $\overline{c} = \overline{c}'$ is immediate. Otherwise, let k_0 be the first position where $\overline{c}$ and $\overline{c}'$ disagree. That is, $M_0 \overset{\overline{c}_l}{\twoheadrightarrow} M_{k_0-1} \xrightarrow{c_{k_0}} M_{k_0} \overset{\overline{c}_r}{\twoheadrightarrow} M$ and $M_0' \overset{\overline{c}_l'}{\twoheadrightarrow} M_{k_0-1}' \xrightarrow{c_{k_0}'} M_{k_0}' \overset{\overline{c}_r'}{\twoheadrightarrow} M'$ such that $\overline{c}_l = \overline{c}_l'$, $c_{k_0} \neq c_{k_0}'$, and $\overline{c}_r' = p(\overline{c}_r)$. Then, $M_{k_0} \approx M_{k_0}'$ is immediate from Theorem 3. And, given that $|c_{k_0}\overline{c}_r| = |c_{k_0}'\overline{c}_r'| < k$, by the inductive hypothesis, $M \approx M'$.

The result follows. □

Theorem 5. *For* W*, the relation* $\approx$ *is the bisimilarity of* $\mathscr{T}_F(W)$*, i.e.,* $\approx = \sim_F$*.*

6 Backward Bisimilarity

Only limited resources are available to devices, especially the edge devices. Emptying the disk or memory of such a device is routine then. To that end, usually, one removes the outdated data to come to a new manageable state. This section deals with when (causal) information is to be removed from devices, say due to resource limitation or outdatedness. That too can be seen as an evolution for a microcosm, albeit *backward* (Definition 10). We show that microcosm analogy (Definition 7) gives rise to a bisimilarity for backward evolution as well (Theorem 8). Besides, this section presents the backward counterpart of Theorem 4 that proves: The order of removal of causal information from bisimilar devices does not matter in that they will again be bisimilar once they are both done with the set of correspondences (Theorem 7).

Definition 10. *Define* $\overset{c}{\dashleftarrow}$ *for the transition system* $\mathscr{T}_B(W) = (\mathcal{M}_W, \mathcal{C}^*_W, \dashleftarrow)$ *such that* $M \overset{c}{\dashleftarrow} M'$ *when* $M = (M' + c)$ *for some* $c \in \mathcal{C}^*_W$. *Call* $\mathscr{T}_B(W)$ *the backward transition system of* W.

The notation $M \overset{c}{\dashleftarrow} M'$ is indeed intended to be read from left to right to denote getting from M to M' by the removal of c.

Definition 11. *Call a binary relation* $\mathcal{R}$ *on* $\mathcal{M}_W$ *a bisimulation for* $\mathscr{T}_B(W)$ *when for every microcosms* M_1 *and* M_2 *of* W *such that* $M_1 \mathcal{R} M_2$, *the following hold:*

- $M_1 \overset{c}{\dashleftarrow} M'_1 \Rightarrow \exists M'_2 \lhd W.\ (M_2 \overset{c}{\dashleftarrow} M'_2) \wedge (M'_1 \mathcal{R} M'_2)$, *and*
- $M_2 \overset{c}{\dashleftarrow} M'_2 \Rightarrow \exists M'_1 \lhd W.\ (M_1 \overset{c}{\dashleftarrow} M'_1) \wedge (M'_1 \mathcal{R} M'_2)$.

Write $\sim_B$ *for the bisimilarity of* $\mathscr{T}_B(W)$, *i.e., the largest bisimulation for* $\mathscr{T}_B(W)$.

Theorem 6. *For every* W, *the relation* $\approx$ *is a bisimulation for* $\mathscr{T}_B(W)$.

We extend $\dashleftarrow$, like $\dot{\twoheadrightarrow}$ to $\dot{\twoheadleftarrow}$ where $\overset{\overline{c}}{\twoheadleftarrow}$ abbreviates $\overset{c_1}{\dashleftarrow} \circ \overset{c_2}{\dashleftarrow} \circ \cdots \circ \overset{c_n}{\dashleftarrow}$. In words, the following theorem states that the order of removal is irrelevant so long as the same set of correspondences are removed from analogous microcosms.

Theorem 7. *Suppose that* $M_0 \approx M'_0$. *Suppose also that* $M_0 \overset{\overline{c}}{\twoheadleftarrow} M$ *and* $M'_0 \overset{\overline{c}'}{\twoheadleftarrow} M'$, *where* $\overline{c}' = p(\overline{c})$. *Then,* $M \approx M'$.

Proof. Similar to Theorem 4. □

Theorem 8. *For* W, *the relation* $\approx$ *is the bisimilarity of* $\mathscr{T}_B(W)$, *i.e.,* $\approx = \sim_B$.

7 Related Work

The partial knowledge of a microcosm w.r.t. its enclosing world of events resembles the classical "knowledge vs common knowledge" model [14,11]. The latter works, however, take an algorithmic approach. Ben-Zvi and Moses [5,4] take the same approach to coin the *Syncausality* as an extension to happens-before for synchronised computations. Gonczarowski and Moses [13] too generalise the classic model to characterise the interactive epistemic state when temporal constraints must be met. The final work in this thread [1] extends the classic model for reasoning about trust in distributed settings.

Burckhardt [6] takes a novel approach to define causal consistency not just in terms of happens-before, but also w.r.t. arbitration order and visibility order. The gain is a more precise definition of how causality is used to ensure consistency. In addition to being model theoretic, unlike our work, his approach is not based on explicit causality [3].

One particular motivation for confining the universal knowledge of a world of events to microcosms is scalability. Systems that reduce the overhead of maintaining scalable causal consistency in wide-area replicated key-value stores include Orbe [9], COPS [16], Eiger [17], and ChainReaction [2]. COPS, in particular, defines *causal+ consistency*, which extends causal consistency with convergent conflict handling. This ensures that replicas that see concurrent updates will be updated in a consistent fashion. The systems mentioned above can incur significant overhead (in computation, storage, network load, and latency) to maintain causal consistency in scalable fashion. Du et. al [10] explain the performance overhead of causal consistency vs. eventual consistency. They introduce a protocol to reduce this overhead.

8 Conclusion and Future Work

We provide the first proof theoretic modelling of causality in distributed systems, with special emphasis on partiality of causal knowledge. In our model, a device has strictly less causal information than a holistic causality store (Lemma 2). We offer rules for deducing causal information both when a device is online and offline (Definitions 4 and 6). We prove properties of our deductions, which are both theoretically attractive and practically valuable (Theorems 1, 2, Corollary 1, and Lemmata 3 and 4). We refute a causality folklore using a mechanical proof (Lemma 5). We define two notions of bisimilarity (Definitions 9 and 11) to prove that the order of addition or removal of causal data is irrelevant for bisimilar devices (Theorems 4 and 7, respectively).

Our modelling does not take it into consideration that information about concurrent events might arrive not at the same time. That lag makes a device observe an internal ordering for concurrent events. The interplay between the concurrency and the internal order becomes more interesting when relaying the concurrency to the next device in the vicinity. A future work for us is the study of that. We anticipate that a new set of proof systems will be required, their status w.r.t. the ones in this paper also requires dedicated study. Another related future work is to take arbitration and visibility into account.

The ability to reason about partial causal information suggests positive interaction with causal+ consistency: replicas that are actually causal but for which the causality is not known yet will remain consistently updated as the known causality increases (i.e., updates do not have to be redone as knowledge increases). This is an important property for causal+ consistency can be a useful model to use together with the deduction systems introduced in this paper. Future work will reveal how the ability to deduce causality can increase the efficiency of COPS (and its counterparts) by reducing the overhead.

References

1. A. Abdul-Rahman, *A Framework for Decentralised Trust Reasoning*, Ph.D. thesis, U. London, 2005.

2. S. Almeida, J. Leitão, and L. E. T. Rodrigues, *ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication*, 8th EuroSys (Z. Hanzálek, H. Härtig, M. Castro, and M. F. Kaashoek, eds.), ACM, April 2013, pp. 85–98.
3. P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, *The Potential Dangers of Causal Consistency and an Explicit Solution*, 3rd SOCC (M. J. Carey and S. Hand, eds.), ACM, October 2012, pp. 22–1–22–7.
4. I. Ben-Zvi, *Causality, Knowledge and Coordination in Distributed Systems*, Ph.D. thesis, Technion, 2010.
5. I. Ben-Zvi and Y. Moses, *Beyond Lamport's* Happened-Before*: On the Role of Time Bounds in Synchronous Systems*, 24*th* DISC (N. A. Lynch and A. A. Shvartsman, eds.), LNCS, vol. 6343, Springer, September 2010, pp. 421–436.
6. S. Burckhardt, *Principles of Eventual Consistency*, FTPL **1** (2014), no. 1-2, 1–150.
7. B. Charron-Bost, *Concerning the Size of Logical Clocks in Distributed Systems*, Inf. Proc. Lett. **39** (1991), no. 1, 11–16.
8. 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*, 21st SOSP, October 2007, pp. 205–220.
9. J. Du, S. Elnikety, A. Roy, and W. Zwaenepoel, *Orbe: Scalable Causal Consistency using Dependency Matrices and Physical Clocks*, SOCC (G. M. Lohman, ed.), ACM, October 2013, pp. 11:1–11:14.
10. J. Du, C. Iorgulescu, A. Roy, and W. Zwaenepoel, *Closing the Performance Gap between Causal Consistency and Eventual Consistency*, 1st PaPEC, no. EPFL-CONF-198281, ACM, April 2014.
11. R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Common Knowledge Revisited*, Knowledge Contributors (V. F. Hendricks, K. F. Jørgensen, and S. A. Pedersen, eds.), Synthese Library, vol. 322, Springer Netherlands, 2003, pp. 87–104.
12. Seth Gilbert and Nancy Lynch, *Brewer's Conjecture and the Feasibility of Consistent Available Partition-Tolerant Web Services*, In ACM SIGACT News, 2002, p. 2002.
13. Y. A. Gonczarowski and Y. Moses, *Timely common knowledge*, 14th TARK (B. C. Schipper, ed.), January 2013.
14. J. Y. Halpern and Y. Moses, *Knowledge and Common Knowledge in a Distributed Environment*, JACM **37** (1990), no. 3, 549–587.
15. L. Lamport, *Time, Clocks, and the Ordering of Events in a Distributed System*, Commun. ACM **21** (1978), no. 7, 558–565.
16. W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, *Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS*, 23rd SOSP (New York, NY, USA), ACM, 2011, pp. 401–416.
17. ———, *Stronger Semantics for Low-Latency Geo-Replicated Storage*, 10th NSDI (N. Feamster and J. C. Mogul, eds.), USENIX, April 2013, pp. 313–328.
18. F. B Schneider, *Implementing Fault-Tolerant Services using the State Machine Approach: A Tutorial*, ACM CSUR **22** (1990), no. 4, 299–319.
19. R. Schwarz and F. Mattern, *Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail*, Dist. Comp. **7** (1994), no. 3, 149–174.
20. P. Sérgio Almeida, C. Baquero, R. Gonçalves, N. M. Preguiça, and V. Fonte, *Scalable and Accurate Causality Tracking for Eventually Consistent Stores*, 14th IFIP DAIS, June 2014, pp. 67–81.