Functional Programming Languages in Computing Clouds

Practical and Theoretical Explorations

A thesis submitted in partial fulfilment

of the requirement for the degree of Doctor of Philosophy

Joerg Fritsch

June 2016

Cardiff University
School of Computer Science & Informatics

Declaration

This work has not been submitted in substance for any other degree or award at this or any other university or place of learning, nor is being submitted concurrently in candidature for any degree or other award.

Signed .... (candidate)

Date ....

Statement 1

This thesis is being submitted in partial fulfilment of the requirements for the degree of PhD.

Signed .... (candidate)

Date ....

Statement 2

This thesis is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by explicit references. The views expressed are my own.

Signed .... (candidate)

Date ....

Statement 3

I hereby give consent for my thesis, if accepted, to be available online in the University's Open Access repository and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed .... (candidate)

Date ....

Abstract

Cloud platforms must integrate three pillars: messaging, coordination of workers and data. This research investigates whether functional programming languages have any special merit when it comes to the implementation of cloud computing platforms. This thesis presents the lightweight message queue CMQ and the DSL CWMWL for the coordination of workers that we use as artefact to proof or disproof the special merit of functional programming languages in computing clouds. We have detailed the design and implementation with the broad aim to match the notions and the requirements of computing clouds. Our approach to evaluate these aims is based on evaluation criteria that are based on a series of comprehensive rationales and specifics that allow the FPL Haskell to be thoroughly analysed.

We find that Haskell is excellent for use cases that do not require the distribution of the application across the boundaries of (physical or virtual) systems, but not appropriate as a whole for the development of distributed cloud based workloads that require communication with the far side and coordination of decoupled workloads. However, Haskell may be able to qualify as a suitable vehicle in the future with future developments of formal mechanisms that embrace non-determinism in the underlying distributed environments leading to applications that are anti-fragile rather than applications that insist on strict determinism that can only be guaranteed on the local system or via slow blocking communication mechanisms.

Acknowledgements

Completing this thesis as a part-time student while being a full-time dad and a full-time employee has easily been one of the most difficult things I have ever undertaken. During the course of this work, my supervisor Coral Walker has been extremely supportive, and has displayed a great deal of flexibility in helping me to reach the finishing line. Coral has my sincere gratitude for her assistance and guidance.

To the Cardiff School of Computer Science & Informatics for making a sizeable AWS account available to develop and test the software produced for chapter 4. In particular, thanks to my fellow PhD candidate Ms Neelam Memon for the many conversations about research making up for the missed out water-cooler talks that I would have undoubtedly had if I had been a full-time student.

I would also like to thank Wilco van Ginkel for actually reading and critiquing this thesis. My special thanks goes to David W. Walker for kindly proof reading the three papers that were extracted from this research.

The second half of this thesis was completed while working at Gartner and I thank them for their understanding in allowing me to publish outside the Gartner research agenda and the financial support over the course of two years.

Most of all though, I would like to acknowledge my family. My deepest gratitude goes to my wife, Nina, who has been nothing but unbelievably patient and encouraging over these years that I have been a part-time student.

Contents

List of Publications

The work introduced in this thesis has been disseminated in the following publications.

Joerg Fritsch and Coral Walker. Cmq-a lightweight, asynchronous high-performance messaging queue for the cloud. *Journal of Cloud Computing*, 1(1):1–13, 2012

Coral Walker Joerg Fritsch. Cwmwl, a linda-based paas fabric for the cloud. *Journal of Communications*, 9(4):286–298, 2014

Joerg Fritsch and Coral Walker. The problem with data. In *7th International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE/ACM, 2014

More specifically, these papers draw from this thesis as follows. Understanding and insights from the implementation of the UDP based message queue CMQ that was gained in [FW12a] have been used to analyse and assess Asynchronous operations and messaging in Chapter 3. The integration with actual workloads is described in [JF14] and lead to the assessment criteria that appear in Chapter 4 Coordination. Finally, the observations concerning the integration with data in Chapter 5 are based on [FW14b].

List of Figures

List of Tables

List of Algorithms and program Code

List of Acronyms

ACP	Algebra of Communicating Processes
AppController	Apppplication Controller
AppDB	Application Database
API	Application Program Interface
AS	Application Servers
ASCII	American Standard Code for Information Interchange
AST	Abstract Syntax Tree
AWS	Amazon Web Services
CD	Continuous Delivery
CI	Continuous Improvement
CSP	Cloud Service Provider
DDL	Dynamic Linked Library
DFG	Data Flow Graph
DSL	Domain Specific Language
DSM	Distributed Shared Memory
EC	Evaluation Criteria

FPL	Functional Programming Language
GPL	General Programming Language
HaLVM	Haskell Lightweight Virtual Machine
HVM	Hardware Virtual Machine
HPC	High Performance Computing
IaaS	Infrastructure as a Service
IO	Input/Output
IoT	Internet of Things
IPC	Inter Process Communication
MMOG	Massive Multiplayer Online Game
MOM	Message Oriented Middleware
NFS	Network File System
PaaS	Platform as a Service
RDBMS	Relational Database Management System
SaaS	Software as a Service
SLA	Service Level Agreement
TS	Tuple Space
UC	Utility Computing
UDT	UDP based Data Transfer Protocol
UML	Unified Modeling Language
VM	Virtual Machine
YARN	Yet Another Resource Negotiator

Chapter 1

Introduction

Cloud computing is a disruptive innovation based on ‘seamless’ scaling multi-tenant infrastructures. According to the author’s own experience[1], since the year 2006 when Amazon Web Services launched the ‘Elastic Compute Cloud’, the first commercial offering, the notions and paradigms sold to the end customer have gone through several revisions. Initially cloud computing was marketed based on the vision of simplicity, elasticity and the illusion of indefinite resources. Cloud Service Providers (CSPs) implemented architectures they had been using in their own IT that was designed for high efficacy and run by highly skilled staff. For example, CSPs did away with network subnetting, network firewalls and DMZs and replacing them with host based access lists. CSPs preferred eventual consistency before the CAP theorem and were not afraid to use non-deterministic approaches as long as the overall result would ‘eventually consistent’ [Vog09], and preferring to recover from failure rather than preventing it.

However, it showed very quickly that the new paradigms were asking to much from the enterprise clients that CSPs were trying to win as customers. During the following years CSPs have put considerable effort in using their platforms to emulate features and experiences that traditional IT had been following ever since. For example, by now it is possible to implement network firewalls and there is support to deploy traditional monolithic enterprise applications,

[1]The author of this Thesis has been planing, implementing and researching cloud paradigms for government and industry since the year 2008.

to name a few. As a result enterprise customers can lift & shift what they have in their traditional data center into the cloud.

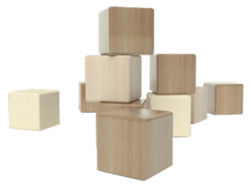
While in the field of cloud computing these adaptations are still ongoing, there are newer approaches that re-use the initial visions. Examples are micro segmentation such as VMware NSX or Cisco ACI, stateless micro services deployed on top of Linux containers managed by Docker, and serverless compute such as AWS Lambda for processing streams and events. Contrary to first generation offerings of computing clouds, the newer offerings are not geared towards traditional enterprise customers but towards innovative use cases such as the IoT and have a high chance to achieve wide spread adoption and understanding. This thesis is focusing on the key paradigms of computing clouds as they were introduced in the years 2006 - 2012 and re-appeared approximately in the year 2014 onwards.

1.1 Background

CSPs have the requirement to deliver on the promise in the most economic and efficient way. Tables 1.1 and 4.1 compare the relevance, limitations and efficacy of the currently prevailing IaaS service model with the industry vision of a fully elastic PaaS (that is not based on an underlying IaaS infrastructure) and Utility Computing as envisioned by Carr [Car05]. The coarse granularity of the units of scale and the identified limitations have negative implications on the elasticity and efficacy that can be achieved (for more information, please read Chapter 4). The coarse granularity has also negative impacts on the customer who needs to run, maintain and secure a potentially very large number of virtualized operating systems to instantiate a clustered application. However, this is an unacceptable overhead to CSPs and their customers that want to leverage the theoretical promise of cloud computing to run applications or services in a lean, seamless scaling and high available environment.

The author investigates whether the functional programming language (FPL) Haskell that provides functions as granular, lightweight and easy access units of scale has special merit to alleviate the issues and limitations that we currently observe in computing clouds. The FPL Haskell is chosen as the programming language to be evaluated for the following reasons:

Table 1.1: The 3(+1) service models of computing clouds

Service Model	Relevance	Elasticity	Limitations
Infrastructure as a Service 	Currently prevailing delivery model	Large units of scale: VM. Means of scale: add VMs. Server centric	Offerings (S, M, L, XL) have fixed computing power and RAM and are not elastic. Maximum scale (for example RAM) determined by the underlying hardware.
Platform as a Service	To date lowest adoption rate, but interest in Linux containers and Docker sparks interest in PaaS	Small(est) units of scale: threads, processes. Means of scale: spawning. ∞ number of threads or processes thinkable. Developer centric	Limited by the message passing performance in distributed architectures (e.g. MPI, Message Queues, Erlang).
Uitility Computing	In the future IT will be purchased as utility [Car05]	Unknown	Cloud and utility computing eventually lead to concentration of hardware, services and data in large Data Center. ‘Cloud scale’ = massive volumes in massively distributed architectures.

- It supports a wide range of concurrency paradigms [Mar11].
- It is a pure functional programming language that
  - Implements all key paradigms of functional programming without alteration leading to the assumption that it may be justifiable to eventually generalize from Haskell to other FPLs.
  - The impact and merit of composition, currying [2] and functional data models can be investigated clearly because they are not modified with elements of imperative programming languages, such as destructive updates to data structures that are possible with the setcadr functions in List and Scheme.
- It is independent from any third-party platform or runtime (for example Clojure and Scala are built on top of JVM, F# on top of the .NET platform).
- It is being actively researched and has an ever increasing large research community. According to the popularity tracking website langpop.com [Unt12d], in 2013, it was ranked 10th out of the 32 most talked-about programming languages on the Internet.
- It is not developed or backed by a commercial company, such as Erlang, that, in case of disagreement with the findings of this thesis, could pursue a lawsuit against the author.

1.2 Methodologies

Early on in the research process the author of this thesis realized that it would be impossible to answer the research question whether FPLs, such as Haskell, have special merit in computing clouds or not using a qualitative methodology. On the other hand, qualitative methodologies quickly lead to the result that FPLs are ideal for computing clouds but contradict the observation that in practice they are of no importance to CSPs and their customers. For an objective judgment the author of this thesis used a method of inquiry that has two main goals:

1. Producing new knowledge and meaning that can be shared.

[2]Currying a technique that is related to, but not the same as, partial function application.

2. Include reflection as an important vehicle of investigation.

The devised evaluation methodology is a transitory productive inquiry that is an adaptation of Kolb's learning cycle [Kol84] and Dewey's theory of inquiry [Dew25] as illustrated in figure 1.1. The phases mapped to this research are:

- **Observe** the implementation of computing clouds, such as large scale commercial public clouds, the operational and technical requirements, and issues.
- **Assume** relevant evaluation criteria and fallacies.
- **Select** categories of evaluation criteria that can be tested through experimentation.
- **Experiment** and create artifacts that either proof or disproof the selected assumptions.
- **Reflect** on the achieved results, for example whether they can be further confirmed using market data.
- **Generalization** and Interpretation of new knowledge.

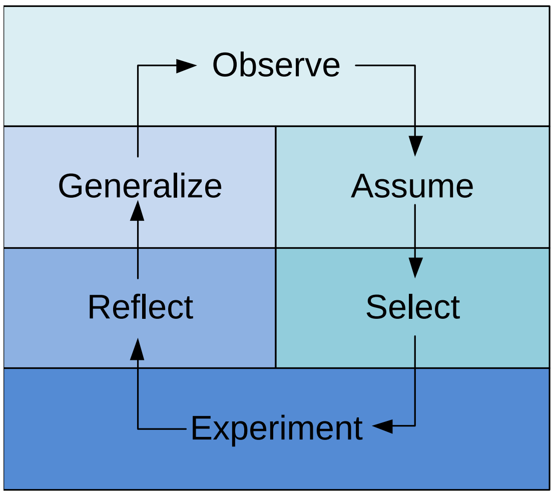


Figure 1.1: Productive inquiry

Using this methodology the author of this thesis (i) devised the architecture [3] for a perfectly elastic PaaS that scales tenant code through spawning, and (ii) wrote demonstrator code ad-

[3]The devised architectures are illustrated in Figures 4.4, 4.5 and 5.5

dressing key issues in the three categories **messaging**, **coordination**, and the integration with **data**.

To have special merit for computing clouds, FPLs would need to make a significant contribution to alleviate the limitations identified in Table 1.1. Examples are the use of asynchronous communications and novel messaging paradigms and enhance the elasticity and efficacy of modern PaaS, for example by striping the units of scale from the virtual system operating system and eventually decoupling the units of scale in space and time using well-known coordination paradigms in each of the three categories.

What if cloud computing did not mean to wrap VMs and operating systems that need to be administered and protected around atomic units of scale, such as processes or threads? What if cloud computing would have small units of scale that support distribution regardless of boundaries and scale of physical hardware? What if cloud computing was not server centric but developer defined? Since all demonstrators confirmed that FPLs would be a perfect match for computing clouds, during the reflection phase the author of this thesis decided to base the eventual judgment on a set of Evaluation Criteria (EC) extracted from his experimental explorations. This is illustrated in Figure 1.2 that shows the experimental explorations on the left side of the flow chart and the corresponding extracted EC on the right side.

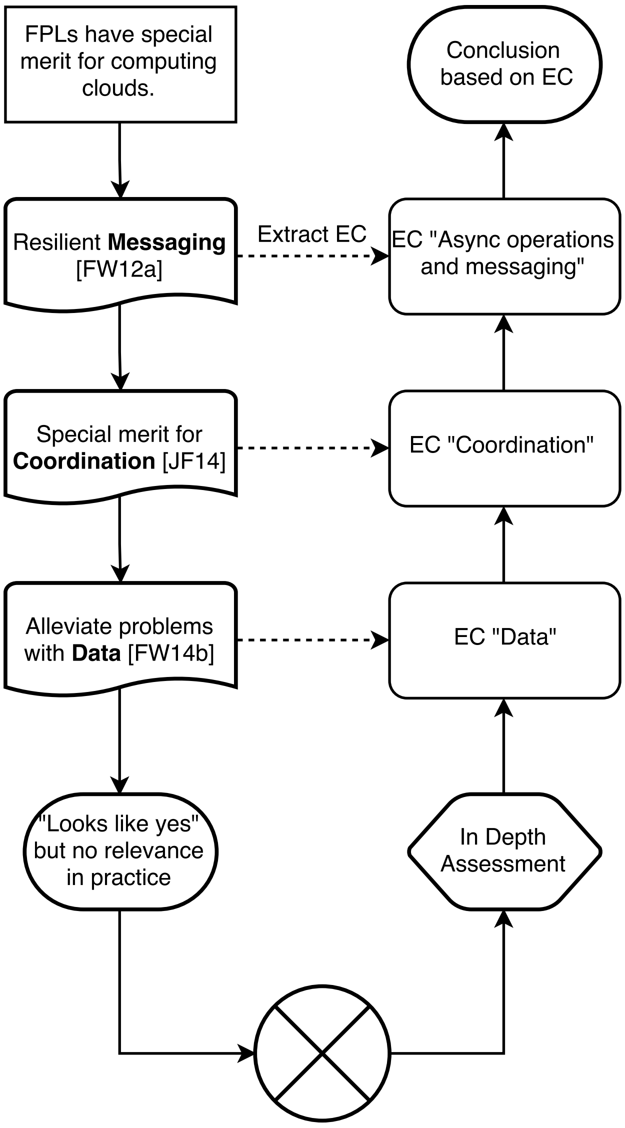

FPLs have special merit for computing clouds.
Resilient **Messaging** [FW12a]
Extract EC
EC "Async operations and messaging"
Conclusion based on EC
Special merit for **Coordination** [JF14]
EC "Coordination"
Alleviate problems with **Data** [FW14b]
EC "Data"
"Looks like yes" but no relevance in practice
In Depth Assessment

Figure 1.2: Extraction of Evaluation Criteria from Productive Inquiry

1.3 Problem Statement

IaaS and PaaS frameworks in computing clouds are coded and architected using two units of scale: Virtual Machines (VMs) and Threads. In the author's opinion neither is a good solution. While VMs as units of scale are coarse and not efficient, threads have in practice no portability beyond the physical system. This study will use a productive inquiry to determine the degree to which functions as provided by FPLs such as Haskell are suitable units of scale that alleviate the technical problems and limitations in current implementations of computing clouds.

1.4 Motivation

This research started in the year 2011 when both academia and industry set out to understand cloud computing and the paradigm changes that came with it. Computer systems and informatics are standing in front of the next sea change. The impact of cloud computing paradigms on the future of information technology could eventually be far bigger than what we might expect at this point in time eventually resulting in a significant push towards utility computing (UC).

To date computing clouds use coarse units of scale and elasticity is based on duplication. For example, IaaS scales tenant application by duplicating VMs and the Operating Systems and applications installed on top of it. PaaS scales tenant applications by duplicating the language app servers, such as JVM or Rails, and the tenant application itself. Furthermore, the application execution engine (that is a part of the PaaS platform) needs to be duplicated when the tenant application is scaled up. Virtualized or physical load balancers are the glue that re-clusters all these duplicated units of scale maintaining the illusion that we are facing truly scalable application. In most cases concurrency is the required enabler for scalability.–These limitations prevent cloud computing from being the landmark innovation that is required to put UC into practice.

When I started this research I asked myself what if cloud computing did not mean to wrap VMs and operating systems that need to be administered and protected around atomic units of scale, such as processes or thread? What if cloud computing would have small units of scale that support distribution regardless of boundaries and scale of physical hardware? What if cloud computing was not server centric but developer defined?

In order to produce better computing clouds, units of scale that are not based on VMs and Threads will need to be used and made compatible with existing cloud computing architectures and paradigms. The bright idea of this thesis is that functions as provided by FPLs could very well be a unit of scale that bring cloud computing closer to UC, or at least brings many improvements to computing clouds, such as quicker instantiation time, better security (because no underlying OS needs to be administered and secured) and less energy consumption (for more information read Chapter 4).

In practice network protocols are the most prominent cost center in distributed environments and traditional protocols will nihilate every advance in optimizing the units of scale across boundaries of physical systems. Protocols in distributed environments ideally will orchestrate and coordinate. This is why my research starts out with investigating network protocols and then continues with investigation of the suitability of FPLs as units of scale for cloud patforms and tenant applications.

1.5 Major Contributions

This dissertation contributes to the area of cloud computing. Specifically, it introduces novel thinking and techniques to the fields of inter-process communication, the identification and coordination of suitable units of scale and the integration with data. The primary objective of this dissertation is to test the hypothesis that:

1. Cloud computing platforms and applications integrate three pillars: Messaging, Coordination and Data.
2. Functions, as provided by FPLs, can be used as easy access and lightweight units of scale enhancing the efficacy across all three pillars.
3. FPLs are required for computing clouds to further develop and serve as basis of utility computing.

It should be noted that it is not possible to formally prove the correctness or falsehood of this hypothesis. Instead, this dissertation is limited to providing, hopefully strong, evidence for or against its validity. It does so by introducing a new optimistic message queue, a tuple-space based demonstration of coordination and novel thinking about data.

This dissertation makes the following contributions:

1. The question whether FPLs (in particular Haskell) are a suitable programming language/paradigm for systems that are distributed by nature like cloud computing has been answered using solid conclusions based on practical experimentation. With our demonstrator code and research we showed that Haskell is great for a single/local system, but not the right choice (unless improvements fto Haskell are made) for cloud computing.
2. The optimistic UDP-based message queue CMQ that uses FPLs to implement an anti-fragile message queue that matches the notion and underlying paradigms of computing clouds. The need for an optimistic message queue is motivated by the key paradigms of computing clouds [Vog08] that is to be prepared to take a loss and quickly recover from it rather than implementing inefficient guarantees to prevent failure by all means.
3. CWMWL, a small DSL with four primitives that uses a central tuple store to coordinate functions across the boundaries of physical system.
4. A set of requirements (Evaluation Criteria, EC) deducted from our implementations to support reflection and decision making whether FPLs really have special merit in computing clouds.
5. Novel thinking concerning the integration with data.

1.6 Thesis structure

The remaining chapters of this thesis can be summarized as follows.

Chapter 3 finds that in computing clouds, the FPL Haskell needs to support node level shared nothing architectures and provide inter node message passing. We investigate suitable programing paradigms for implementing the inter process communication (IPC) of a cloud-computing framework: (1) Remote Procedure Calls (RPC), (2) 'Erlang-Style' message passing and (3) shared memory. We present the lightweight, UDP based message queue CMQ. The concept to use UDP instead of TCP is motivated by our understanding that, in Cloud Computing, omnipresent off-the-shelf technologies (both in hard and software) are encouraged, and if preventing errors from occurring becomes too costly, dealing with the errors may be a better solution. Although CMQ is a message queue oriented communication approach, CMQ is different from the conventional MOM approach because it challenges a number of assumptions under which conventional MOM is built. For instance, in conventional MOM, messages are 'always' delivered, routed, queued and frequently follow the publish/subscriber paradigm. It is often accepted that this requires an additional layer of infrastructure and software where logic is split from the application and configured in the additional layer. On the contrary, CMQ does just enough. It does not offer any guarantees, such as reliable transmission, thus is very light weight with low overhead and fast speed. Although it does not offer guarantees, it appears to be stable in the presence of errors.

Chapter 4 assesses whether functions in the FPL Haskell can serve as easy access composable lightweight units of scale to develop scalable software that executes on computing clouds. We begin by highlighting the problems with threads as units of scale for the implementation of scalable software on computing clouds and explain why functions are a better alternative. This motivates the assessment of functions as easy access lightweight units of scale that can be coordinated to achieve scalable distributed applications. To make functions available at all across the boundaries of physical systems we introduce a PaaS framework based on the LINDA coordination language. This approach is inspired by the works of Lee [Lee06], [Lee07] and Gelertner [Gel85][4]. We find that while the FPL Haskell functions provide easy access units of scale, the FPL must be extended with a means for coordination.

Our foremost design goal is simplicity by achieving a novel unified platform rather than virtualizing and replicating the implementation of a load balanced web application that has existed since the end of the 1990s. Multiple times we challenge the academic concept of deterministic computing arguing that nondeterministic means must be introduced where needed to fit the

[4]It seems that for his works Lee had scalable software on the local multicore system in mind, while Gelertner took physically distributed system as starting point

nature and requirements of computing clouds–without disrupting the essential determinism of programming languages. In both, Chapter 3 and Chapter 4 we frequently felt as if we turned back the clock 30 years but in a good way.

In the 'Details' section of this chapter we provide more information about the instances, services and the logical architecture that we provisioned on Amazon Web Services for our experiments. Furthermore we discuss our coordination language and how it can be used. Our job here is to assess functions as suitable units of scale and to convince the reader that the approach via a coordination language has merit.

In **Chapter 5**, we move the focus from computation to data. We tackle the problem of the integration with data, a problem that has only recently been added to the feature lists and research agendas of commercial cloud service providers (CSPs). Currently research in cloud computing and distributed computing commonly investigates the distribution of computation but seems to forget that computation frequently needs data. Although in academia the underlying issue that computation should better be governed by the movement of data rather than by the 'von Neuman-style' program counters has been recognized and investigated by Kahn[5] as early as 1974 [Gil74], data driven programming languages (and architecture) paradigms have never been put to widespread use. This motivates the investigation of the nature of data and how data is currently represented in computing clouds. In this chapter we re-apply what we have learned in Chapter 3 and Chapter 4 and propose a high level framework how a next generation computing cloud that is optimized for data can be architected. This chapter has a different structure and is not an evaluation chapter. Our mission here is to evangelize the audience to open up their mind to give data the importance it deserves. Furthermore, we propagate the term 'Data Gravity' that reflects the nature of data to attract computation, not the other way around.

We argue that a distributed in memory Tuple Space (as proposed in Chapter 4) is effective in serving as backbone for a distributed scalable fabric that acts as an abstraction layer and virtualizes both data and computation. The exploration of data uncovered some interesting use cases for functions: serverless computing and the IoT that frequently do not require internode message passing and coordination. Although out of scope of this thesis, the author finds that

[5]Kahn processes use functions communicating through FIFO channels; in principal very similar t our architecture

the FPL Haskell may be a much better fit for these use cases than for computing clouds.

We conclude that our experimental work and demonstrator code was geared towards compensating that Haskell has been designed with the local system in mind (for more information we refer to Sections 4.7.1 to 4.7.6 of this thesis). Although Haskell implements all the correct paradigms and approaches, it is tied to the local system. Haskell is excellent for use cases that do not require the distribution of the application across the boundaries of (physical or virtual) systems but not appropriate as a whole for the development of distributed cloud based workloads that require communication with the far side and coordination of decoupled workloads. We argue that interaction of functions, or any other lightweight unit of scale, in computing clouds need to be dealt with in ways that are intrinsically different from objects that interact in a single system. However, Haskell may be able to qualify as a suitable vehicle in the future with future developments of formal mechanisms that embrace non-determinism in the underlying distributed environments leading to applications that are anti-fragile [Tal12] rather than applications that insist on strict determinism that can only be guaranteed on the local system or via slow blocking communication mechanisms.

1.7 Evaluation Criteria

Users need to know the strengths and deficiencies inherent in a language, and how well a language applies to a specific domain [How95]. The strengths and deficiencies inherent in FPLs and how well FPLs apply to the domain of computing clouds is assessed based on a set of EC. The EC are low level requirements that can be checked, for example, against literature (we provide the appropriate references) and against our experimental results.

While a programming language could be fully compliant with the evaluation criteria or some of them, in practice this does not necessarily mean that it truly can work in a cloud environment that is almost equally determined by the eight fallacies of distributed computing [RGO06] rather than by strengths and deficiencies inherent in a language only. The fallacies were originally stated by SUN Microsystems in the year 1994 when it first published the Java programming language–a lingua franca for the new world of distributed computing and "asserted that programmers new to distributed applications invariably make a set of assumption ... and these assumptions ultimately prove false, resulting either in the failure of the system, a substantial reduction in system scope, or in large unplanned expenses required to redesign the system to meet its original goals." [Wik15c].

The eight fallacies of distributed computing are:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology does not change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

It can be observed that the seven out of the eight fallacies of distributed computing are network centric [6] and we have reflected the importance of network centric aspects to computing clouds in the EC of the categories 'Asynchronous operations and messaging' and 'Coordination'. Although Java has become a huge platform since it first appeared in 1995, it did not (yet) attempt to abstract the eight fallacies away from the developers and to eventually make them a non-concern. A possible reason may be that the eight fallacies are rooted in the practice of the application domain distributed computing while the development of programming languages is rooted in academic programming language theory. Mitigating the eight fallacies has always been left up to the developer. To reflect this our evaluation criteria consider both, what the FPL Haskell inherently provides as part of the language and how it must be used or extended to match the evaluation criteria.

We are aware that the evaluation criteria can neither be complete nor can the answers to the question 'Which programming language is the best?' be authoritative. However, we attempt to

[6]Fallacies 1, 2, 3, 4, 5, 7 and 8 are network centric.

provide a framework for the evaluation of FPLs in computing clouds from that the reader can draw their own conclusions. We propose three top level categories of application domain criteria that we considers essential for FPLs to have special merit in computing clouds. The three categories are then subdivided into appropriate requirements (or attributes) that we consider of equal importance. A rating is given for every evaluation criteria to allow for programming languages features to partially fulfill the evaluation criteria and to state whether a particular implementation is ‘Best of Breed’ = 1, ‘Sufficient’ = 2 or ‘Deficient’ = 3.

Chapter 2

Literature Review

Cloud computing is a disruptive innovation that is characterized by Mell [MG11] as having:

- Five key concepts: Self Service, Broad Network Access, Resource Pooling, Rapid Elasticity and Measured Service
- Three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS)
- Four deployment models: Private Cloud, Community Cloud, Public Cloud and Hybrid Cloud

While most commercial clouds currently focus on IaaS and thus the deployment of dedicated virtual machines [ABLG10, 95-105], Gartner found that the final 'prevailing patterns [. . .] have not been established yet' and that by the end of 2011, all major vendors had pushed into the PaaS market (Gartner, 2011). This research is inspired by the vision that eventually utility computing [Car09] – that is much closer to the idea of PaaS rather than IaaS, will prevail.

Birman et al. recognize that cloud computing has two perspectives [BCvR09, 68-80]:

1. An outward looking perspective that embodies an elastic application executed in a secure

container[1] and accessible over the Internet as seen by developers and end users.

2. An an inward looking perspective that describes the large scale distributed cloud computing platform and its middleware as implemented and operated by the provider.

Thus, in other words, a PaaS framework is a large-scale distributed software framework (the inside perspective) that should enable the elastic secure execution of tenant applications from a 'one : many' ratio up to a 'many : one' ratio where the tenant application is eventually a large scale distributed application itself. Birman et al. reason that cloud computing research must 'not focus overly narrow' and that 'new middleware abstractions that are known to perform well when scaled out' and self-healing (autonomic) cloud computing platforms are some of the emergent research themes.

Although Birman is very determined in his call for a holistic approach and thus proposes a wide variety of research themes, the importance of the combination of hardware and software, - or the importance of hardware in general -, that eventually creates 'the illusion' of cloud computing [AFG$^+$10] slipped his attention. However, this is not surprising, since the majority of research attempting to scope cloud computing by contributing introductions, definitions and preliminary research agendas was not published before 2009 [KHSS10] when it had become clear to researchers that cloud computing would be successful. Some of the newer work however have taken an approach that goes beyond scoping and that focuses on the synergies between computer architecture and programming language paradigms [Pat10], [Sin11].

Khajeh-Hosseini groups cloud-computing research (papers) into three focus areas: general introductions, technological aspects of cloud computing and organizational aspects. Although there seems to be no shortage of (partially shallow) general introductions and research that focuses on technological aspects of IaaS, comparable research that focuses on technological aspects of PaaS is clearly lacking. Eventually, there was a total of two papers found that explicitly focus on PaaS.

Chohan et al. [CBP$^+$10] describe in their paper about the AppScale project that is an open source version of the Google App Engine (GAE) PaaS framework [goo]. According to the

[1]Birman's research predates the conception of Linux Containers managed by Docker. Birman is not referring to 'Docker Containers' and micro services here.

authors AppScale should (amongst other purposes) be a framework that scientists can use to investigate PaaS. Eventually Chohan et al. describe a component based PaaS architecture that is based on an IaaS foundation. Because components, such as application controllers, load balancers and tenant processes deployed on top of Chohan's PaaS architecture frequently can start, configure and replicate themselves, it is justified to say that the Appscale framework is acting as an autonomous framework (or autonomous programming language runtime) reflecting one of the emergent research themes identified by Birman.

While Cohan's components resemble applications with interfaces and self-organizing capabilities, Kächele et al. [KDH11], [HKDS12] take this approach one step further. Although still based on an IaaS underlay, in their COSCA PaaS implementation the tenant applications must be composed of multiple components so that 'The platform grants isolation between the components of different applications by providing a virtual container for each application. Thus, the platform can arbitrarily mix applications of different users on the available nodes.' Kächele is basing his research on (market) requirements and on the analysis of commercial cloud offerings and does not link his research to any previous scholarly papers. Nor does he give an example or closer description of a COSCA user application. Although this seems shallow at first, it is very similar to Birman's approach who tried to learn about cloud computing from commercial entities (IBM, eBay, and Microsoft) and is willing to let research follow the market requirements rather than force fitting a strictly scientific method onto cloud computing. The overall design of the COSCA platform, however, seems to be closer to the actual workings of the GAE than Appscale since, at least for Java Applications that shall be deployed on the GAE, Google recommends component based approach using Java EE containers [Kri10]. This starting point for achieving 'cloud characteristics' is also supported and further elaborated on by Ramdas and Srinivas [RS]. While Cohan's and Kächele's approach to PaaS is consistent with Mell's definition of cloud computing, the resulting cloud computing platform would be limited to elastic hosting of traditional web applications, components would represent extremely expensive and heavyweight assemblies of processes and the importance of 'Big Data' is not considered at all. In his famous introduction to cloud computing, Armbrust [AFG$^+$10] however, in a very detailed analysis based on scholarly research, lists six application types as promising use cases for cloud computing: mobile interactive applications, parallel batch processing, analytics, extension of compute intensive desktop applications and 'earthbound' applications. A concept that industry practitioners would later refer to as ‘Data Gravity’.

As to choosing the right cloud programming model, Foster et al. [FZRL08] in their introduction to cloud computing set cloud computing programming models equal to the more recent parallel programming models MapReduce [DG08a], Hadoop [SKRC10], [VMD$^+$13] and DryadLINQ [YIFB08] that are used for large scale/elastic batch processing and analytics. In contrast to Murray [MH10], Foster does not call these programming frameworks ‘coordination languages’. Foster argues that programming models based on message passing are typical grid-programming models. Although he sees some overlap between grid computing and cloud computing, he does not explicitly argue that the message passing paradigm would be suitable for cloud computing. However, he states that more and more grid applications are developed as services and gives the Web Services Resource Framework (WSRF) as an example.

Eventually Epstein [EBPJ11] puts together what Haskell ‘brings to the table’ in terms of its suitability for cloud computing. Epstein compares Haskell to Erlang and finds that Haskell offers more choices regarding data (e.g. it is up to the implementation to decide whether data is shared or copied), a wider range of concurrency as well as characteristics like purity, types and monads.

Moreover Murray [MH10] introduces a new language called Skywriting that is eventually based on the PythonVM. On the one hand Skywriting acts like a data centric coordination language, but on the other hand it is ‘turing-powerful’ so that it also can be used to ‘express iterative computations’. While Murray’s approach leads to the development of a new programming language. Lee [Lee07], [Lee06] argues repeatedly that new programming languages would not be required. Lee writes that ‘We should not replace established languages. We should instead build on them.’ and stipulates that coordination languages are ‘the right answer’. But the differences go even deeper. While Murray apparently sees that non-determinism can potentially be useful in parallel programming [Mur11], Lee believes that ‘essentially deterministic ‘composable components’ are the key to success. Regrettably Lee’s papers are not backed up by any experimental validation of his assumptions, while Murray has gone to great length here to proof the performance of Skywriting.

There are a number of publications with partially contradictory findings and conclusions concerning the feasibility of the DSL development. Some authors encourage to consider DSL development more frequently [MH05], whilst Lee recommends to re-use the given and avoid the high cost of DSL development [Lee07]. Contradictory to this, in yet another paper [Lee] Lee states that concurrency (and thus multicore problems) are better solved on a component level rather than on a thread level and suggests to develop coordination languages (a specific DSL that is) that will address his findings.

Hence, based on requirements implied by the six ‘cloud application groups’ as envisioned by Armbrust, Foster’s view when setting cloud computing programming models equal to the paradigms that are shared by, for instance, Hadoop and MapReduce and the future economic importance of data as an asset class [Ano11]; it is inevitable that the (host) programming language for a cloud computing platform (PaaS) has answers, choices and paradigms concerning data.

Turning back to Mell’s five key concepts and the concept of cloud platforms as shared resource pools, it is very surprising, that none of the programming paradigms has yet been investigated for its suitability to securely discriminate and execute code from multiple tenants. The most applicable research here goes back to Brown [BS99] and his works on SSErl, a prototype of a safe Erlang. In his papers Brown explained how security would need to be retrofitted into Erlang so that potentially hostile code from several tenants can be executed on the same platform.

On the basis of the available information it can be concluded that there are research gaps in emergent research themes in cloud computing platforms (PaaS) and programming languages that:

- Match recent advances in hardware (computer architectures) and the user experience of ‘infinite computing’.
- Implements self-healing, for example by behaving anti-fragile.
- Provides new layers of abstraction for task parallelism, data parallelism and security policies by means of either a coordination language or a new programming language, to name a few examples.

- Designs a cloud platform that can run request/response web applications as well as any application from the six groups proposed by Armbrust [5].
- Implements the cloud platform (the inside perspective) using lightweight autonomic and parallelizable components.
- Aims to play well with data centric applications and algorithms.

Chapter 3

Asynchronous operations and messaging

In Chapter Literature Review we established the view that cloud computing has two perspectives:

The outward-looking perspective. *The guest systems in clouds often have to cope with the suboptimal network conditions caused by software devices, a problem that the VEPA*[1] *standard tried to solve in 2009. The software devices, such as vswitches and vrouters, are responsible for regulating network traffic inside the cloud nodes and are guest systems themselves. Depending on the virtualization ratio, one virtual switch could be responsible for up to 64 guest systems. Guest systems frequently have to cope with packet loss [WN10] that, when using TCP/IP, costs many CPU cycles on systems that are themselves billed according to the available CPU cycles. Packet loss in TCP/IP can easily cause guest systems to grind to a halt.*

[1]IEEE 802.1Qbg

The inward-looking perspective. *The physical cloud nodes in computing clouds are organized into 'Points of Delivery'*[2] *and interconnected via equipment that either switches at line rate, or uses lossless Ethernet fabric*[3] *technologies. In switched data center networks, all Ethernet (RFC 894) based networking protocols are switched without discrimination. Congestion and packet loss are extremely unlikely in such data center networks. The overhead of TCP/IP in 10Gbps data center fabrics has led to CPU performance issues ([RGAB10], [FBB$^+$05], [RMI$^+$04]) and has given rise to new connectionless Ethernet protocols, such as RDMA over converged Ethernet (RoCE) and the Internet Wide Area RDMA Protocol (iWARP). However, both protocols require specialized hardware (network cards, switching gear) that is not in line with the trend to build clouds from commodity hardware [BDH03], and accept occasional failures rather than preventing failure at any cost [Vog08].*

In this chapter we start with the inner view and investigate technologies for implementing the inter process communication (IPC) of a cloud-computing framework and how the FPL Haskell can support these. We create a distributed middleware platform and investigate three paradigms to implement the IPC of a cloud computing framework in Haskell (1) Remote Procedure Calls (RPC), (2) 'Erlang-Style' message passing using a well-known message queuing library and (3) CMQ, a UDP-based inherently asynchronous message queue that we implemented as Haskell library.

The RPC framework that is used to implement the first paradigm should be lightweight not add (much) latencies to the process. MessagePack [Fur]) that is available in Haskell and still under active development. MessagePack supports an interface description language (IDL), is largely programming language independent and creates the possibility to provide complementary features by disjoint languages [Lee07]. For example, in a component based architecture the best language for each component could be chosen without having to alter the communication protocol. According to Henning [Hen04], RPC based messaging systems are relatively expensive and perform better in data driven context than in in contexts where only commands and states

[2]physical unit of scale in a cloud, e.g. a standardized rack of interconnected servers

[3]e.g. IEEE 802.3x PAUSE frames or vendor specific technologies

are exchanged. - A problem, that seems to be caused by poorly structured serialization. MessagePack (-RPC) addresses this problem by using the binary format to represent data structures and claims to outperform e.g. Google Protocol Buffers or Thrift.

The second route is based on ‘Erlang-Style’ message passing. There are currently two alternatives that implement ‘Erlang-Style’ message passing in Haskell [EBPJ11], [Huc99] and ZeroMQ (ØMQ) [Untc], [KH]. While Cloud Haskell is a DSL that focuses on Haskell, ØMQ is a message queuing library that has APIs for many languages and thus, similar to MessagePack, opens up the possibility to use the benefits of message passing across programming languages. Cloud Haskell can invoke remote processes whereas ØMQ needs means either inside Haskell (e.g. using Distributed Haskell as a basis [Huc99]) or outside Haskell (e.g. POSIX pthreads) to make this possible. Even though, ØMQ seems to be the better alternative since it is programming language ‘agnostic’, is harmonized with data (messages are treated as blobs) and can implement message brokers. For example message brokers can have routing, security and filtering functions or the message brokers could be one building block towards a ‘cloud service bus’ that controls the interaction of services, components and applications, somewhat similar to a SOA-like enterprise service bus.

3.1 CMQ Implementation

According to [BCvR09], “the cloud demands obedience to [its] overarching design goals”, and “failing to keep the broader principles in mind” leads to a disconnection of cloud computing research from real world computing clouds. Furthermore, scientists “seem to be guilty of fine-tuning specific solutions without adequately thinking about the context in which they are used and the real needs to which they respond”. One overarching design goal however is to avoid strong synchronization provided by locking services. Wherever possible, all building blocks of a computing cloud should be inherently asynchronous. CMQ, being designed to meet the real needs of cloud computing, is strictly asynchronous and is the combined research result from many different research fields, including network- and data- center design, network protocols, message-oriented middleware and functional programming languages.

CMQ is a lightweight message queue implemented in Haskell. The code is published on github.com and available at `https://github.com/viloocity/CMQ`. CMQ has currently three primitives: newRq (to initialize the queue and data structures), cwPush (to push a message into the queue), and cwPop (to pop a message from the queue).

3.1.1 cwPush

Messages for remote processes are identified by a key tuple consisting of the IP address of the remote system and an integer which is reserved for future use, for example, it can be used to specify the PID of the remote process. When a message is pushed with cwPush two things happen:

- The key-tuple and the data are stored in a map and implemented using the Haskell library Data.Map (a dictionary that is implemented as a balanced binary tree).
- The key-tuple and the creation time are stored in a priority search queue (PSQ), implemented using the Haskell library Data.PSQueue [Hin01] and used as a pointer to the corresponding binding in the map.

Figure 3.1 shows how CMQ works on the side of the sender. When cwPush is called to push a new message, a key-tuple k is built that consists of the IP address of the destination and a unique identifier (i.e. the PID). If the given key is not already a member of the PSQ, then a new binding (k, p) is inserted where the priority p is the creation-time of the binding. At the same time a new key-value pair (k, a) is inserted into the map, where a is a finite list that contains the pushed messages (Figure 3.1 (a)). Message queues are stored in the map structure and the map structure stores key-value pairs. The value of each key-value pair is a reference to a separate queue for a specific destination process.

If at the time when a new message is pushed its key k is already a member of the PSQ, the new message is appended to the end of the queue that corresponds to k (Figure 3.1 (b)). When the total amount of messages in a queue (the gross length of all messages) for a specific key-tuple exceeds a set threshold (qthresh) then the whole queue will be serialized and transmitted to the recipient (Figure 3.1 (d)). In order to ensure that messages only stay in the queue for a short time, a timeout threshold is used. No matter whether the data threshold qthresh is reached or not, once the timeout threshold is reached, all the messages in the queue will be serialized and sent once the timeout threshold is reached (Figure 3.1 (c)). The function sendAllTo from the Haskell library Network.Socket.ByteString is used to bring the UDP datagrams onto the wire. The function sendAllTo guarantees that all data are successfully brought onto the wire and that there were no errors on the local network interface.

Since CMQ is implemented in a pure functional programming language (Haskell), and pure functional data types are immutable, updating a node by writing directly to memory is not supported. The actual appending operation (++), which appends a new message to the end of a queue, does not update the tail node by changing its pointer so that it points to the new added message node, but recreates recursively each node in the queue, so that instead of writing a small node and a pointer to memory, the function returns, a complete new queue with the newly-added message returns [Unt09b], [Lip11]. This operation takes O(n) time.

It is observed that, while the appending operation has time complexity O(n), adding an new message to the head of a queue, by consing (cons :) the new message node directly to the head of queue, takes O(1) time. So an alternative method for the appending operation is to add a new message node to the head of a queue instead of the tail. The queue created using such a method maintains a reverse ordering of a FIFO queue. Before transmission of a particular queue, a reverse operation is performed on the queue to reverse the queue back to its normal FIFO form. The reverse operation takes O(n) time.

cwPush is implemented using two parallel threads, where thread1 enqueues messages and checks the total amount of messages; thread2 surveys whether the timeout for a particular queue is reached. By using time profiling (see Section on Messaging passing performance) it was discovered that thread2 was very costly and could use up 70% of the CPU time. As a result, a function called threadDelay was introduced to control and limit the maximum number of times that the PSQ is checked.

From the above discussion, we see that CMQ can be tuned using two parameters: qthresh (the maximum amount of messages in bytes allowed in the queue) and the timeout (the maximum waiting time a message stays in the queue before it is sent).

All map queries that are used in CMQ, including insertion and deletion, have a complexity of O(log n). A function called findMin is used to check the PSQ for any queues that have exceeded the timeout threshold. The findMin function is implemented with a complexity of O(1), which is an attractive feature, considering it is one of the most frequently used functions.

An alternative design solution is to use a more conventional method. Such a method, instead of using a map data structure with a PSQ as pointer, uses a sequence [Unt12a] of tuples (creationtime, message-queue) with each sequence data structure being responsible for a specific destination of messages. However, this method does not scale well. Although it is possible to examine the right (viewR) and left (viewL) end of a sequence with O(1) complexity, all sequence data structures require identification and organization, which will increase the complexity of queries and insertions to up to O(n) time. Thus, this method becomes inefficient when the number of sequences become very large, which unfortunately is a common case in cloud or large scale computing environments. Aiming for better scalability, CMQ is implemented in a way such that the identification information is maintained in the key k that associates queues with their creation time (in the PSQ) and recipients with their specific queues (in the map). Using the identification information, the system can quickly identify the queue for a newly pushed message. Since all map related queries take O(log n) time and all PSQ related queries take O(1) time, comparing with a sequence based solution, CMQ demonstrates a clear advantage in terms of its efficiency and scalability.

3.1.2 Petri net verification of cwPush

In this subsection we verify the concurrent and asynchronous implementation of cwPush using a classical Petri net.

According to Hoare, as cited in Hayman [Hay10], a feature of concurrent [and asynchronous] systems [such as CMQ and scalable software running on computing clouds] in the physical world is that they are often spatially separated, operating on completely different resources and not interacting. When this is so, the systems are independent of each other and therefore it is unnecessary to consider how they interact. Hayman further argues that Petri nets are an appropriate independence model that correctly describes the effect of events on local components of state called conditions making it possible to describe how events might occur concurrently,

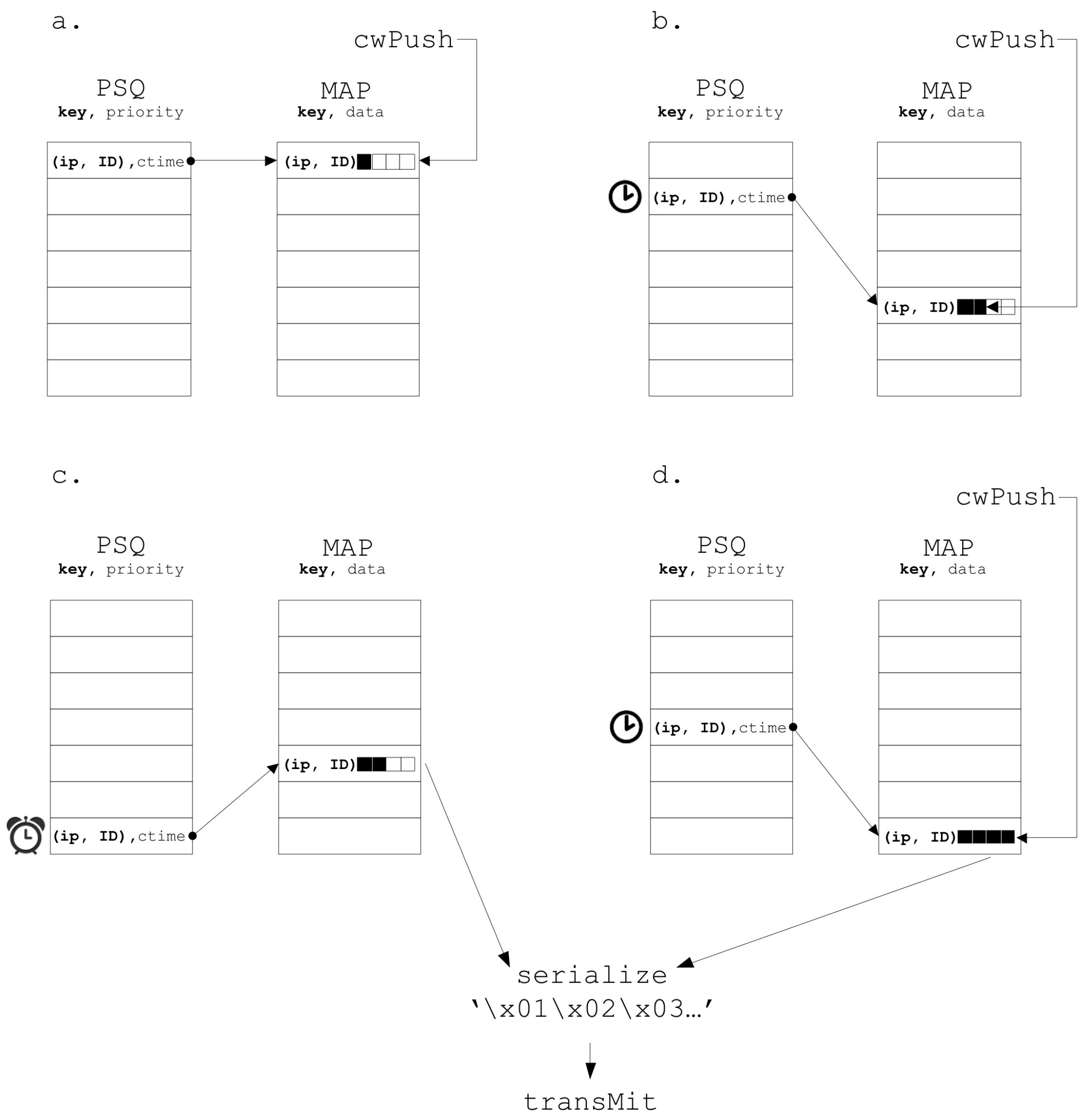


Figure 3.1: Map and PSQ pointer in CMQ. cwPush is called when the key for the recipient process is not present (a), cwPush is called when the key for the recipient process is already present (b), the timeout for a key has been reached (c), cwPush is called when the key for the recipient process is present and the data length amounts to qthresh (d).

how they might conflict with each other and how they might causally depend on each other. Petri nets are providing a formal model (the 'semantics') to developers to proof that a program is correct.

The semantic model essentially describes the behaviour that is illustrated Figure 3.1 and explained in Section 3.1.1. The Petri net semantic model gives less insight in the implementation details, such as the underlying Map and the PSQ, but focuses illustrating parallelization. Figure 3.2 illustrates a marking of the starting situation of cwPush described as a classical Petri net where the data is still with the application in place *p0*. The execution of the *transMit* function (in place *p5*) depends on the availability if two resources: data and time whereas the actual transition onto the network transition *t6*) depends on the presence of the network as a logical resource. This highlights and supports our finding that the availability of the network as logical resource is the most critical resource in play because it is a blocking resource that nihilates the earlier efforts of creating the two non-blocking threads *Thread1 and Thread2* that represent the actual data operations of the local queue.

The Petri net verification of our assumptions is useful demonstrates our implementation and our previous findings to be sound. The Petri net shows the parallel operation of the two threads and correctly identifies the network as most critical resource.

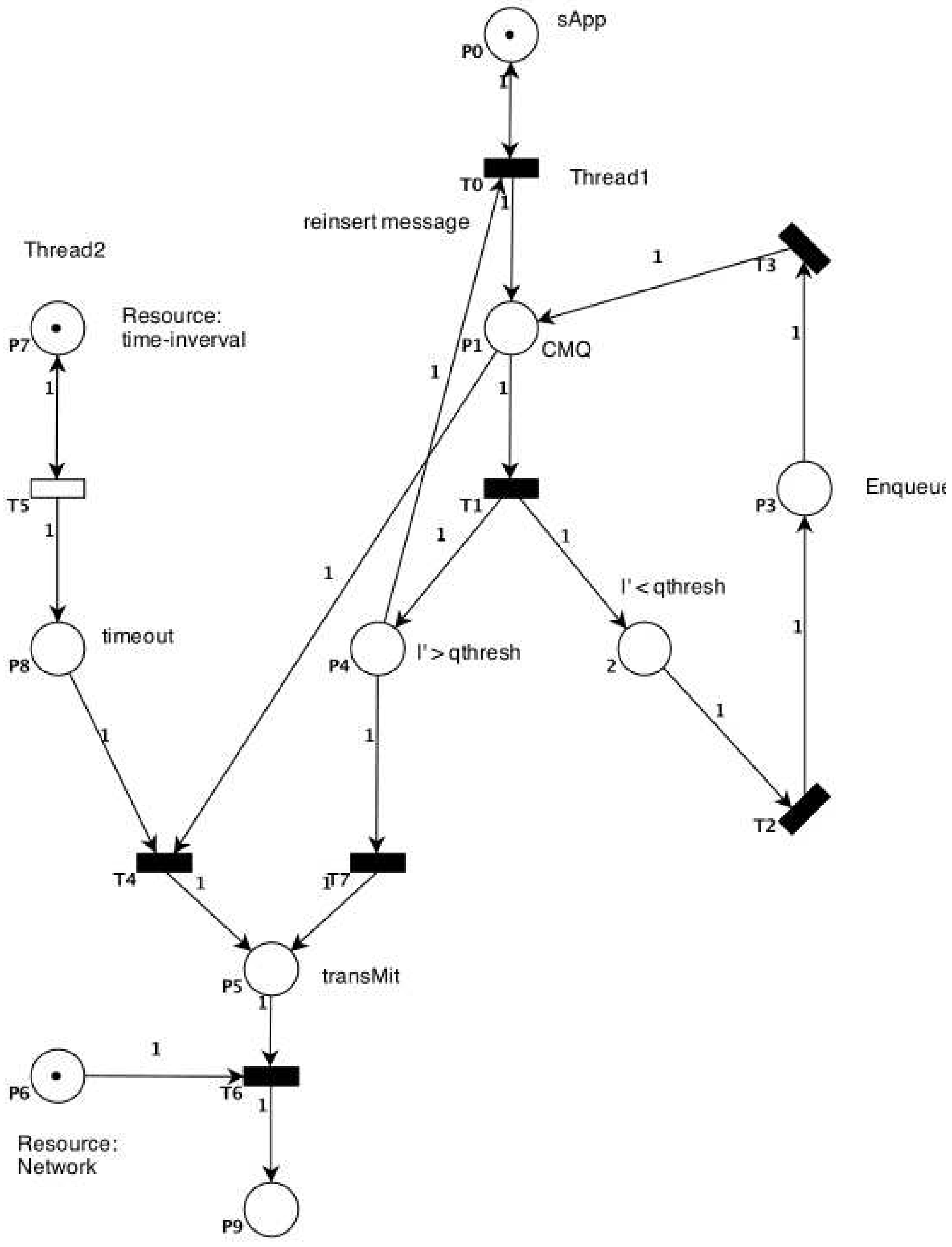


Figure 3.2: cwPush described as Petri Net

3.1.3 cwPop

On the recipient the serialized data structure with all its messages is received, deserialized, and transferred onto a transactional channel (TChan). TChan is an unbounded FIFO channel implemented in Software Transactional Memory (STM, [HMJH08]). Once the messages are transferred into TChan, they are ready to be consumed. The function cwPop is used to pop an individual message from the queue. cwPop is a non-blocking function that examines the TChan to check whether there are messages before attempting to read messages from the TChan. If there are waiting messages they are returned having the type Maybe String whereas in a blocking implementation the returned messages would have the type String.

Whilst in Erlang processes communicate with each other via mailboxes that are identified by the PID of the mailbox owner, in Haskell the preferred method for interprocess communication (IPC) are transactional channels TChan. TChan is created whenever it is needed. It has no dedicated owner and is not associated with any identifiers or addressing scheme. As a consequence, TChan is created by the developer and its identifier needs to be propagated. There have been some attempts to add additional layers of abstraction to TChan to make it work similar to Erlang mailboxes (e.g., Epass [Untd]) and more applicable to actor-based approaches. The majority of the attempts that are actually working and publicly available work only in local environment. Thus, they cannot send messages to a remote TChan . CMQ removes this imitation by allowing messages to be transmitted to a remote TChan via a CMQ queue.

3.1.4 The use of cwPush and cwPop

Listing 3.1 and Listing 3.2 give a simple example that demonstrates how cwPush and cwPop are used in a real application. Listing 3.1 shows an example of a sender application which sends 10000 messages each of which is contains a 4-byte string. The message type can be any Haskell data type that is a member of the Haskell serialize class. The application developer specifies the UPD port number (here UDP port 4711) to create the socket for UDP data transport. Listing 3.2 shows the example of the receiving application that uses cwPop to retrieve messages.

```

{-# LANGUAGE OverloadedStrings #-}
```

```
import System.CMQ
import Network.Socket hiding (send, sendTo, recv, recvFrom)
import Control.Monad

main = withSocketsDo $ do
      qs <- socket AF_INET Datagram defaultProtocol
      hostAddr <- inet_addr ''192.168.35.84''
      bindSocket qs (SockAddrInet 4711 hostAddr)
      (token) <- newRq qs 512 200 --initializes the queue with
          the desired parameters
      --qlength = 512B and max delay time in the queue is 200ms
           (minimum is 40ms)
      --token is the queue identifier where messages are sent
          to or poped off
      forM_ [0..10000] $ \i -> do
          cwPush qs (''192.168.35.69'', 0) (''ping'' :: String)
              token --send message ''ping'' to
          --ipv4 address 192.168.35.69 using the queue specified
               in token
```

Listing 3.1: Example of a sending application that uses cwPush.

```
import System.CMQ
import Network.Socket hiding (send, sendTo, recv, recvFrom)
import Control.Monad
import Data.Maybe

main = withSocketsDo $ do
      qs <- socket AF_INET Datagram defaultProtocol
      hostAddr <- inet_addr ''192.168.35.69''
```

```
    bindSocket qs (SockAddrInet 4711 hostAddr)
    token <- newRq qs 512 200
    forever $ do
       msg <- cwPop token :: IO (Maybe String)
       print msg
```

Listing 3.2: Example of a receiving application that uses cwPop.

3.1.5 Where is the queue?

There are two queues involved for every recipient process: the queue stored in the map on the sender and the TChan, which is in fact a simple STM-based FIFO queue on the remote recipient host. However, the detailed implementation is completely hidden from the users, who can see the CMQ message queuing system as a single distributed queue with two functions cwPush and cwPop. The function cwPush is called when a message is needed to be sent to a recipient, and the function cwPop is called when the recipient process reads a message.

3.2 Testbed specifications

The testbed consisted of a cloudstack [Untb] POD implemented on data centre grade hardware (listed in Table 3.1) analogous to commercial computing clouds. Figure 3.3 shows a logical diagram of two cloud computing nodes from our POD. The guest virtual machines (VMs) used for CMQ testing were resident on two separate computing nodes. Direct networking based on VLAN tagging is configured between the VMs and the physical networking gear. The VMs on our POD communicate via VLAN ID 2012, which is part of a VLAN trunk terminated on the computing nodes. The cloudstack (CS) virtual router is used to provide DHCP functionality and provide IP addressing to the VMs but, in this configuration, does not actually take part in routing and forwarding of packets.

The code that was used for benchmarking is published on github.com and is available at `https://github.com/viloocity/Haskell-IPC-Benchmarks`.

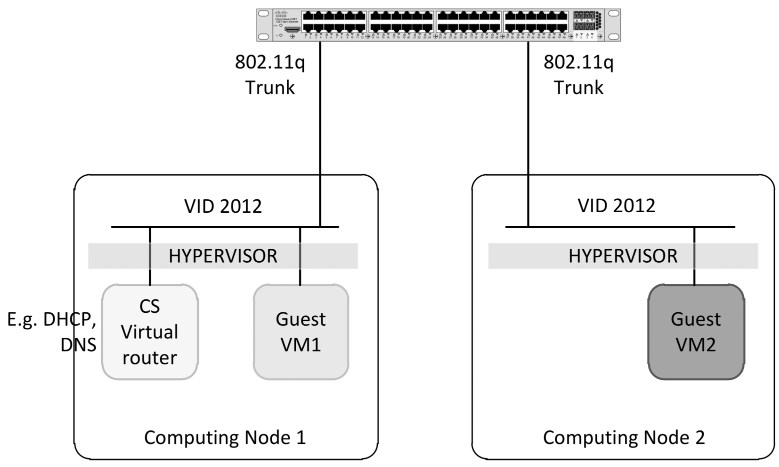


Figure 3.3: Logical diagram of the testbed configuration.

3.3 Benchmarking methodology

The Haskell benchmarking library Criterion [O'S15] is used for all the tests and a garbage collection was performed after every test. The CMQ testbed was setup in a client-server model where at first the client and the server would alternately send and receive messages similar to the ping pong test of the INTEL MPI benchmarks (IMB) [INT06]. In order to investigate the benefits of asynchronous message exchange (fire-and-forget messaging) and queuing, CMQ itself was allowed to use asynchronous non-blocking send operations (which means CMQ was allowed to send the next message before a reply to the previous message had been received) similar to the IMB ping ping test.

Table 3.1: CMQ testbed setup

	Hardware	Software
2 Server	HP DL 580 G5 4 quad core Intel Xeon E5450 @ 3GHz 32GB RAM	Citrix XEN 5.6
2 Linux VMS on separate XEN Server	ARCH Linux 2011.8, Haskell GHC 7.0.4	2 vCPUs @ 2GHz 4GB RAM
2 Fabric extenders	Cisco Nexus 2248TP-E	
1 Core switch	Cisco Nexus 7000	NXOS 5.1(2)

3.4 Evaluation criteria

ASM1	**Asynchronous I/O or Non-Blocking I/O**
References	[Bro11], [OT10]
Rationale	Asynchronous I/O, or non-blocking I/O, is a form of input/output processing that permits other processing to continue before the transmission has finished. Input and output (I/O) operations on a computer can be extremely slow compared to the processing of data. For example, during a disk operation that takes ten milliseconds to perform, a processor that is clocked at one gigahertz could have performed ten million instruction-processing cycles. [Wik15a]
Specifics	Typically, we think of I/O as ‘blocking’, for example working with files, databases or when data is serialized and transmitted to a remote process using TCP/IP, the process will wait for the transmission to finish before continuing, as the return value determines whether the transmission was successful. Blocking behaviour is in the way of scalable programming.
Ratings	1. The programming language does non-blocking I/O by default. 2. The programming model supports asynchronous I/O. An open/read/write operation on devices and resources (sockets, filesystem, etc.) does not block the calling thread. 3. There FPL is limited to the synchronous ‘C-like’ model.

ASM2	**Preventing Deadlocks**
References	[Sin89], [Mar12]
Rationale	Deadlocks consist of a set of processes each holding a resource and waiting to acquire a resource held by an other process in the set.

ASM2	**Preventing Deadlocks**
Specifics	The management of locks is important in concurrent, multi-threaded environments such as computing clouds. While it is difficult to measure how well a programming languages supports concurrency, it is comparatively easy to observe and to judge how deadlocks are prevented. Deadlock can arise if four conditions hold simultaneously: 1. **Mutual exclusion:** only one process at a time can use a resource. 2. **Hold and wait:** holding at least one resource and is waiting to acquire additional resources held by others. 3. **No preemption:** a resource can be released only voluntarily by the process holding it. 4. **Circular wait:** there exists a set {P0, P1, ..., Pn } of waiting processes such that: P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, ..., Pn-1 is waiting for a resource that is held by Pn, and Pn is waiting for a resource that is held by P0.
Ratings	1. The programming language implements concurrency models that prevent deadlocks (make it impossible to occur). 2. The programming language can deal with deadlocks, for example avoiding deadlocks (careful resource allocation), ignoring deadlocks (Ostrich algorithm), detect deadlocks (let them occur, detect, recover). 3. Unpredictable or avoiding deadlocks is possible but not practical.

ASM3	**Support for node-level shared nothing architectures**
References	[Hen06], [Sto86]
Rationale	Each node is independent and self-sufficient, and there is no single point of contention across the distributed system. None of the nodes share memory or disk storage.
Specifics	Applications in distributed architectures span across several systems (physical as well as virtual) or containers. Shared nothing architectures offer advantages that are sought after in computing clouds, for example systems clustering, eliminating single points of failure, self healing capabilities, non-disruptive hardware and software upgrades (also called 'zero downtime upgrades') but require appropriate communication with the remote unit of scale, such as processes running on the far system.
Ratings	1. The FPL provides rich and precise means to create shared nothing architectures, for example by closely integrating with a concurrency model that applies to local and remote processes. 2. Node-level shared nothing architectures can be implemented using third party libraries but they are not cost effective and not sufficient. 3. There is neither explicit nor implicit support of node-level shared nothing architectures.

ASM4	**Message passing**
References	[Vin07], [Hew10], [Agh86]
Rationale	Inter node messaging is required to transmit data, such as state information, and re-synchronize the workloads, such as functions, processes or micro services in distributed environments.

ASM4	**Message passing**
Specifics	Efficient message passing implementations to re-synchronize processes and transmit data across the distributed environment are required.
Ratings	1. The programming language has built-in support for inter-node message passing. 2. The programming language has support for message passing between local processes that can be extended to obtain inter-node message passing. 3. Third party message queues are the best alternative to obtain inter-node message passing.

3.5 Assessment of asynchronous operations and messaging in the functional programming language Haskell

Now, Haskell is assessed against the criteria developed in the previous section. The feasibility is demonstrate using the source code of CMQ, a UDP-based inherently asynchronous message queue. CMQ has been developed as artefact to verify the feasibility of using Haskell to support inherently asynchronous designs that perform especially well in modern Layer 2 switches in data center networks, as well as in the presence of errors. Taking the physical implementation of computing clouds into account we avoid falling for the eight fallacies of distributed computing and see whether Haskell abstracts these concerns away from the developer or leaves the issues up to the developer to solve by using for example external libraries or language extensions.

3.5.1 ASM1: Asynchronous I/O or Non-Blocking I/O

Haskell supports asynchronous I/O through [4]:

1. Strict separation of I/O and computation.
2. `forkIO`, a function that creates lightweight ”green threads” that are designated to handle I/O operations.
3. Software Transactional Memory (STM) that includes for example the `atomically` combinator and channels such as `TChan` that alleviates the composability of asynchronous elements.

Strict separation of I/O and computation

According to [OGS08] "Haskell strictly separates pure code from code that could cause things to occur in the world. That is, it provides a complete isolation from side-effects in pure code. Besides helping programmers to reason about the correctness of their code, it also permits compilers to automatically introduce optimizations and parallelism." I/O and side effects are considered by encoding them into values of the type `IO a`, such as the polymorphic identifier of the read/write queue `IO (Cmq a)` in listing 3.3.

Values of the type `(IO a)` cannot be casted into pure values: `unsafe :: IO a -> a`. However, this does not hold in practice since many Haskell applications (and modules) make use of the one or the other work around by using for example a mutable data type such as `IORef` to handle state or the function `unsafePerformIO` create for example global mutual variables. Furthermore Haskell’s Foreign Function Interfaces (FFIs) that is used to call functions that are provided by imperative languages frequently require developers to make use of mutual data types or the function `unsafePerformIO` thus diluting the experience of a pure

[4]Additionally the package `Control.Concurrent.Async` provides asynchronous actions `Async a` that correspond to threads by providing essentially an allegedly safe wrapper for threads. This Haskell package has not been investigated within the scope of this thesis where we address the most prevalent Haskell packages.

functional programming language that Haskell aspires to be. The authors of this thesis touch upon the argument whether a FPL has to be ‘pure’ in various places of this thesis.

From our own observations, we assume that it is much harder to run into issues with asynchronous code and concurrency with an essentially side-effect free programming language like Haskell than in programming languages with unrestricted side effects.

```

newRq :: Serialize a => Socket -- Socket does not need to be
   in connected state.
        -> Int          -- Maximum Queue length in bytes.
        -> Rational     -- Maximum Queue age in ms.
        -> IO (Cmq a)   -- Token returned to identify the
           Queue.
```

Listing 3.3: Type signature of the function newRq

Lightweight threads designated to handle I/O operations

Listing 3.4 shows source code of the function `newRq`, which initializes one instance of the message queue that sends and receives messages. Users can initialize as many instances as required, for example, one instance per application or peer. The message queue is instantiated using the `forkIO` function to create both a sending thread (3.4 line 7) and a receiving thread (3.4 line 8) demonstrating a basic case of performing two I/O operations in parallel and continuing those without blocking the main function.

```

2 newRq s qthresh tdelay = do
        q <- atomically $ newTVar (PSQ.empty)
4       m <- atomically $ newTVar (Map.empty)
        t <- newTChanIO
6       let cmq = Cmq qthresh tdelay q m t
        forkIO $ loopMyQ s cmq q m
```

```
      forkIO $ loadTChan s t
      return cmq
```

Listing 3.4: Initializing the asynchronous message queue CMQ

Software Transactional Memory

In the case of CMQ several threads work on the same data structures, a Priority Search Queue (PSQ) and a Map that are used as message store and to the message store. Haskell Software Transactional Memory (STM) uses transactions on memory similar to database transactions [HMPJH05] to provide consistency and isolation of data structures that might be accessed by several concurrent threads, eventually alleviating the need for locks. It must be stressed that the `atomically` combinator can in practice alleviate the need for threads that may otherwise would have multiple locks and thus become difficult to compose and a risk for the creation of deadlocks. STM abstracts these concerns away from the developer thus greatly reducing the burden on the developer.

For example, the `atomically` combinator, that is part of the Haskell `Control.Monad.STM` package, is used in Listing 3.5 line 4 - 9) to isolate the function `newTVar` from other threads while it is updating the PSQ and the Map. The update of the queue will only be done if the whole series of STM actions in the atomic block succeed. Otherwise the transaction will be rolled back.

```
  insertSglton :: a -> KEY -> TPSQ -> TMap a -> IO ()
2 insertSglton newmsgs key q m = do
        time <- getPOSIXTime
4       atomically $ do
        qT <- readTVar q
6       mT <- readTVar m
        writeTVar q (PSQ.insert key time qT)
8       writeTVar m (Map.insert key [newmsgs] mT)
        return ()
```

Listing 3.5: Inserting a new message into the queue

While green threads can be considered as a Haskell default, the use of STM is an optional trade-off between safety and speed. However, many blocking operations, such as network I/O or file activity cannot work under the arbitrary. Besides STM, Haskell has for example the already mentioned mutable data types and mutexes (locks) that are more efficient. The investigation of the efficacy of for example mutexes vs STM will be touched upon in the following section on deadlocks but is largely out of scope of this thesis.

Rating: 1. Haskell does non-blocking I/O by default.

3.5.2 ASM2: Preventing Deadlocks

Haskell can deadlock, even when the STM Monad is used. Haskell supports the detection of deadlocks and avoiding deadlocks through:

1. Deadlock detection, techniques to detect and debug concurrent programs.
2. Careful resource allocation, such as STM.

Deadlock detection

The Haskell platform includes options and packages that contain tools that can find deadlocks and debug threads, for example the Threadscope tool, functions that are part of the package `GHC.conc` and exceptions that are part of the Haskell base package `Control.Exception.Base`.

The Haskell package `GHC.conc` provides the function `threadStatus` that returns the detailed status of a (blocked) thread, for example: `threadBlocked [Blockreason]`. However, these functions are not useful for prevention of deadlocks in the control flow of an application since they break abstractions [Mar12] thus providing out of context exception that may refer to the implementation of a function or data type rather than staying on the level of the code that needs debugging.

The GHC runtime system can detect deadlocks and send the exception `BlockedIndefinitelyOnSTM` if the thread is blocked by an STM transaction or `BlockedIndefinitelyOnMVar` if the thread is blocked by a mutex. Rather than hanging in a deadlock state, in practice the program ends with an error message that can be used for debugging. Thus, deadlocks can be detected but not recovered.

Careful resource allocation

STM in Haskell prevents deadlocks but is still vulnerable to starvation [OGS08] and can livelock. STM is not a silver bullet and, for high concurrency systems, it leaves open many questions. For example, STM is an optimistic model that has no timeouts and may lead to slowdowns until a transaction finally goes through after many retries.

Potential alternatives to STM that let developers control thousands of freely interacting objects and libraries (that are in practice maintained by multiple developers in different corners of the planet) would be single threaded code, for example `node.js`. In chapter Coordination, we will further argue whether the abstractions that threads provide are essential or not. Additionally we also have to consider human dimensions, such as understandability of the software project and productivity of developers that confirm the feasibility of STM in high concurrency environments.

Rating: 2. The programming language can deal with deadlocks, for example avoiding deadlocks (careful resource allocation), ignoring deadlocks (Ostrich algorithm), detect deadlocks (let them occur, detect and recover).

3.5.3 ASM3: Support for node-level 'shared nothing' architectures

Shared nothing architectures for database systems have been under research since the year 1986 when Stonebraker [Sto86] found that node-level SN architectures have special merit compared to shared disk and shared memory architectures. Shared Nothing (SN) architectures mitigate the consequences of distribution, for example synchronization issues. Distribution has consequences on the following levels:

1. **Thread-level**, for example using STM instead of shared memory so that by design no thread can crash the other.
2. **Process-level**, for example Inter Process Communication (IPC).
3. **Node-level**, for example IPC, Message Queues, APIs (such as REST APIs) or Protocols (such as TCP/IP).

Node-level (server clusters) Share Nothing architectures help developers accounting for failures and mitigating failures by design. For more information on the importance of accounting for failure in computing clouds read also Section 3 Subsection The outward-looking perspective earlier in this chapter. It is possible of course that features that implement SN architectures at process-level or thread-level also implicitly implement SN architectures on node-level.

For example, the actor model [5] is often credited to implement a SN model on programming language level as well as in distributed environments. Examples of the actor models are Erlang/OTP message passing or the Akka library for the programming language Scala, a multi-paradigm programming language that includes functional elements.

However, Tasharofi et al [TDJ13], examines sixteen popular Scala programs published on Github that use actors, finds that only three programs use remote actors to implement distributed computation and concludes that “developers tend to use other ways than remote actors for implementing distributed computations”. According to Tasharofi developers frequently choose other ways to implement distributed computations because the absence of a means of coordination (of actors at large scale) and inadequacies in supporting asynchronous IO. Elements that the authors of this thesis have identified as essential for a FPL to have special merit in computing clouds.

Haskell supports node-level Shared Nothing architectures with:

1. Elimination of mutual data and state.
2. Third party message queues that developers can use to implement concurrency models across systems.

[5]The actor model is a concurrency model

3. Cloud Haskell, a proof of concept.

Elimination of mutable data and state

Haskell is a pure functional language where data is by default immutable. Copies of the data are shared rather than the data itself. This makes distributed programming cleaner and safer.

When bindings in functional data structure, such as a Map or a PSQ[6] are inserted, deleted or modified, strictly speaking, the returned data structure is not the original data structure but a data structure that is identical with the previous data structure but containing the alteration. Their insert, update and delete operations involve some degree of copying as opposed to typical mutable data structures where changes are written directly to the memory. To be more precise, a Map or PSQ insertion involves the copy of O(log n) amount of data for a data structure with n elements [Oka99] plus some additional logarithmic overhead [BJDM97]. It remains to be shown by future research whether or not a pure lazy language (e.g., Haskell) and its data types can retain the same asymptotic memory use as an impure strict one (e.g., Erlang) in all situations. However, in return functional data structures make it easier to keep multiple modified versions of the same data structure without storing whole copies.

Third party message queues

Several third party message queues provide Haskell bindings but they are not cost effective. Because protocols and message queues are at the center of distributed systems, we investigate message queues separately in Section 3.5.4 of this thesis.

Cloud Haskell

Cloud Haskell, proposed in [EBPJ11], aimed to further develop Haskell as a programming language for developing distributed applications. It was influenced by Erlang, and was intended to provide support for the actor model, message passing, and the mobility (with limitations) of

[6]Priority Search Queue

functions with co-located data (closures). It must be noted however, that, based on the experience of the authors, the ability to allow the migration of both, code and execution state from one executing unit to a different system (that is called 'code mobility') has to date no importance in cloud computing environments. Code mobility must not be mixed up with concurrency models that can cross the system boundaries, such as the actor model. These concurrency models are migrating data and not code.

Cloud Haskell, targeting language-level support for distributed applications, explores a lower, compiler-level implementation, while message queues and middleware are intended for a higher-level support. The benefit of a higher-level implementation is the flexibility that the approach is language-independent[7], can be easily adapted to other languages and environments, and thus serve the ultimate goal: finding appropriate communication approaches for cloud computing and the ubiquitous computing paradigm.

Cloud Haskell is currently still in proof of concept stage and most likely not ready for the adoption in large scale mission critical systems, such as computing clouds.

Rating: 2. Node-level shared nothing architectures can be implemented using third party libraries but they are not cost effective and not sufficient.

3.5.4 ASM4: Message passing

The relevance of message passing for computing clouds stems from the distributed programming model that is chosen to code either the cloud platform or the guest application. [Hew10] sees the Actor concurrency model [Agh86] as the foundation of cloud computing. The Actor model enables "asynchronous communication and control structures as patterns of passing messages" [Hew77]. Two well-known implementations of the actor model are e.g. the functional programming language Erlang [AVW93] and the Akka toolkit [Unta].

It may seem odd that this assessment talks about the benefits of UDP as message passing protocol when the focus of the evaluation is the merit of FPLs in the context of computing

[7]Actors could be written in a variety of languages while using the same message queue that is used to implement for example the actor framework

clouds. However, the design of the UDP protocol fits the already-mentioned notion of cloud computing well, that is, to accept occasional failure and manage it, rather than struggling to prevent it.

The increasingly wide adaptation of the UDP protocol indicates the suitability of the UDP protocol as an efficient transport protocol for supporting distributed applications. For example, UDP is used for data transportation in Network File System (NFS) and for state and event transportation in Massive Multiplayer Online Games (MMOGs) [WCC$^+$09], [Net]. EverQuest, City of Heroes, Asheron's Call, Ultima Online, Final Fantasy XI, etc. are among many MMOGs that use UDP as its transport protocol. The fact that MMOG applications are by nature of large, but elastic scale make them ideal customers for IaaS and PaaS offerings. By using cloud computing and storage facilities, not only cost and risks, that are usually linked to building new MMOGs, reduced [Sun10], but also over-provisioning MMOG hardware to be on standby for peak times will be avoided [MFD10]. However, whether computing clouds can fulfil the stringent real-time requirements of MMOGs is still an open issue [NPFI09].

In contrast to MMOGs that are largely event driven where the size of individual messages is expected to be small [Hen04], NFS is data driven with larger packet sizes and higher throughput. NFS, according to [KWW94], the most successful distributed application ever, has been using UDP as underlying transport protocol for more than two decades and was a stateless protocol up to NFSv3[8]. Compared with a large-scale cloud environment, NFS is arguably designed for a limited scale. The UDP based Data Transfer Protocol (UDT), described in [GG07], has showed the applicability of using the UDP-based UDT protocol for "cloud span applications" and won the bandwidth challenge at the International Conference for High Performance Computing, Networking, Storage, and Analysis 2009 (SC09). Furthermore, [RTLQ09] also discovers that reliable transport protocols that outperform TCP transport protocols can be designed in the basis of UDP.

While a UDP message queue for Actors is a new idea, UDP-based MOM (Message-Oriented Middleware) is not. The open source Light Weight Event System (LWES) [Unt09a] is a UDP-based MOM that is described as having a strong position in large scale, real-time systems that

[8]NFSv4 preserves state and is no longer built to deal with packet loss, thus requires the TCP protocol.

need to be non-blocking and is also described by Yahoo! as part of US Patent 2009/0094073 ”Real Time Click (RTC) System and Methods”. LWES is also described as being useful (for transporting large data to computing nodes) for parallel batch processing with Hadoop [BCC12], which is an open source implementation of Google's Map Reduce [DG08a]. In fact, MapReduce and Hadoop are posited, in [FZRL08], as the right cloud computing programming models.

Haskell has message passing support; such message passing channels (read also section Careful resource allocation). However, these means are implemented to exploit multi core architectures[9] on the node and have no built-in support for physically distributed environments such as computing clouds. The Haskell Wiki[Has12], that is the main source for documentation of the Haskell FPL, gives a 'trivial chat server' as an example to show how inter-node message passing with multiple connected users can be obtained using channels. The authors of this thesis extend on this example to evaluate if Haskell has special merit for obtaining an efficient lightweight inter-node message queue for communicating processes.

We find that Haskell has moderate merit for creating an efficient lightweight inter-node message queue with the following features:

1. Serialization libraries.
2. Resilience under error conditions.

Serialization libraries

Time and space profiling of the CMQ message queue indicated that serialization is the major concern. The heap profile Figure 3.4 is split into the 20 most prominent cost centres as inserted by the compiler (qthresh was set to 512K and timeout to 200ms. The maximum length tested was five hours where the pattern could be sustained). The memory consumption of 80K is lean and we find that the memory consumption related to serialization cost (the band labelled as 'PINNED') is the most prominent feature. In CMQ a message queue that consists

[9]or streaming data from files

of multiple messages is serialized, but these efforts do not seem to significantly alleviate the overhead of serialization.

The nominal performance of the Haskell serialization libraries depends on the functional data structure that needs be be serialized for example, serialization of polymorphic data structures or maps is slower than the serialization of simple data structures. The messages used in our performance tests are composed of only 8-bit ASCII characters and the Haskell library `Data.Bytestring.Char8` is used for the serialization. For messages or objects with other character encodings, the Haskell `Data.Bytestring` library or non-native serialization libraries that provide Haskell bindings, such as MessagePack, may be used for the serialization.

Although for the performance tests we send messages composed of 8-bit ASCII characters, CMQ is internally built with polymorphic functions and can transfer arbitrary Haskell Data Types under the condition that they can be serialized. In order to achieve polymorphism, CMQ must compare the queue length to qthresh when the queue is serialized, since functions that can determine the length of an ASCII based queue do not fire any more under these circumstances. Thus, more serialization activities are necessary compared to an implementation that would be limited to the data type String. Overall, there was no evidence of space leaks that is a measure of code quality.

Message passing performance. It is difficult to make generalized statements whether the native serialization libraries in Haskell are competitive with for example serialization in other programming languages.

To get a quantitative indication we used the same Haskell in a cloud computing test bed (illustrated in Section 3.2, Figure 3.3 and Table 3.1) to benchmark three message passing architectures and paradigms. MessagePack [Unt12b] and 0MQ [Untc] are chosen in the CMQ performance evaluation. MessagePack and 0MQ are two IPC systems that provide Haskell bindings. MessagePack is a library that is based on RPC and focuses on object serialization. 0MQ provides a framework that focuses solely on message passing and queuing.

MessagePack uses RPC to transfer messages that, in fact, are all serialized objects, and was initially described as IPC to ”pass serialized objects across network connections” [Unt12c].

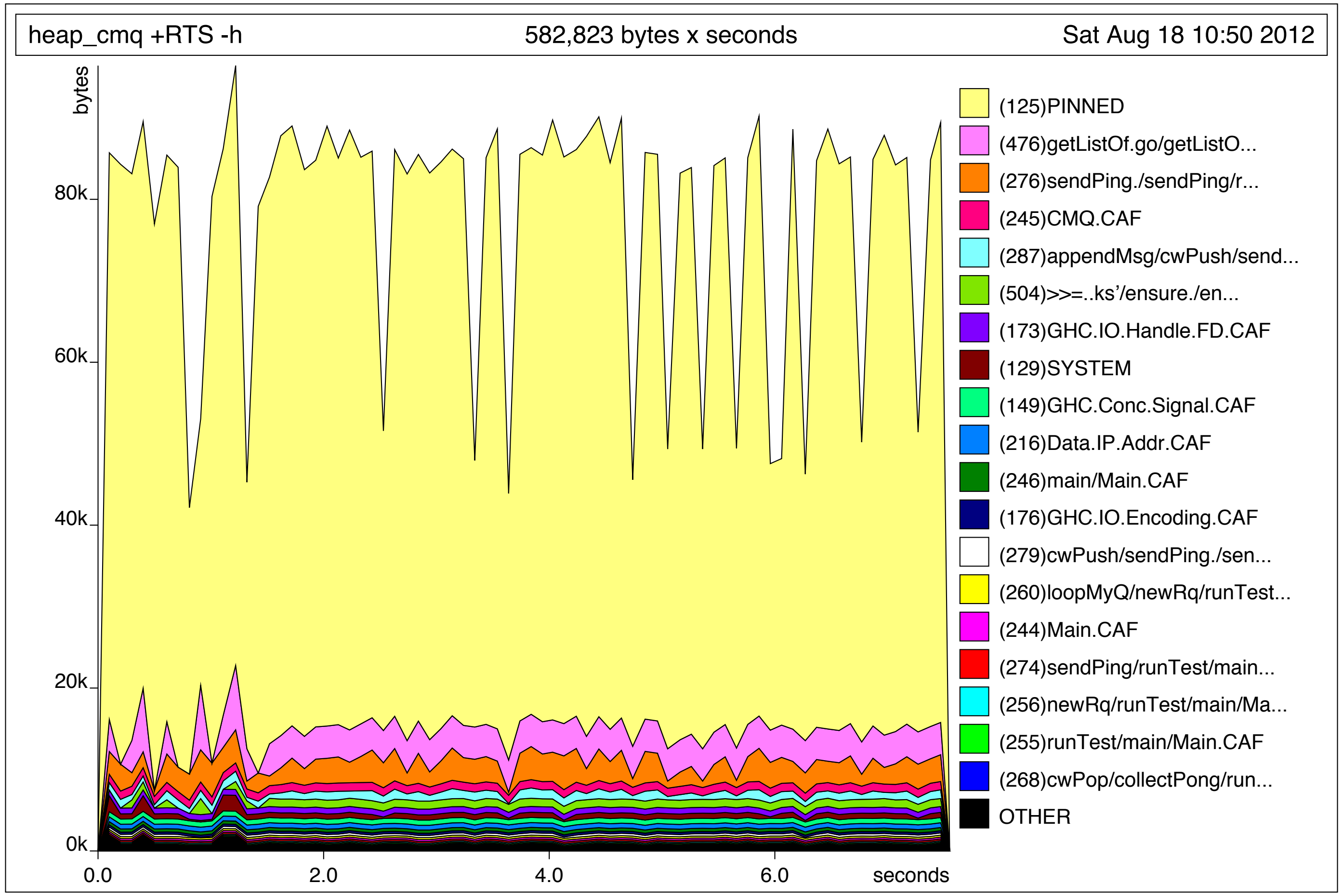


Figure 3.4: Graph for memory usage on the heap. The heap is split into the 20 most prominent cost centres as inserted by the compiler. qthresh was set to 512K and timeout to 200ms. The maximum length tested was five hours where the pattern could be sustained..

Although the most recent descriptions of MessagePack focus mainly on its outstanding object serialization capabilities, it serves also as a general message passing mechanism.

Mean performance. By and large, the story of our efforts revolves around qualitative measurements of the networking and serialization performance of different implementations and bindings in Haskell. Our benchmarks address raw message passing performance where serialization is tested implicitly. Other test data and code may yield different results. However, our results give a raw estimation of serialization performance.

Figure 3.5 presents the mean performance of UDP Sockets, MessagePack, 0MQ and CMQ for exchanging 1000 messages with message sizes between 4 B and 16 KB. UDP Sockets are not included in Figure 3.6 for the reason that the benchmarking application used for UDP sockets supports only synchronous operations in lossless environments. The test results show that when message sizes are less than 1 KB UDP sockets perform comparable to TCP-based messaging queues; when message sizes are larger than 1 KB, UDP sockets outperform all tested TCP queuing methods. As for CMQ, it, in general, outperforms all other messaging queuing methods. CMQ demonstrates a clear advantage for small to medium sized messages up to 4KB. It shows a speed increase of up to 100 times for the transmission of small messages such as integers (e.g. error codes), flags or applications that need only a single request - response [Pes06], since TCP messaging requires the establishment of a TCP connection which would incur a 60% overhead for a small sized message. From the tests it was discovered that CMQ achieves its best performance with a qthresh of 512B (a value that is also used by the DNS protocol) and a queue timeout threshold of 200ms.

Figure 3.5 shows that this optimisation makes invoking CMQ to send messages from Haskell faster than simply sending UDP packets from Haskell provided that the message size is less than approximately 8KB. However, the length of time taken to send UDP datagrams appears to be extraordinarily high. This leads me to be concerned that the numbers are telling us more about the benefits of avoiding the implementation or API to UDP in Haskell rather than the benefits of the CMQ design and implementation in general.

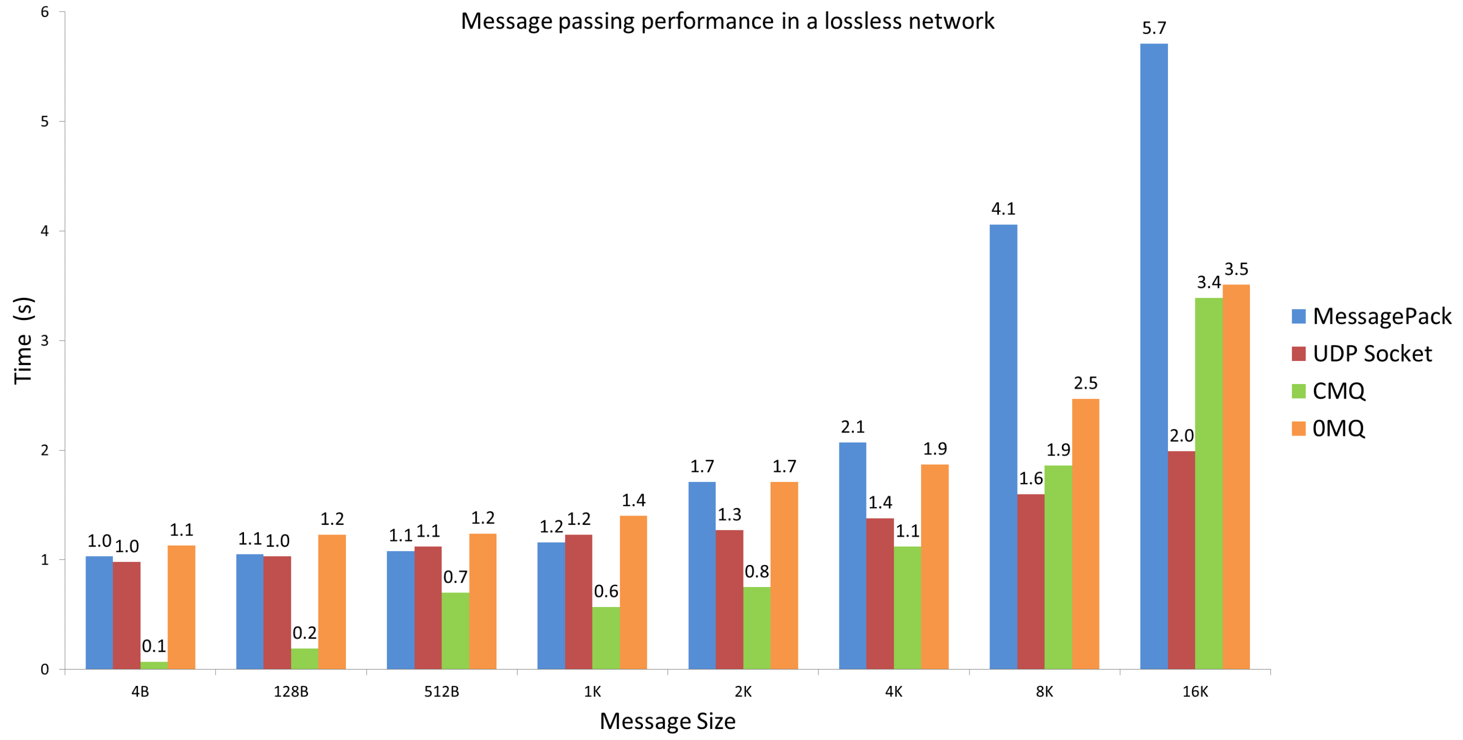


Figure 3.5: Message passing performance on a lossless data center network

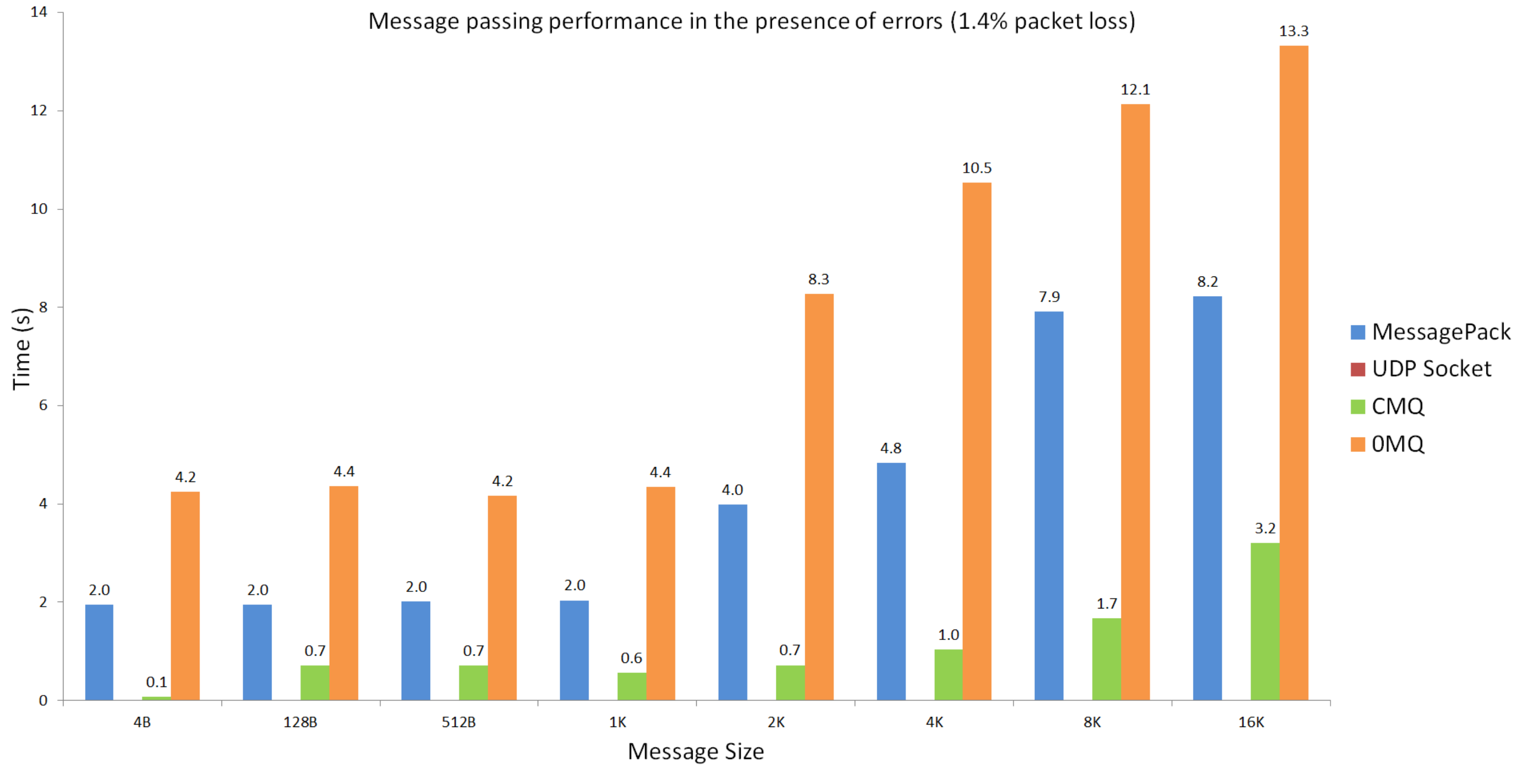


Figure 3.6: Message passing performance on a data center network with 1.4% packet loss

Reality-check. To give a more nuanced flavour of the sorts of problems we encountered, we have chosen to share a few in more detail.[10].

Figure 3.5 shows sending 1000 messages using UDP sockets (no CMQ) taking between 1.0 seconds and 1.2 seconds for messages of size 4 bytes through 1Kbytes. This gives a packet-send time of about 1 ms. Testing on consumer-grade hardware (a T400 Thinkpad and an Intel Q4800 4-core PC) using fast Ethernet connectivity, wireless connectivity and localhost connectivity, we get packet-send times of between 10 times and 100 times less than that. For example, using the open source network benchmarking program `nutccp` over 100 Mbps Ethernet gives UDP send times of 6.8 µs for 5-byte messages rising steadily to 46.2 µs for 512-byte messages to 87.2 us for 1024 bytes. At 512-byte packets, for wireless (limited to about 14 Mbps for my network), I get 280 µs and for localhost (avoiding most of the networking stack and all the physical network but leaving the network API latency) we get 7.14 µs per packet.

Although `nutccp` is compiled code, it remains unanswered whether the increased latency displayed in Figure 3.5 can be best explained with the use of a high-level language, the overhead of serialization (as indicated in our findings) or other inefficacies that are inherent to the underlying Haskell UDP API.

A quick experiment using Perl as a high-level language to send 100000 UDP datagrams over the above wireless network and receive them using `netcat` and `awk` gives the same figure of 280 µs per packet. This is using the following receiver on the PC:

```
% nc -lu 1234 | awk '{n++}($1 == "start"){s = systime();
print "start=",s}($1 =="end"){e = systime();
print "end=",e,"delta=",e-s,"count=",n;exit}'
```

and sending with the following Perl commands, that can in practice be expressed as one-liner:

```
% perl -MIO::Socket::INET -le '$msg = "a"x511;
$s =IO::Socket::INET->new(PeerAddr => "10.0.0.1", PeerPort => 1234,
Proto =>"udp");
```

[10]These measurements have been contributed by an anonymous reviewer

```
print $s "start";
for($i=0;$i<100000;$i++){print $s $msg} print $s "end"'
```

This produces the output:

```
start= 1343748860
end= 1343748888 delta= 28 count= 100002
```

This verifies the 280µs per packet and a `tcpdump` showed the required 512-byte UDP datagrams being sent. The majority of that 280 µs is from the slow speed of the network medium and indeed the same experiment done on just the Thinkpad pointing at localhost show packet times of 13us (with 0.9% packet-drop). Since Figure 3.5 shows 512-byte UDP messages taking 1100 µs rather than times such as 13-280 µs that we measured (albeit simplistically and not in our cloud computing testbed), our concern was that the actual subset of that time taken up with the UDP implementation is a very small proportion of the overall times measured.

More detailed measurements, such as the heap and time profiles (for more details see also Figure 3.4) were done to show the difference between the times spent in implementation-specific UDP wrapper code and the times spent in unavoidable (or not easily avoidable) network APIs and network send times. All findings indicated that serialization, that is unavoidable for complex data types, was the most prominent performance bottleneck requiring further expert diagnosis.

In spite of these difficulties, we are satisfied with the performance we have achieved to date.

Resilience and temporal predictability

The need to investigate resilience stems from two considerations:

1. ”The network is reliable” is the first fallacy of distributed computing–for more information read the section Testbed specifications of this thesis.
2. More recent and immediately related to computing clouds Birman and Chockler [BCvR09] state that currently “Not enough is known about stability of large-scale event notification

platforms, management technologies, or other cloud computing solutions" and identifies the development of testing methods that can validate the relevance and demonstrate the scalability of any new solution . . . without working at some company [e.g. Yahoo, Google, Amazon] that operates a massive but proprietary infrastructure" as an item on the cloud computing research agenda.

TCP is a reliable protocol that provides reliable, connection-oriented delivery of data. It detects for example packet loss, delay, congestion and replays lost packets when required. However, the reliability causes a significant overhead, especially for messages of small sizes. This is one of the disadvantages of using TCP. More seriously, when delay or packet loss are detected, TCP assumes congestion and slows down the rate of outgoing data [Pes06]. [MSMO97] and [SW08] propose formula to calculate the effective bandwidth of TCP connections in the presence of errors where, for example, a 0.2% packet loss eventually slows down and limits the effective connection speed to 52.2Mbps irrespective of the nominal bandwidth. In practice, retransmitting packets is very costly since it also involves queuing and reordering packets that arrive until the re-transmit is complete, thus stopping time-sensitive data from going through in the meantime [Fie09]. Furthermore CPU usage spikes when TCP retransmissions are needed and applications frequently become unresponsive.

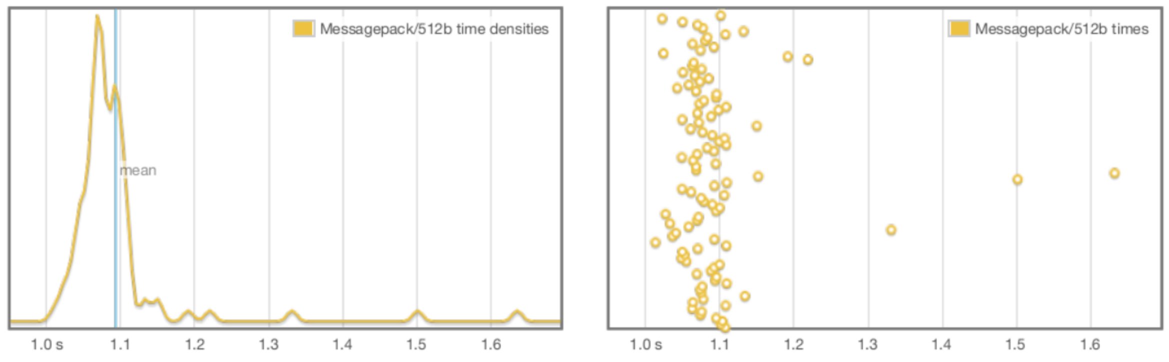
We measured the Ping–Pong message passing performance using message sizes of 4 byte, 128 byte, 512 byte, 1Kb, 2Kb, 4Kb, 8Kb and 16Kb in a loss less data center network built with state of the art data center hardware (for more information abut the testbed read Section 3.2 of this thesis) and in an unreliable network based on the same architecture where the packet loss is simulated with iptables [Unte] (see command listed below) on one of the Linux VMs to drop incoming packets with 0.5% probability.

Figures 3.5 and 3.6 illustrate that the impact of packet loss on different message passing paradigms –except for UDP based messaging– brings about unacceptable delays and consequences for the operation of computing clouds and the deployed applications.

```
iptables –A INPUT –m statistic ––mode random ––probability 0.050 –j
    DROP
```

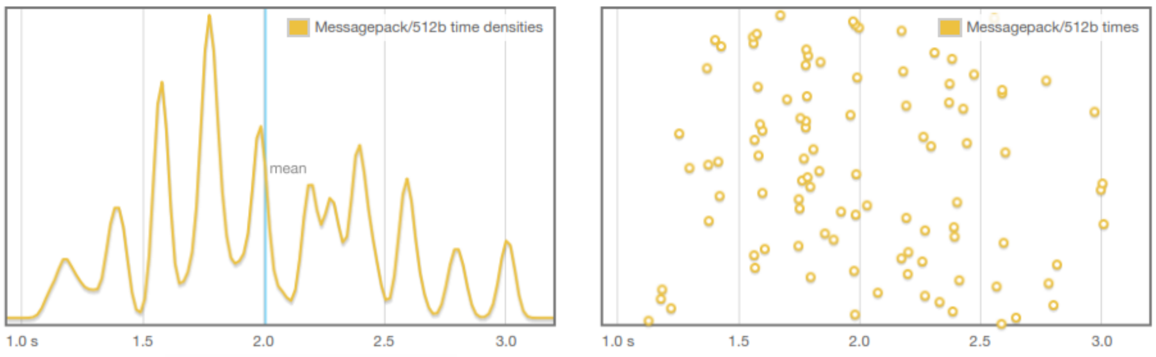
We found that CMQ is largely unaffected and produces the same performance results (within the standard deviations) as if there were no errors on the network. All other benchmarked queuing methods show an overall delay of approximately a factor of 4 and became largely unpredictable. It was also discovered that in the presence of errors the benchmark results of CMQ still show a narrow standard deviation. For example, for a message of 512 Bytes, the standard deviation of CMQ stayed at 52ms whilst the standard deviation of MessagePack increased from 54ms (with no simulated data loss) to 820ms.

Figures 3.7, 3.8 and 3.8 are Criterion micro benchmarks that depicture the raw measurements on the right and on the left a kernel density estimate (KDE) [Wik15d] of time measurements. Measurements are displayed on the y axis in the order in which they occurred. This graphs the probability of any given time measurement occurring. A spike indicates that a measurement of a particular time occurred; its height indicates how often that measurement was repeated [O'S15]. Both parts of the graphs show that in unreliable networks the UDP based CMQ has much higher temporal predictability. The associated graphs are sharper bounded, that means it gives only a small set of possible outcomes, which could be a few possible results or results within a small range.



	lower bound	estimate	upper bound
Mean execution time	1.082 s	1.093 s	1.115 s
Standard deviation	54.52 ms	79.21 ms	125.2 ms

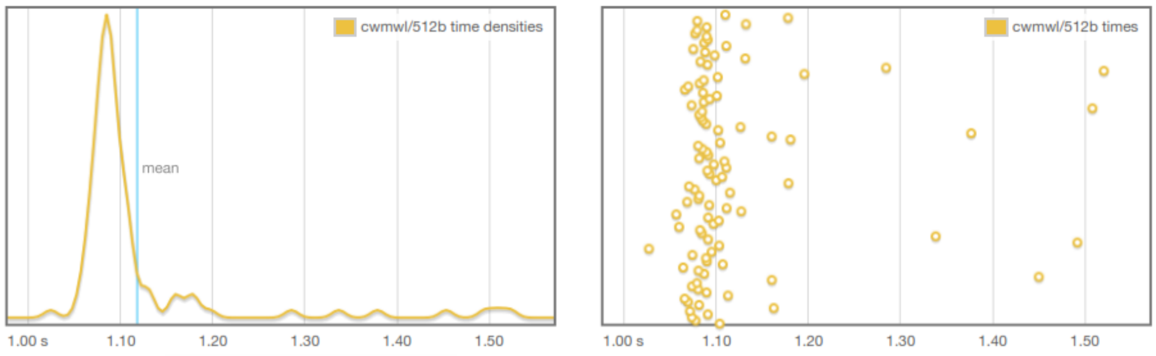
Figure 3.7: MessagePack benchmark for 1000 send/receive cycles with 512 byte payload in a "lossless" datacenter network.



	lower bound	estimate	upper bound
Mean execution time	1.922 s	2.007 s	2.077 s
Standard deviation	422.8 ms	465.8 ms	509.4 ms

Outlying measurements have severe (0.9573261008061968%) effect on estimated standard deviation.

Figure 3.8: MessagePack benchmark for 1000 send/receive cycles with 512 byte payload in a network with 0.5% loss.



	lower bound	estimate	upper bound
Mean execution time	1.111 s	1.118 s	1.130 s
Standard deviation	58.39 ms	91.42 ms	119.0 ms

Outlying measurements have severe (0.7175424658674567%) effect on estimated standard deviation.

Figure 3.9: CMQ Benchmark of 1000 send/receive cycles with 512 byte payload in a network with 0.5% loss.

In this category Haskell does not have enough merit to achieve higher ratings. The extension of the local support of message passing into physically distributed environments is possible but less feasible than the use of state of the art message passing libraries and frameworks.

Rating: 3. Third party message queues are the best alternative to obtain inter-node message passing.

3.6 Intermediary result

Table 3.2 summarizes the ratings of the Haskell programming language in the evaluation criteria category Asynchronous operations and messaging.

Table 3.2: Rating of asynchronous operation and messaging in Haskell

Requirement	**Rating**	**Details**		
ASM1 Asynchrounous, non-blocking IO	1 The programming language does non-blocking IO by default.	Strict separation of IO and computation	Lighweight ‘green’ threads for IO operations	Alleviating composability of asynchronous elements
ASM2 Preventing Deadlocks	2 The programming language can deal with deadloacks, for example avoiding deadlocks (careful resource allocation)	Deadlock detection, techniques to debug deadlocks, no runtime recovery	Careful resource allocation, such as STM	
ASM3 Support for node-level shared nothing architectures	2 Node-level shared nothing architectures can be implemented using third party libraries but they are not cost effective and not sufficient.	Elimination of mutual data and state	Third party message queues	Cloud Haskell not production ready, no ongoing development
ASM 4 Message passing	3 Third party message queues are the best alternative to obtain inter-node message passing.	Serialization libraries not very effective. Needs expert debugging.	Resilience under error conditions up to developer or third party frameworks.	

3.7 Summary

We find that in computing clouds, the FPL Haskell needs to support node level shared nothing architectures and provide inter node message passing. We present the lightweight, UDP based message queue CMQ. The concept to use UDP instead of TCP is motivated by our understanding that, in Cloud Computing, omnipresent off-the-shelf technologies (both in hard- and software) are encouraged, and if preventing errors from occurring becomes too costly, dealing with the errors may be a better solution. This chapter has demonstrated the capability of using UDP for message queuing in the presence of errors, and has shown the stability of UDP messaging in such conditions. Methods that deal with packet loss at the application level are also discussed. The implementation of CMQ is a Haskell Module that utilizes pure functional data structures. The implementation of CMQ is available as module System.CMQ from the hackageDB[11], and also can be installed automatically via the Haskell package manager cabal on the Haskell Platform.

Although CMQ is a message queue oriented communication approach, CMQ is different than the conventional MOM approach because it challenges a number of assumptions under which conventional MOM is built. For instance, in conventional MOM, messages are 'always' delivered, routed, queued and frequently follow the publish/subscriber paradigm. It is often accepted that this requires an additional layer of infrastructure and software where logic is split from the application and configured in the additional layer. On the contrary, CMQ does just enough. It does not offer guarantees, thus is very lightweight with low overhead and fast speed. Although it does not offer guarantees, such as reliable transmission, it appears to be stable in the presence of errors.

[11]`http://hackage.haskell.org/packages/hackage.html`

Chapter 4

Coordination

In this chapter we assess whether functions in the FPL Haskell can serve as easy access composable lightweight units of scale to develop scalable software that executes on computing clouds. We introduce a new PaaS environment that combines the LINDA coordination language, an in-memory key-value store, with functional programming to preserve state and facilitate execution and coordination of functions. In our architecture a tuple space is a central element to support deterministic services for basic parallel programming, including message passing, persistent infinite message pools and transactions. Redis, a key-value store, serves as the in-memory tuple space that glues together parallel constructs (i.e. skeletons) of formerly monolithic business applications to form an elastic distributed application.

Traditionally scalable software deployed on computing clouds is developed and implemented using two units of scale:

1. Virtual Machines and Containers generally consisting of one JVM, Rails, or some other runtime system per container in which the guest code is deployed [VRMB11], [CBP$^+$10].[1]

[1]Under rare circumstances, such as with the execution in the Apache MESOS platform, this can also be a Java archive.

to achieve scalability and fault tolerance across multiple virtual or physical systems. Applications provisioned into VMs or containers is portable between physical systems but as units of scale they are still coarse and not efficient.

2. Threads to achieve parallelism and scalability across multiple CPU cores on the (guest) system. Amongst other issues, threads have in practice no portability beyond the physical platform.

Virtual Machines and Containers scale guest code through replicating the VM or container and subsequently create the illusion of seamless scaling by clustering the replicas with a load balancer. It must be noted, that although both VMs and containers scale software by replicating coarse units of scale, VMs and containers are not the same. Containers are a light weight operating system virtualization technology the provisioned software shares the same kernel and, depending on the configuration, large parts of the OS (illustrated in Figure 4.1). Table 4.1 compares the efficacy of different virtualization paradigms, IaaS (that is VMs) are in the second row whereas OS virtualization (that is Containers) are in the third row. It is visible that containers have less overhead, and in practice they can be instantiated within milliseconds whereas instantiating a VM may take minutes.

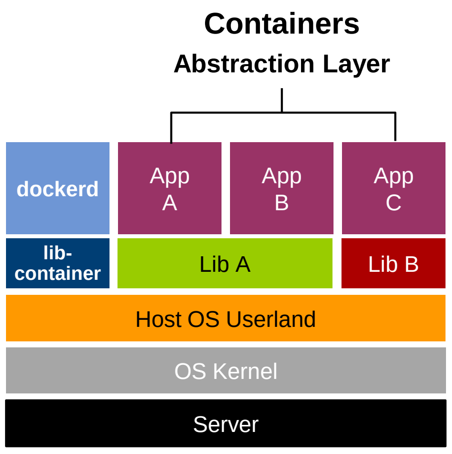


Figure 4.1: OS-Level Virtualization Managed by the Docker Daemon (Source: Gartner 2015).

Considering that at least one instance of every guest application must be running [2] and that frequently additional components such as the host operating system and the application execution engine need to be replicated as well, it becomes clear that modern PaaS clouds have a huge footprint and the size of the infrastructure is still dictated by the number of clients and their running instances rather than by the actual load [SK12]. Table 4.1 compares six distributed computing architectures and the resources required, such as number of deployed operating systems, to deploy three application instances. Linux containers running without `systemd` need the least resources to deploy three application instances and have thus a higher efficacy than for example IaaS or hardware load balancing.

Threads are the prevailing parallel execution paradigm, also in functional programming languages that have recently adopted new runtime technologies to achieve parallel execution. Threads are a very small unit of scale that commonly has no mobility beyond the physical platform where it is started and there is no obvious way to match functions and threads. Lee [Lee07] is generally sceptical about threads as an effective means to support parallelism and argues in favour of coordination languages instead. He writes:

> If we expect concurrent programming to be mainstream, and if we demand reliability and predictability from programs, then we must discard threads as a programming model. Concurrent programming models can be constructed that are much more predictable and understandable than threads.

Lee argues further:

> The message is clear. We should not replace established languages. We should instead build on them. However, building on them using only libraries is not satisfactory. Libraries offer little structure, no enforcement of patterns, and few composable properties. I believe that the right answer is coordination languages. Coordination languages do introduce new syntax, but that syntax serves purposes

[2]Starting a 'legacy' application instance in a cloud takes 30 seconds to one minute, making a 'start on demand' paradigm unfeasible. OS-less instances, such as Linux containers, can be started within milliseconds using coordination languages or a coordination service.

> that are orthogonal to those of established programming languages. Whereas a general-purpose concurrent language like Erlang or Ada has to include syntax for mundane operations such as arithmetic expressions, a coordination language needs not specify anything more than coordination.

While Lee investigates threads and coordination, he does not investigate suitable units of scale and keeps his research in this area rather generic using ‘processes’ as the underlying unit of scale for his investigation. We extend Lee’s research by combining our idea of functions being an appropriate unit of scale for software in computing clouds with Lee’s findings, therefore we assess if the FPL Haskell can be used to obtain a shallow embedded domain specific coordination language that provides a set of primitives for coordination [3]. A shallow embedded domain specific language[4] is chosen to fulfil Lee’s requirements to:

- Build on existing programming languages rather than replacing them.
- Go beyond the capability of programming language libraries.
- Avoid threads as units of scale.
- Provide a set of primitives for coordination.

Elastic and scalable software in computing clouds requires additional tools for dynamic service registration and service discovery. Well known tools, for example Apache ZooKeeper [HKJR10] , serf [ser15], etcd [etc15] or the use of a traditional DNS based infrastructure as described by Spotify Labs [spo15], are commonly referred to as ‘coordination services’. Coordination services have workloads (also referred to as ‘services’) determine the IP address, port and capabilities of services in the distributed environment so that they can interact. Coordination Services are not in scope of this research.

[3]resulting in a distributed architecture that is concurrent and non-deterministic

[4] An alternative to a DSL could be for example a well structured API that is implemented with / on top of the coordination mechanism. However, APIs follow an imperative design, for example, a REST API would need to be called in an imperative manner and hardly make use of any of the functional paradigms that we are investigating in this thesis. Strictly speaking, the DSL we implemented during our research represents an API.

Efficacy is a concern in computing clouds. Corporations, service developers and end users are not interested in information technology [Car05]. Their goals are, for example, to run a business process that requires certain resources (e.g. computation and storage), to use an existing application or service, or providing them for third parties. With the advent of virtualization technology and Infrastructures-as-a-Service (IaaS) corporations began to eliminate overcapacity in terms of available computing power. Platform-as-a-Service (PaaS) is the cloud computing layer that further reduces redundant functionality–and, allegedly resources. Table 4.1 contrasts and compares the efficacy of the available virtualisation technologies for three deployed application instances. It is visible that IaaS, the currently prevailing technology is at the same time one of the least efficient technologies[5].

To date improvements in efficacy are generally sought in reducing the Power Usage Effectiveness (PUE)[6] by improving of for example data center hardware and cooling. Efficacy of software and application (architectures) is not considered. While it is undoubtedly true that data centers must be built ‘green’ and that local legislation frequently limits the data center PUE to a very low maximum, efficient software would certainly help a lot to reduce the overall footprint of applications and data centers by increasing the application density per server.

4.1 High level design of Cwmwl PaaS framework

The Cwmwl PaaS framework is a flat fabric that unifies the traditional data-tier, messaging, and computation. We share the views of Shalom [Sha06] who states that the emulation of tier-based computing in the cloud or in PaaS platforms does not scale effectively enough to the cloud scale. A unified PaaS fabric scales more predictably than traditional web applications (e.g. a LAMP stack) that need a mix of technologies and protocols to scale each tier separately. In our approach a single unified communication protocol is used to store data, coordinate computation, and exchange information about state. Static (partially hierarchic) relationships between web-frontend servers, application servers, message buses and database servers are removed.

[5]Definition and underlying architecture of SaaS offerings are not clear cut and out of scope of this assessment.

[6]The Power Usage Effectiveness is defined as the total facility energy divided by the IT equipment energy. Regulations may require PUEs of data centers to be close to 1.2 or smaller.

The key differences between a conventional PaaS framework and the unified Cwmwl PaaS fabric can be observed from Figures 4.2 and 4.3. A conventional PaaS framework scales horizontally through replication of the runtime containers for applications, while the Cwmwl PaaS fabric scales through distribution of the application instances.

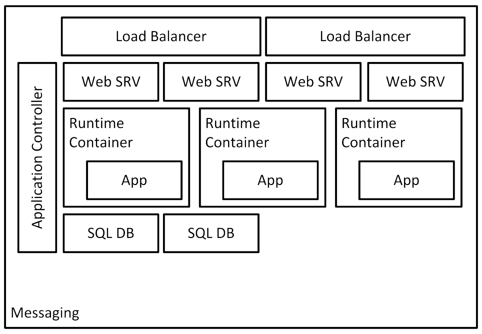


Figure 4.2: Software stack of a conventional PaaS framework.

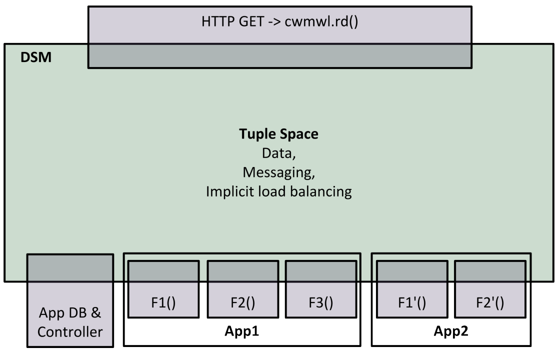


Figure 4.3: Unified Cwmwl PaaS fabric.

Cwmwl is developed to exploit distributed systems in cloud data centers by leveraging the strengths of coordination that we find in commodity cluster management tools (e.g. Clustrx [clu], parallel Gaussian [FTS$^+$]) and commercial space-based PaaS frameworks (e.g. Gigaspaces XAP [Gig]).

4.2 How the DSL CWMWL is implemented

The term ‘coordination language’ was coined in the year 1992 by Gelertner [GC92] to describe the LINDA programming language that he had proposed in the year 1985 [Gel85]. Since then coordination languages have first influenced distributed computing and HPC, and later Jini/Apache River and web services. The LINDA coordination uses a matching based mechanism. The LINDA model consists of a tuple space and a library that implements four primitives, `rd()`, `in()`, `out()` and `eval()`, as extensions to virtually any (non parallel) programming language.

- `rd()` retrieves a tuple that matches the given template.
- `in()` retrieves a tuple that matches the given template and permanently removes the retrieved tuple from the tuple space.
- `out()` stores a tuple into a tuple space.

Because remote execution not being in scope of our research (for more information on remote execution also read Section 4.7.1, we did not include the LINDA `eval()` primitive that spawns a new process in our investigation.

The LINDA primitives manipulate and store tuples, which are key-value pairs, in the tuple space acting as distributed shared memory (DSM). The sender publishes a tuple to the DSM and the receiver queries the DSM without the need to maintain knowledge from where to receive or what process to send to.

Cwmwl is a PaaS fabric that consists of:

- Messaging and serialization based on the UDP protocol.
- Secure and elastic workers.
- A means to identify and communicate with workers and processes.

Table 4.1: Efficacy of distributed computing architectures

Virtualization (either 1: many or many:1)	Hardware Platforms	Operating Systems Instantiated	Operating Systems Administered	Deployed application Instances	Example
Hardware Virtualization	1	3	3	3	IBM z/VM
Hypervisor (IaaS)	1	4 (including hypervisor OS)	3	3	KVM, VM-Ware
OS Virtualization	1	1	0 - 3*	3	Linux Containers ("Docker")
PaaS (current)	1	4	0	3	App-scale
PaaS (ideal**)	1	1	0	3	
Clustering	3	3	3	3	Hardware load-balancing

* `systemd` containers that are entered with ssh need to be administered like a VM, containers that are entered with nsenter do not require OS administration

**A PaaS that does not wrap VMs and operating systems that need to be administered and protected around atomic units of scale, such as processes or threads.

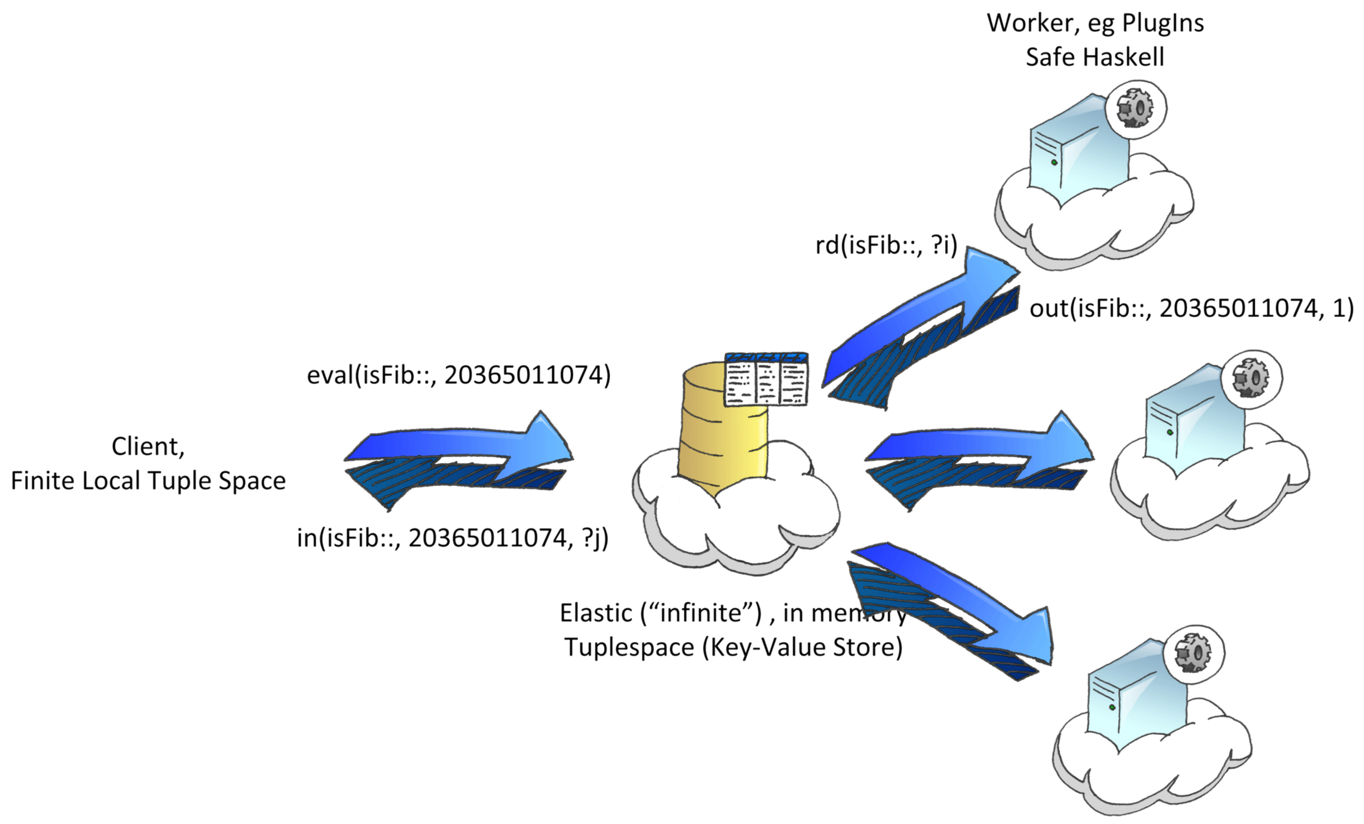

Worker, eg PlugIns
Safe Haskell
rd(isFib::, ?i)
out(isFib::, 20365011074, 1)
eval(isFib::, 20365011074)
Client,
Finite Local Tuple Space
in(isFib::, 20365011074, ?j)
Elastic ("infinite") , in memory
Tuplespace (Key-Value Store)

Figure 4.4: High level architecture of the Cwmwl PaaS fabric. Workers can consist of any unit of scale: e.g. plugins, functions..

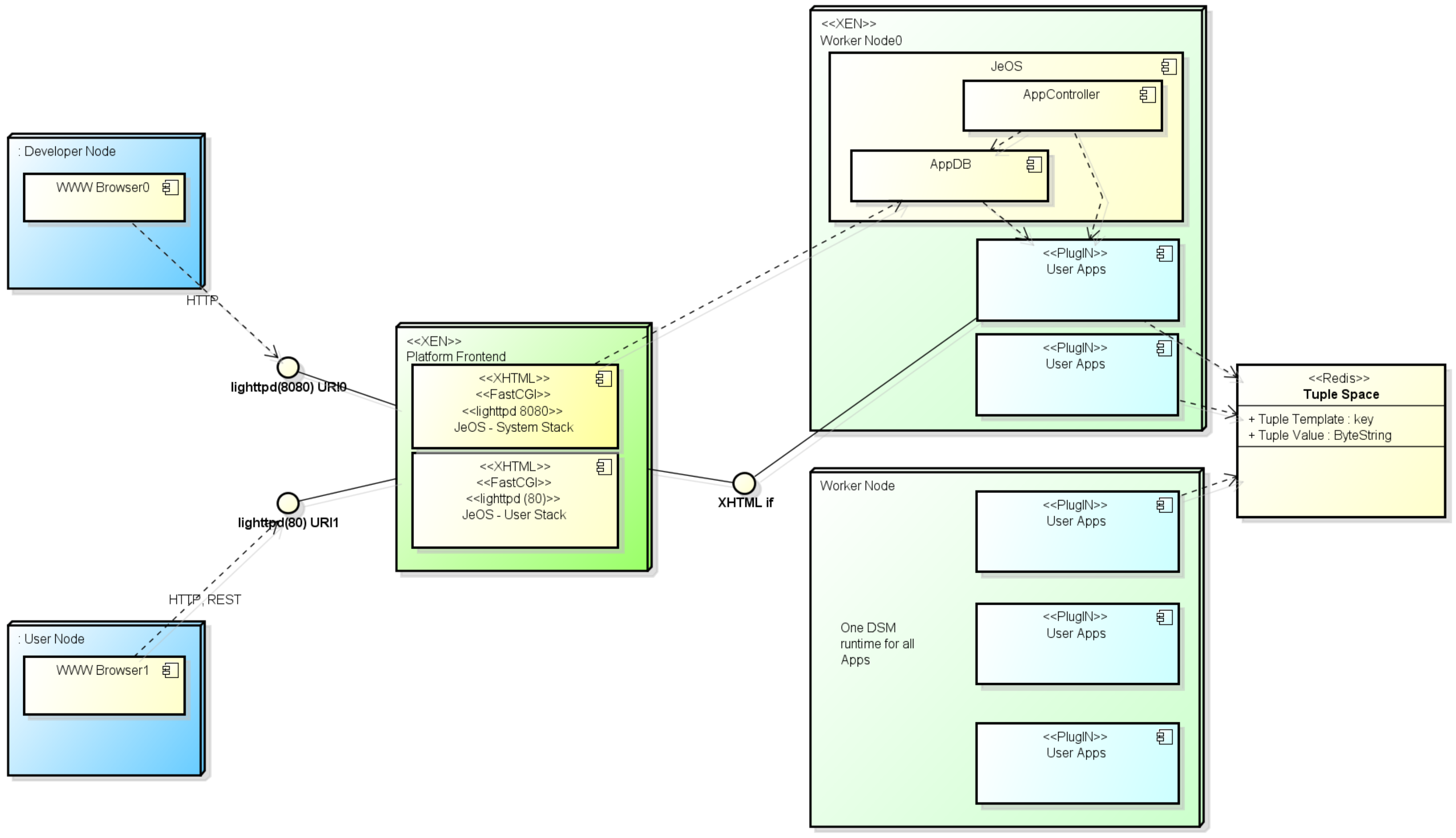


Figure 4.5: UML diagram of the Cwmwl PaaS fabric.

Figure 4.4 illustrates a high-level overview of the Cwmwl structure. Cwmwl consists of a central tuple space (TS) and several application servers (AS). For a distributed application that uses the master-worker skeletons developers will need to write a worker process, upload the code package to an application server and start it. To interact with the worker the developer uses the Cwmwl primitives to query the TS for tuples that have been published by a worker process. The TS serves as DSM, IPC and in-memory data grid, thus collapsing traditional multi-tier applications. To scale the distributed application more workers must be deployed [7]. This is merely a replication and does not require the implementation of an application load balancer that would represent additional overhead, to make use of the added computation power. The TS and the AS nodes are instances of guest virtual machines in commercial clouds. Cwmwl script is a means of coordinating and imposing constraints on a large pool of workers that, on large scale with hundreds or thousands of worker instances, will otherwise quickly become unmanageable. Whilst, for example, in commodity cluster management tools the coordinating middleware and the executed application are separate layers of software, Cwmwl script is part of the distributed application itself. Figure 4.5 depicts a UML deployment diagram of the Cwmwl fabric. The upper part of the UML diagram illustrates a developer accessing the platform to upload the application (code) to the application database (AppDB). An application controller (AppController) is in charge of deploying the user's applications from the AppDB to the available guest virtual machines. The lower part of the UML diagram illustrates a client accessing the deployed application through an intermediary web server (here lighttpd) using HTTP GET requests to the tuple store as a replacement for the LINDA `rd()` primitive.

4.2.1 Tuples and Templates

The syntax of the LINDA primitives that are used in Cwmwl has been kept similar to the original formulation of tuple spaces. Although the host language Haskell is strongly typed, Cwmwl does not use Haskell data types and tuple matching is purely syntactic. As proposed by Wells [Wel05], data-type conversions are handled by Cwmwl to create a truly heterogeneous system that is not limited to any (type of) programming language.

[7]Application (auto) scalers are currently a field of ongoing research (for example Calcavecchia et al [CCDN+12]) to which Cwmwl does not contribute at this time.

A limited number of types (that could easily be extended) are currently inferred by the Cwmwl parser. The abstract syntax tree (AST) of the Cwmwl interpreter is built for the types *string*, *integer* and some other types (*identifier*, *query*, *operation*) that implement the domain abstractions that are required to model the LINDA functionalities. Strong typing can be achieved by segmenting the tuple space where every segment is created to store only a specific type [vdGSW97]. A further option to preserve types is encapsulation, for example using `JSON` or `ByteString`.

Tuples stored in the key-value store (or in any tuple space) are accessed using a method that is called associative lookup, which matches tuples based on templates (also called anti-tuple). A template is similar to a tuple, except some of its fields may be replaced by a `NULL` [8]; for example `(isFib, )` value for wildcard matching. A template is said to match a tuple provided the following two conditions are met [Atk08]:

- The template is the same length as the tuple.
- Any values specified in the template match the tuple's values in the corresponding fields. There are two kinds of matches between a template value and a tuple value: exact match where the two values are exactly the same, or wildcard match where a `NULL` template value matches any tuple value.

For example, the template `(isFib, NULL)` will match the tuples `(isFib, 20365011074)` and `(isFib, 11021972)`, but not the tuple `(isFib, 20365011074, 'true')`. The example shows that a given template can match several tuples so the matching between template and tuple is not always unique. In Cwmwl, the interpreter uses the tuple template as a key to the Redis set data type, which is an unordered collection of strings. Since the interpreter cannot forecast if the key to a new key-value pair will always represent a unique mapping or if additional key-value pairs that reduce to the same key will follow, all new entries must start out as a Redis set with a single member. Tuples can be added to a set or removed from a set in O(1) constant time. Sorted sets are available and could, for example, be used to queue tuples

[8]The wildcard `NULL` can be omitted. To keep the key unambiguous the tuple delimiters must be retained.

that represent tasks or workers using a FIFO strategy. Identical work tasks need to be saved only once, and can be executed several times, either in parallel or sequentially.

4.2.2 Units of scale

A tuple space matches the master-worker scheme quite naturally. Campbell [Cap97] argues that the LINDA model is particularly natural for implementing some cases, such as task queues and recursive partitions. As for other skeletons he states that the segmentation of the tuple store (a technique that is frequently used in commercial tuple space implementations) and a degree of coordination would be beneficial to harness the evolving complexity. In the following we give two examples that are based on ACP:

1. Sequential (pipelined) execution processes.
2. Non-binding sequences.

Sequential (pipelined) execution of processes with *data flowing* between them, featuring systolic access patterns (see Figure 4.6), if implemented using a tuple space, would have each stage consuming the tuple from the previous stage and produces the tuple for the next stage. This is inefficient, considering that in order for tuples to pass between stages, the developer would have to index, track, and modify the tuples at each stage. An example of such tuple could be `out (myWorker, prev_state, next_state, data)`. The wildcard `NULL`. As a possible solution, Campbell proposes to segment the tuple space and let the segments represent queues similar to pub/sub channels. For example, a sequential process may consist of the stages (abcd) where stage (a) produces a tuple in its result tuple space (segment) for stage (b). Stage (b) reads the next tuple from there and writes its output to its own result tuple space. The resulting tuple space segments are comparable to pub/sub channels and reduce the amount of state and control information that needs to be passed around with every tuple, thus increasing the overall efficiency.

```
import language.CWMWL

main :: IO()
```

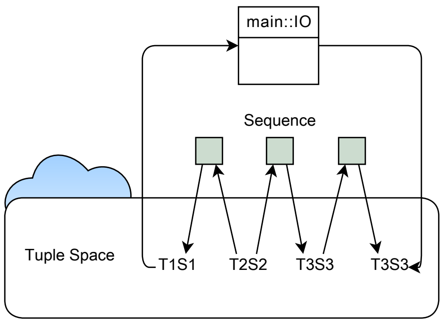


Figure 4.6: Tuple space used with an algorithmic skeleton that requires sequential execution (pipelining) and results in a systolic access pattern to the tuple space..

```

runCWMWL[
while (data) {
   in (myWorker, NULL) ;
  --reads a tuple that belongs to a
  --sequential computation

   out (myWorker, data) ;
  --writes back a tuple and updates
  --the state of the computation
}
]
```

Listing 4.1: Pseudo Cwmwl implementation of a pipelined algorithm.

An application that is run in our DSL may involve multiple execution instances that must be executed within the constraints that are specified by the DSL operators and the axioms of the ACP. The subset of the ACP that is illustrated in Table 4.2 and Equation 4.2 is implemented in the DSL interpreter through storing the running state of each sequential instance in a tuple template rather than in the tuple itself. A key is used for tracking a particular state and has the form of `(myWorker, NULL):$id` where `$id` is an integer that represents the current state of the instance `myWorker`. It can be incremented and queried to make sure communication take place at the right state. Redis commands are used in manipulating, incrementing and tracking state keys.

A (non-binding) sequencing operator '`&`' is implemented for all computations where there is a choice as to what operand is evaluated first. The execution results of the instances involved are stored in the Redis set datatype. For commutative sequencing which task is to end when all instances have signalled a certain state, the Redis `scard` query is used to return the number of elements in the set containing the results of the computations. Each instance must succeed (i.e., store a result in the set) to let the parallel commutative composition succeed. The number of elements in the set determines whether this is the case or not. This can be used for a barrier operation by requiring (ready) signals from all the involved processes before the next processing stage. In addition to the two sequencing operators the DSL supports a choice operator '`|`' that succeeds if any of the operands succeeds.

```
import language.CWMWL

main :: IO()

    runCWMWL[
    in (myWorker, NULL, NULL) & 2
    --And parallelism coordinated with
    --a non-binding sequencing operator
    --valid if two tuples are present

    in (myWorker_a, NULL, NULL)
         & (myWorker_b, NULL, NULL)
    --And parallelism of two different
    --workers, valid if both
    --computations have finished
    ]
```

Listing 4.2: Pseudo Cwmwl implementation of a two non-binding sequences.

4.2.3 Map Reduce: a data-oriented example

Programs 4.3 and 4.4[9] give a data-oriented example of how the Cwmwl TS can be used to support distributed computations by providing DSM and a channel for interprocess communication. The programs implement a data intensive map-reduce algorithm for matrix multiplication [Nor11] that partitions Matrix A and Matrix B into sub-matrices, and then performs the multiplications in parallel. Deploying this small distributed application involves the creation of the Redis Tuple Space and corresponding computing nodes - a mapper node and multiple reducer nodes. A mapper node loads two sparse matrices from a csv file into the Cwmwl tuple space. The reducer nodes carry out the multiplication. Currently, AWS EC2 (and many other large commercial clouds) does not support IP multicast, thus the tuple space must be registered to the computing nodes by means of a configuration file.

Depending on the available network bandwidth and speed, the performance of the mapper may benefit from the fact that loading data into key-value stores is much faster than casting data into a relational database [PPR+09]. Since the Cwmwl TS is based on a key-value store, the tuple (template) must contain all information required to address the matrices and their cells. In this example, we work with tuples of the format (a:i:k, value), in which a, i, k denote the matrix, row and column, respectively.

The reducer function multiplies, depending on the configuration of the indices, one or more sub-matrices and sums the results. It would be possible to introduce an intermediary mapper by, for example, splitting the multiplication from the addition and have the reducer sum up the results only.

There are two problems involved in extending Haskell with the Cwmwl tuple space. Firstly, access to a tuple space involves network communication, and any communication over a network is placed in Haskell in the IO Monad, which makes all its subsequent computation (such as the multiplications in our example) ‘impure’. This is a problem unique to Haskell, while in some other functional languages, such as Erlang, communications are untyped. Secondly, Haskell uses a static type system, which requires the two processes that use a tuple space as DSM to use the same data type. Cwmwl can work around this problem using JSON Frames (see also

[9]The reducer function has been contributed by Aaron Stevens

Section 4.2.1 of this paper) or Scoped Type Variables [JS04].

The map reduce example shows that Cwmwl can easily connect multiple processes that reside on different physical nodes and provides a means for distributed programs to access data in a way similar to accessing data locally. Using the Cwmwl TS, interprocess communication, and DSM is abstracted away from the developer. Accessing data, memory or inter-process communication are three distinct tasks in most state-of-the-art distributed applications that each require distinct efforts to implement, but in Cwmwl, they are merged into one simple task - accessing tuples in Cwmwl TS.

```
import qualified language.CWMWL as cwmwl

mapper = do

    do csv <- getContents
        case parse csvFile "(stdin)" csv of
              Left e -> do putStrLn "Error in csv file
                 import:"
                          print e
              Right r -> do
                          (key, value) <- castTpl r
                          --e.g. (a:i:k, value) ... (b:k:j
                             , value)
                          cwmwl.out (key, value)
```

Listing 4.3: The mapper code that casts matrices A and B from a csv file into Tuple Space..

```
reducer = do
    let indices = [(i, j, k) | i <- [0..rowsA],
                               j <- [0..colsB],
                               k <- [0..colsA]]

    -- Since `cwmwl . in` returns a value in the IO monad,
    -- we can't just multiply the values returned.
    r1 <- mapM (\(i, j, k) -> (cwmwl . in) (a, i, k)) indices
    r2 <- mapM (\(i, j, k) -> (cwmwl . in) (b, k, j)) indices

    -- We can multiply r1 and r2 together though,
    -- since they are values extracted from the IO monad
    return $ sum $ zipWith (*) r1 r2
```

Listing 4.4: The reducer code that executes matrix multiplication.

4.3 Testbed specification

The main goal of the experimental design was to approximate the environment as seen by real applications and to assess the impact of bandwidth and latency on the measured mean performance.

All of the experiments were conducted in the Amazon EC2 cloud. The central tuple store used a *Cluster Compute* instance so that throughput would not be limited by the available network bandwidth, application memory or computing capacity. Since Redis is single threaded and is best deployed on bare metal hardware without hypervisor [SN], an AWS EC2 *Cluster Compute* instance is the best available match for these requirements. AWS EC2 *Cluster Compute* instances are based on Hardware Virtual Machines (HVM) where the guest VM runs as if it were on a native hardware platform [Incb].

AWS EC2 *Cluster Compute* instances have 60.5GB RAM, 8 physical cores [10] and 10-Gigabit Ethernet connectivity. The clients $C_1 ... C_{20}$ were on separate instances of the type *M1 Large*. *M1 Large* instances have 7.5GB RAM, 2 virtual cores (4 EC2 compute units) and high I/O performance with unspecified network speed.

[10]88 EC2 compute units that equals 11 EC2 compute units per core as opposed to the usual 2.5 EC2 compute units per virtual core

Before the benchmarks were executed, the systems were modified to reuse and recycle TCP connections. Additionally the default range for TCP source ports was changed to the maximum port range: 1024 - 65535 (see Program 4.5).

```
sudo bash -c 'echo 1 > \
 /proc/sys/net/ipv4/tcp_tw_reuse'

sudo bash -c 'echo 1 > \
 /proc/sys/net/ipv4/tcp_tw_recycle'

sudo bash -c 'echo 1024 65535 > \
 /proc/sys/net/ipv4/ip_local_port_range'
```

Listing 4.5: Linux configuration file \etc\rc.local to modify all systems at boot time.

4.4 Benchmarking methodology

Two scenarios have been tested in our benchmarks: one with a simple tuple structure with changing payload, and the other with a large set of different tuple structures.

The first scenario involves a simple data tuple `('someData', PAYLOAD)` with payload sizes ranging from three bytes to 12KB. Regardless of the payload size, all tuples map to the same template `('someData', NULL)`, which is used as the key in the Redis key-value store and consequently are stored in the same Redis set. Such a usage pattern eventually reduces to the Redis `SADD` command that works in O(N) time where N is the number of tuples to be added to the set [SN]. However, in our test, only one tuple is added at a time, and thus the time complexity for each addition is O(1).

In the second scenario, a series of tuples with different structures of the format `('someData00001', PAYLOAD)` are involved. `'somedata00001'` ranges from `'somedata00001'` to `'somedata10000'`, for the reason that the tuple space must easily

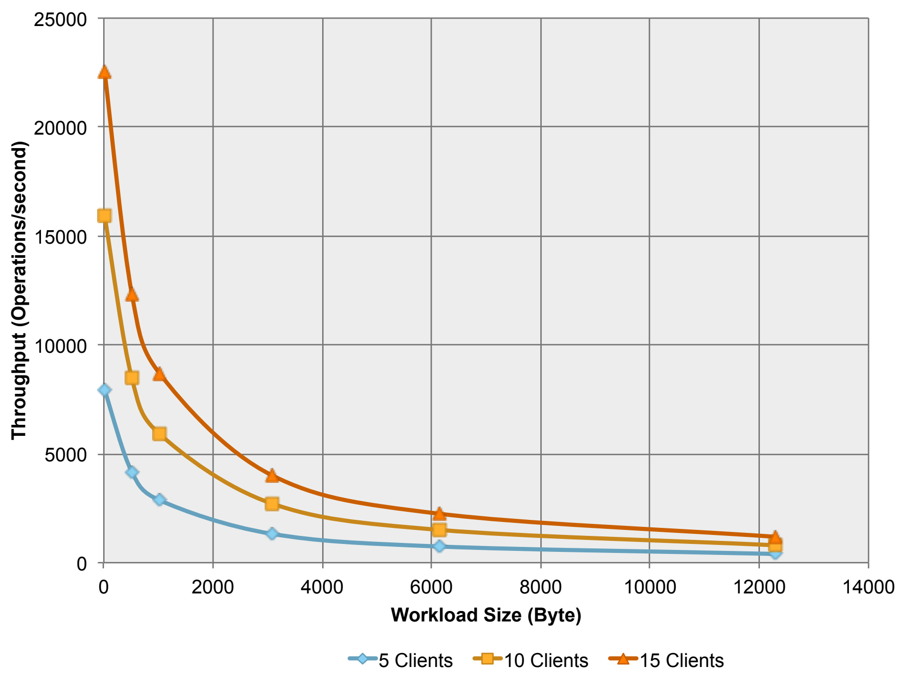


Figure 4.7: Impact of workload size on throughput for 5, 10, and 15 clients

fit into the memory of an EC2 Cluster Compute instance, even with the largest tuple payload. This usage pattern eventually reduces to the creation of a new Redis set with a single member.

To further simulate the environment as seen by real applications, prior to testing in each of the scenarios, the Cwmwl tuple space was aged with one million tuples with an automatically generated payload of the same payload size.

4.5 Tuple space performance

The initial state of the tuple space, which was simulated by tuple space ageing, had no impact on the performance results. The results confirm that both operations, inserting a new key-value pair and adding an additional value to an existing set, work in O(1) time. Figure 4.7 shows a nearly logarithmic relation of the tuple workload size (measured in bytes) on the tuple space throughput (measured in operations per second) for 5, 10 and 15 concurrent clients. The tuple

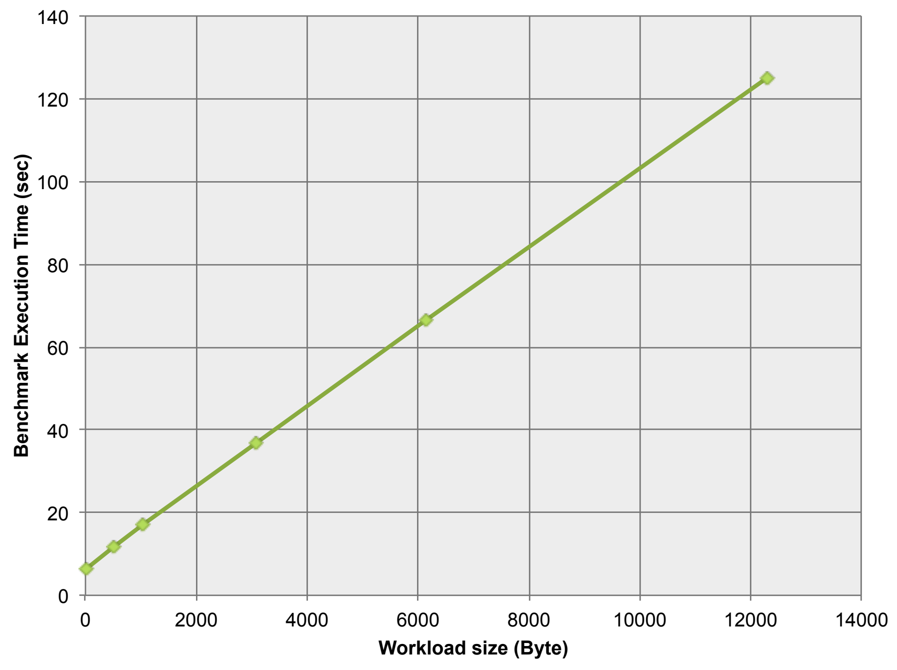


Figure 4.8: Benchmark execution time for up to 15 Clients

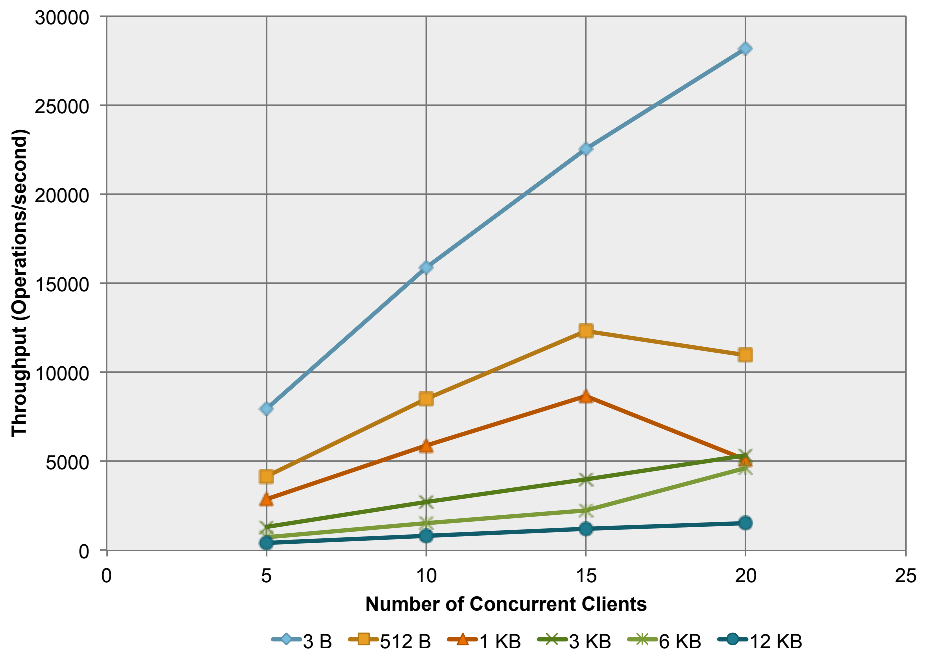


Figure 4.9: Impact of the number of clients on throughput for 3 B to 12 KB

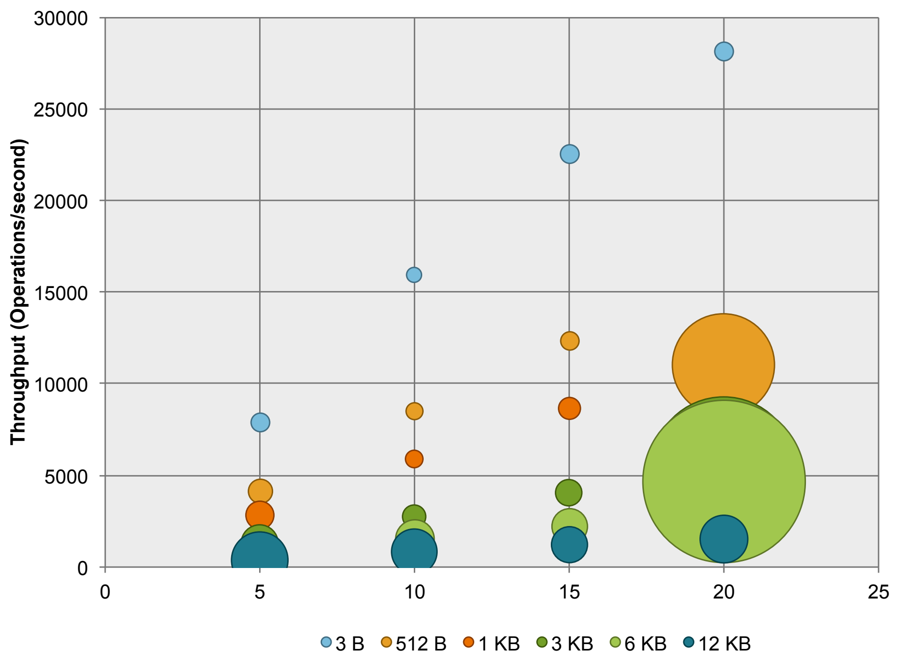


Figure 4.10: Impact of the number of clients on throughput for 3 B to 12 KB. The bubble size represents the standard deviation from the mean value..

space throughput drops logarithmically with the workload size. The standard deviation of the measurements for the benchmark with 20 concurrent clients were very high (see also Figure 4.10) and this benchmark is thus not included in this diagram.

Figure 4.8 shows that the gross benchmark execution time for up to 15 clients is (within the standard deviation) constant and proportional to the workload size of the tuple. Figure 4.9 shows that within the boundaries of our experimental design the tuple space throughput keeps increasing with the number of concurrent clients executing the same benchmark. Obviously, the performance limits of our tuple space implementation could not be reached within our experimental design. On the other hand, the large standard deviations (Figure 4.10) of the benchmarks with 20 concurrent clients and the 'knee' at 15 concurrent clients in Figure 4.9 may imply that there is a performance boundary between 15 and 20 concurrent clients. Interestingly this is in line with Fiedler [FWR$^+$05] who also finds a 'knee' around 15 concurrent clients executing the same benchmark on JavaSpaces.

The Cwmwl `rd()` primitive produced a constant benchmark execution time of around 6.5s that was not influenced by the size of the payload or by the number of concurrent clients. However, we had the impression that the `rd()` was altogether less scalable and locked the system network queues significantly longer than the `out()` primitive that in turn leads to undesirable exhaustions of the connection pools.

The overall results were very consistent and predictable giving a good basis to understand the impact of the Cwmwl tuple space on application performance and scalability. The repeatability of the benchmarks for up to 15 concurrent clients confirms the validity of our findings within the expressed range. Spot checks show that the absolute values (operations / second) are approximately half the performance of the `redis-benchmark` tool that is included in a Redis installation. The difference could be caused by differences in serialization, sources and exploitation of randomness to generate the tuple payloads or the Redis bindings for Haskell. The gross throughput with 15 clients was around 1.8 Gbps, slightly higher than the nominal throughput of SATA 1.0 (1.5Gbps).

It is natural to question the performance advantages of the Cwmwl Paas fabric over the current PaaS frameworks. However, most of the current PaaS frameworks are based on a complex software stack and computation speed or IO alone cannot measure their performance. According to Zhang et. al. [ZHC$^+$12], the performance of current PaaS frameworks may be assessed tier by tier. In practice, PaaS performance is often discussed qualitatively in terms of the time and effort required to deploy a new (web) application or to do a major application upgrade. Also, it is often discussed to what extent, and with what effort, it is possible to elastically 'right-size' (scale up and down) a deployed application. Frequently this is supported by additional middleware that must be subscribed (e.g. Rightscale [Cla10]) and not by the cloud computing platform itself. Map Reduce is frequently sold outside PaaS frameworks as a separate capability that must be configured using workflows and storage, showing again the lack of harmonization of computation, data, applications and web applications in computing clouds.

Cwmwl is intended to rethink PaaS design and to merge brute replication and re-clustering (which is the current methodology to implement PaaS) with distributed computing to improve efficiency and cost. Our foremost design goal is simplicity by achieving a novel unified platform rather than virtualizing and replicating the implementation of a load balanced web application that has existed since the end of the 1990s. The performance Figures of the Cwmwl TS that is accessed with the Cwmwl primitives support our claim that this can be done. After all, achieving a performance close to SATA 1.0 is a good start.

4.6 Evaluation criteria

Our evaluation criteria are based on Papadopoulos [PA98] who discriminates coordination languages according to seven main characteristics:

1. Entities being coordinated (COO1).
2. Mechanisms of coordination (COO2,–read ASM4).
3. Medium of coordination (COO3,–not scored).
4. Semantics / rules protocols (COO4).
5. Degree of Decoupling (COO5,–read ASM3).
6. Range of programming languages supported (not in scope of this thesis).
7. Application domain (COO6).

COO1	**The PL provides entities suitable for coordination**
References	[GC92] , [PA98]
Rationale	Elasticity and scalability beyond the platform where the application is started requires to decompose applications into units of scale the can be distributed and scaled independently.
Specifics	Coordination languages require easy access to entities suitable for coordination, such as functional skeletons, nodes, processes, sequential tasks or web services.

COO1	**The PL provides entities suitable for coordination**
Ratings	1. Applications in the FPL are comprised of **portable**, **inter operable** and **self-contained** computational entities. 2. Applications in the FPL are comprised of entities that miss one or more characteristics or they are not easily accessible. 3. The FPL does not provide entities suitable for coordination.

COO2	**The PL provides a mechanism for coordination**
References	[PA98] , [CJLL02]
Rationale	The (host) language must provide a component interaction mechanism that can be used for coordination.
Specifics	[PA98] recognizes the following mechanisms: • Events and channels (for example message passing). • Function application and composition (as used in functional skeletons). • (Remote) Method invocation. • Tuple exchange. • Specialized mechanisms, such as chemical reaction.

COO2	**The PL provides a mechanism for coordination**
Ratings	1. The programming language provides an interaction mechanism for distributed units of scale (or components). 2. The programming language provides an interaction mechanism for units of scale on the local system that is easy accessible and can be extended for distributed environments. 3. The programming language must be extended to implement the required mechanism.

COO3	**Medium of Coordination can be provisioned and accessed (Not scored)**
References	[RC90]
Rationale	Almost all coordination models use a Shared Dataspace. Roman [RC90] as quoted in Papadopoulos [PA98] explains: “A Shared Dataspace is a common, content-addressable data structure. All processes involved in some computation can communicate among themselves only indirectly via this medium. In particular, they can post or broadcast information into the medium and also they can retrieve information from the medium either by actually removing this information out of the shared medium or merely taking a copy of it.”
Specifics	Performance and scalability are crucial for the usefulness of a Shared Dataspace. Distributed shared memory [GHW12a], [GHW12b] has been successfully implemented and commercialized [11] demonstrating the feasibility.
Ratings	

[11]For example by the company RNA Networks, which was acquired by Dell Inc. in the year 2011.

COO4	**Rigorous semantics, rules or protocols can be implemented**
References	[Gel85] [CGW91]
Rationale	The semantics of the subset / coordination language need to be sufficiently to allow logic reasoning about coordination.
Specifics	Formal reasoning supports static code analysis and benefits the development of compilers. Well-defined DSLs can be better understood and support the development of quality software.
Ratings	1. The General Programming Language (GPL) supports the implementation of a DSL / Coordination language independently from a third party platform or runtime (e.g. , Clojure and Scala are built on top of JVM, F# on top of the .NET platform). 2. The General Programming Language (GPL) supports the implementation of a DSL / Coordination language but has further dependencies on third party platforms. 3. The GPL is inherently limited and cannot serve as base language for a DSL.

COO5	**Degree of decoupling**
References	
Rationale	Direct coupling between client and server, as is the case with remote invocations, should be avoided. This property is assessed in Section 3.5.3 of this thesis.
Specifics	**Indirect communication** paradigms seek a level of uncoupling. For example, uncoupling of entities in space and time where the coordination medium and the medium's contents are independent of the life history of the processes involved.

COO5	**Degree of decoupling**
Ratings	1. The FPL provides rich and precise means to create shared nothing architectures, for example by closely integrating with a concurrency model that applies to local and remote processes. 2. Node-level shared nothing architectures can be implemented using third party libraries but they are not cost effective and not sufficient. 3. There is neither explicit nor implicit support of node-level shared nothing architectures.

COO6	**Relevance and applicability to the domain computing clouds**
References	[KDH11]
Rationale	The obtained coordination language needs to have special merit (be ‘relevant’) to the development of applications in computing clouds.
Specifics	
Ratings	1. The programming language is relevant for the application domain computing clouds. 2. The programming language needs to be extended in order to be relevant for the application domain computing clouds. 3. The programming language is not relevant for use in computing clouds.

4.7 Assessment of functions as lightweight units of scale for distributed applications in computing clouds

Now, Haskell is assessed against the criteria developed in the previous section.

4.7.1 COO1: Entities suitable for coordination

Haskell is a pure functional programming language. Application source code written in Haskell is composed of functions (frequently also referred to as ‘expressions’). Functions as units of scale of scalable applications for computing clouds have the following strengths and weaknesses:

Strengths

- Functions always evaluate to the same value for a given argument (”free of side effects”).
- Variables are assigned once.
- The order of function execution can be re-arranged. For example, Functions can be composed, curried, etc..
- All pure functions can be executed in parallel.
- Function evaluation can be cached.

The strengths support the assumption that functions can be decoupled in space and time, providing strong support for coordination models where deterministic entities (here functions) are composed into a deterministic outcome using explicitly non-deterministic mechanisms.

Weaknesses

- Functions need to be part of a program. A `main` function is required as entry point into a Haskell program. The use of Haskell Plugins [PSSC04] can abstract the need for a `main` function away from the developer. A (cloud) service provider could for example maintain a platform where client code is dynamically loaded as plug-ins.

- Haskell extensions for parallel and concurrent programming are designed to exploit multicore CPUs on the local system. At the time of writing this thesis Haskell does not support remote execution / remote method invocation such as Java RMI. According to Marlow [Mar12] there are efforts (such as 'Cloud-Haskell', now called 'remote') under way to eventually support remote execution. However, Marlow perceives non-deterministic means as "a little unfortunate", while the authors of this thesis argue that non-deterministic means must be introduced where needed to fit the nature and requirements of computing clouds–without disrupting the essential determinism of programming languages.

It is a wide spread misconception that distributed software for computing clouds requires mobility of code or objects, such as available in Java RMI or partially in the Haskell small`remote` package. The authors of this thesis believe that there are not many use cases that mandate code mobility. Scalable applications for computing clouds require self-contained workloads that can use indirect communication and coordination to 'glue' together the eventual application.

Self-contained workloads can be (partially) dynamically linked Haskell binaries or for example containers that consist of the Haskell binary and the required libraries. Popular options to deploy containers are Linux Containers managed by Docker or by Apache Mesos [HKZ+11].

Docker containers fit well into DevOps driven environments where constant integration and constant deployment paradigms are sought. Docker containers can share DLLs with the underlying operating system somewhat reducing the size of the required Haskell binary. Resource management requires additional frameworks, such as Kubernetes or Apache Mesos.

Apache Mesos fit better to environments with very large scale where the emphasis is on the separation and management of workloads and resources. But there is one more issue. While networking plays a key role in data center infrastructure, distributed applications and coordination models that decouple in space and time, networking is, for now, beyond the scope of

Docker or Mesos.

$$[[[function + entrypoint]^{\text{Haskell binary}} + DLLs]^{\text{Container}} + ResourceManager] + Networking \tag{4.1}$$

Equation 4.1 illustrates the components required to build a PaaS framework around Haskell functions. In summary, a framework that is using Haskell functions as units of scale would consist of:

- Haskell Binary.
- Required DLLs, for example packaged together with the binary into a Linux container or shared between multiple Linux containers.
- Data Center Operating System to manage resources and deploy workloads, such as Apache Mesos is out of scope of this Thesis. The authors of this thesis were using the Ansible [RH16] configuration management tool to distribute the workloads in a semi-automated way. But this does not provide resource management.
- Networking is in this research achieved through bindings to a coordination language in the `main` function.

Item	**Haskell**	**Java**
Archive	Hasekll Binary	.jar file
Supporting Code	DLLs or PlugIn loader	JVM[12]
Resource Manager	Required	Required
Networking	Required	Required

Table 4.1: Comparison of Haskell Functions and .jar archives when used as units of scale

[12]Generally one JVM needs to be deployed per container, since multiple applications per JVM represent operational challenges that must again be resolved with appropriate frameworks, such as OSGi

Table 4.1 compares the effort and adjunct features required to use Haskell functions and Java .jar archives as portable units of scale considering the limitations explained above. On the one hand we see that both programming languages are sufficiently similar in their requirements and cannot easily deduct a special merit of FPL Haskell. On the other hand, based on the results of our practical evaluations, the authors of this thesis believe that the strengths listed earlier in this chapter cannot easily reproduced by for example Java processes or objects and that large scale function executing services, such as AWS Lambda [Inca], can be achieved using Haskell Plugins. The size of this infrastructure would however still be dictated by the number of clients and their running functions rather than by the actual load.

FPLs are also a good basis to build unikernels. Besides containers and data center operating systems, such as Apache Mesos, unikernels have started to gain some traction because they allow building scalable systems in a very efficient way. Unikernels such as Mirage [MMS+10], Erlang on Xen (Ling) [SK12] and the Haskell Lightweight Virtual Machine (HaLVM) [hal15] show that it is possible to convert high-level functional language source code into a kernel (also called exo kernel) that runs directly on the XEN hypervisor that is used in most commercial public clouds such as Amazon EC2. The instantiation time of a system based on unikernels is in the ms range. Hence, it is possible[13] to build PaaS frameworks that reduce the required footprint to the absolute minimum (see also Table 4.1). Furthermore, all three aforementioned unikernels are single threaded that makes them good candidates for coordination models, and a stable coordination layer based on a tuple space could be an ideal basis to glue both OS-less applications and OS-equipped legacy VMs to a hybrid elastic cloud platform.

Rating: 2. Applications in the FPL are comprised of entities that miss one or more characteristics or they are not easily accessible.

4.7.2 COO2: Mechanism of coordination

Papadopoulus [PA98] lists five mechanisms of coordination (for a complete list of the five mechanisms read Subsection COO2 in Section 4.6). Haskell can use the following two mechanisms of coordination that are generally limited to the local system:

[13]Unikernels have not been evaluated in our cloud computing test bed.

- Events and channels.
- Function application and composition.

It must be noted that functional skeletons in Haskell have been investigated [Col04], frequently as they relate to Eden [Loo12], a Haskell extension for parallel functional programming. However, similar to events and channels, functional skeletons are generally focusing on the local system and do not support distributed environments.

Section 3.5.4 evaluates Haskell for support of message passing to transmit data, such as state information, and re-synchronize the workloads, for example, functions, processes or micro services in distributed environments and we align the rating of this requirement with Section 3.5.4.

Rating: 3 The programming language must be extended to implement the required mechanism.

4.7.3 COO3: Medium of coordination (Not rated)

It would not be appropriate to expect that a data store, such as a tuple space, must be a first class citizen of a programming language. Although before the advent of relational databases, there have been programming languages and operating systems that merged programming language and data stores, for example, the Clipper Compiler [Spe91] dBase [Wik15b] , FoxPro [www15], the IBM Application System 400 (AS/400) or PickBasic [Sis87], such close couplings for programming languages have lost popularity a long time ago. To date it is the databases that encapsulate the programming language such as SQL or PL/SQL.

Although it would have been possible to co-locate the mechanism of coordination with the programming language Haskell by using for example a Binary Search Tree (BST) or making better use of the Haskell Type System for the creation and matching of tuples; for the proof of concept of this thesis we have selected to implement the mechanism of coordination based on an existing data store [14] that has

1. Relevance and popularity in the application domain of computing clouds.

2. Advantages for the implementation of a medium of coordination.

Relevance to computing clouds. Key-value stores, noSQL and Big Data, all of which are strongly linked to cloud computing, have increasingly gained pace in recent years. Cwmwl suggests a promising cloud infrastructure through combining these new paradigms with a high performance tuple space, living outside the web services world, and obtaining its applicability by simply using tuple spaces and LINDA-based coordination languages. In the Cwmwl infrastructure a key-value store is used to serve application needs, store Big Data, and as a tuple space, which, if employed wisely, can greatly reduce the software stack, complexity and footprint of applications in the cloud. The reduction of the software stack, complexity, and footprint makes Cwmwl a flat PaaS fabric that differs from the more common hierarchical or tier-based PaaS paradigms.

Regarding the implementation of key-value stores, two seemingly opposing trends have been observed: *in-memory data grids* and *distributed key-value stores*. In-memory data grids, where all data is kept in memory, provide fast access to data with low latency and high performance. They are frequently used for near real time (BIG) Data analytics. The Lewis Carroll Diagram in 4.11 shows how popular cloud computing offerings (purple shaded areas) overlap with use cases from the application domains web services, clusters, distributed computing and HPC. Key-Value stores are relevant in the categories web services (for example, the implementation of failure tolerant shopping carts or session stores), distributed computing (for example, search) and HPC (for example, storage and analysis of log data).

The corresponding commercial offerings for this market segment are 'bare metal clouds' that allocate dedicated servers and offerings based on hardware-virtualization where one single in-

[14]Additional reasons for not integrating the data store with the programming languages are reduced lock-in and the use of well-defined APIs as de-facto standard that allows access to data in a common way, regardless of programming language, data store / database or schema (as is the case with NoSQL databases).

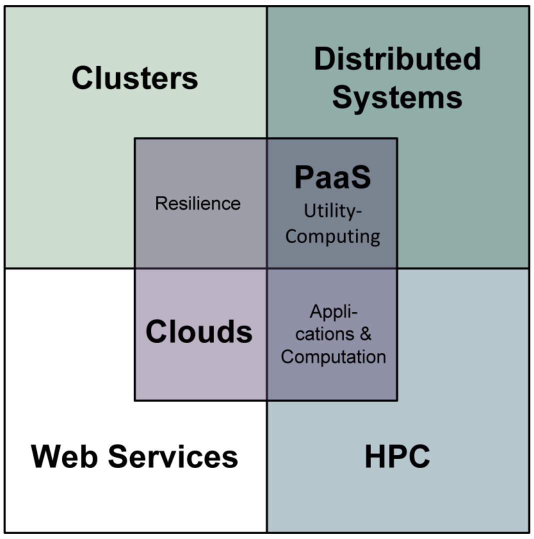


Figure 4.11: Lewis Carroll Diagram showing how cloud computing offerings (purple area) overlap with web services, clusters and HPC.

stance can have more than 200GB RAM [15]. Distributed key-value stores that may involve map reducing [16] and additional delays through vector clocks before a query is responded to. Dabek and Peng [PD10] introduce the Google development called Percolator, which is a combination of both types of key-value stores.

We believe that the next generation of cloud computing platforms will embrace upcoming peta- and exa-scale mainstream systems and shift in focus from a technology that mainly delivers web services (Figure 4.11 bottom left) to a more abstract service that leverages distributed computing to support non-web-service applications, or rather common business applications (Figure 4.11 top right).

Special merit for implementation of a coordination medium. Tuple spaces are a lightweight means of memory virtualization and consequently very similar to persistent storage memory with the added value that tuple spaces can easily scale across the physical boundaries of nodes. An in-memory key-value store was chosen as the basis for the Cwmwl tuple space [17]

[15]For example, the AWS HPC offerings or the Storm bare metal servers

[16]For example RIAK [ria] or the Amazon Dynamo [DHJ+07a]

[17]Transient tuple spaces appeared first in JavaSpaces. The focus however was on volatility of tuples at the time of system reset rather than creating an optimal in-memory data grid.

in order to make it suitable for the implementation of in-memory data grids as well as for the virtualization of (application) memory, data storage, and interprocess communication (IPC) across computer or network architectures. The in-memory key-value store Redis [SN] was chosen for the following reasons:

- It is very fast and lightweight, which makes it ideal for frequent random access with very low latency as required by tuple space implementations. The expected speed gain compared to a disk based key-value store is approximately 1:100000 [Los10].
- It supports persistent tuple spaces (if data and tuples are long lived) and preserves state across restarts if required.
- It supports atomic operations. Most key-value stores do not support transactions and use, for example, vector clock schemes [BR02] to detect and resolve conflicts.
- It supports a number of data structures (for example sets, lists, and ordered sets) that can be used to support functions such as tuple matching and advanced coordination (see Section 4.2.1 of this document).
- It supports key-value pair expiry times which are used to release expired tuples.
- It supports persistence. Regarded as a less important feature of an in-memory tuple store, in practice, it cannot be underestimated as a source for debugging information.

There are two problems involved in extending Haskell with the medium of coordination.

1. Access to a remote medium of coordination involves network communication, and any communication over a network is placed in Haskell in the IO Monad, which makes all its subsequent computation ‘impure’. This is a problem unique to Haskell, while in some other functional languages, such as Erlang, communications are untyped.
2. Haskell uses a static type system, which requires the two processes that use a tuple space as DSM to use the same data type. Cwmwl can work around this problem using JSON Frames (see also Section 4.2.1 of this paper) or Scoped Type Variables [JS04].

The above issues are consequences of the control the Haskell (developers) have over effects and can also bee seen as an advantage of the FPL Haskell over other programming languages that have not the same strict control over effects. Furthermore it is natural that the types of an EDSL need to be more limited then the types of the host language itself. For more information where the increased control over effects is helpful read also Subsection 4.7.4.

4.7.4 COO4: Rigorous semantics, rules or protocols can be implemented

We used Haskell as base programming language to implement a Domain Specific Language (DSL) that has the primitives of the LINDA coordination language and to define custom operators and priorities to achieve the required semantics.

We make three operators available that express sequences and parallelism (see Table 4.2 for a description of the DSL script operators) enabling the interpreter to automatically maintain the computation states and thus abstracting the need to manually maintain state and control information away from the developer. The operators of our DSL are aligned with the Algebra of Communication Processes (ACP) [BK86], among which there is a strong binding sequence operator '`;`' that is used to express the sequential execution of processes, a non-binding sequence operator'`&`' that is used for barrier execution of processes and an exclusive choice operator '`|`' that performs eureka execution. Furthermore our DSL adopts the axioms and the process algebra as defined by the ACP. Equation 4.2 reprints the ACP axioms using the Cwmwl script operators in the notation. ACP is defined by more axioms that are not relevant to our use case and thus not reprinted in this paper.

$$
\begin{aligned}
a\&b &= b\&a \\
(a\&b)\&c &= a\&(b\&c) \\
a\&a &= a \\
(a\&b);c &= a;c\&b;c \\
(a;b);c &= a;(b;c)
\end{aligned}
\tag{4.2}
$$

Table 4.2: DSL operators

Operator	Description
;	strongly binding sequence
&	non-binding sequence (commutative), and 'parallelism', barrier network
|	exclusive choice, succeeds if any of the operands does so, heureka network

We found that Haskell supports the creation of our DSL with:

- Type system.
- Syntax flexibility.

Type system. We used the Haskell Type system to create an Abstract Syntax Tree (AST)– illustrated in Listing 4.6 that limits and operations that are supported by our DSL. The abstract syntax tree (AST) of the our DSL is built for the types `string`, `integer`, and some other types (`identifier`, `query`, `operation`) that implement the domain abstractions that are required to model the LINDA functionalities. Strong typing can be achieved by segmenting the tuple space where every segment is created to store only a specific type [vdGSW97]. A further option to preserve types is encapsulation, for example using `JSON` or `ByteString`.

```haskell
{-# LANGUAGE DeriveDataTypeable #-}
{-# LANGUAGE DeriveGeneric #-}



module AST where

```

```haskell
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

import Data.Typeable

-- import Data.Serialize

data AST
    -- = Number Double
    = Number Integer
    | Identifier String
    | String String
    -- | Operation BinOp AST AST
    | Query String
    deriving (Show, Eq, Generic)

data Tuple = Tuple {cmd :: String,
                    cid :: AST,
                    argumentList :: [AST],
                    queryList :: [AST]} deriving (Show, Eq,
                       Generic)

data ACPop = Plus | Minus | Mul | Div
    deriving (Show, Eq, Enum, Typeable)

type CWLANG = AST

instance Serialize AST
instance Serialize Tuple
```

Listing 4.6: Abstract Syntax Tree

Although the host language Haskell is strongly typed, we do not use Haskell data types and tuple matching is purely syntactic. As proposed by Wells [Wel05], data-type conversions are handled by our DSL to create a truly heterogeneous system that is not limited to any (type of) programming language. [18]

Since the coordination of workloads in distributed environments, such as computing clouds, requires network IO, our computations are not pure and the types used in our DSL need to be in the IO Monad (such as the matrix multiplication in Listings 4.3 and 4.4) . In the IO monad, side effects are allowed, and we needed to give up control over effects that is frequently seen as one of the key advantages of Haskell over other programming languages.

Syntax flexibility Haskell allows DSLs to use nearly arbitrary syntax via Quasi Quotes [Mai07] and via the Haskell parsing library Parsec [LM01]–we give our DSL the look & feel / syntax based on the language specification published by Gelertner [Gel85] (read also Subsection 4.7.2 for an overview of the required syntax). Listing 4.7 shows the parsers that we developed for the implementation of LINDA primitives. Together with the AST and lexer that reserves the new primitives (listing 4.8 shows the lexer without the required biolerplate code) the parsers can be used as a compiler frontend to tokenize the new primitives (listing 4.9).

Rating: 1 The General Programming Language (GPL) supports the implementation of a DSL / Coordination language independently from a third party platform or runtime (e.g. , Clojure and Scala are built on top of JVM, F# on top of the .NET platform).

```
module Parser where

import Control.Monad (liftM)
import Control.Applicative hiding (many, (<|>))
import Text.Parsec
import Text.Parsec.String (Parser)
import Lexer
```

[18]Andres Loeh contributed an example of a typed LINDA where a list is used as medium of coordination. The example is reprinted in the appendix of this thesis.

```haskell
import AST

acomma = lexeme comma

aparens = lexeme (symbol ")")

baldString = lexeme . fmap String $
    (:) <$> noneOf "? ,)"
        <*> many (noneOf " ,)")    -- problematic - see comment below
        <* acomma

number = lexeme . fmap Number $
     read <$> many1 digit
          <* acomma

niceTuple = Tuple <$> lexeme resvd_cmd <* lexeme (char '(')
                  <*> lexeme ident <* acomma
                  <*> many ( number <|> baldString )
                  <*> many queries

ident = liftM Identifier identifier <?> "WorkerName"

queries = lexeme . fmap Query $
          (:) <$> oneOf "?"
              -- <*> many letter
              <*> many (noneOf " ,)")
              <*  (  aparens <|> acomma )

resvd_cmd = do { reserved "rd"; return ("rd") }
            <|> do { reserved "eval"; return ("eval") }
            <|> do { reserved "read"; return ("read") }
            <|> do { reserved "in"; return ("in") }
            <|> do { reserved "out"; return ("out") }
            <?> "LINDA-like Tuple"
```

Listing 4.7: Tuple Parser

```
module Lexer (
            identifier, reserved, operator, reservedOp,
               charLiteral, stringLiteral,
            natural, integer, float, naturalOrFloat, decimal,
               hexadecimal, octal,
            symbol, lexeme, whiteSpace, parens, braces, angles
               , brackets, semi,
            comma, colon, dot, semiSep, semiSep1, commaSep,
               commaSep1
    )where

import Text.Parsec
import qualified Text.Parsec.Token as P
import Text.Parsec.Language (haskellStyle)

lexer = P.makeTokenParser ( haskellStyle
                             {P.reservedNames = ["rd", "in", "
                                out", "eval", "take"]}
                          )
```

Listing 4.8: Lexer without boilerplate

```
ubuntu@ip-10-244-148-6:~$ ghci Parser.hs
GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 3] Compiling AST              ( AST.hs, interpreted )
[2 of 3] Compiling Lexer            ( Lexer.hs, interpreted )
[3 of 3] Compiling Parser           ( Parser.hs, interpreted )
Ok, modules loaded: Lexer, Parser, AST.
*Parser> parseTest niceTuple "rd (isFib, test2, 100, 200, ?
   STRING)"
Loading package bytestring-0.9.2.1 ... linking ... done.
Loading package transformers-0.2.2.0 ... linking ... done.
Loading package mtl-2.0.1.0 ... linking ... done.
Loading package array-0.4.0.0 ... linking ... done.
Loading package deepseq-1.3.0.0 ... linking ... done.
Loading package text-0.11.2.3 ... linking ... done.
Loading package parsec-3.1.3 ... linking ... done.
Tuple {cmd = "rd", id = Identifier "isFib", argumentList = [
   String "test2",Number 100,Number 200], queryList = [Query "
   ?STRING"]}
*Parser> parseTest niceTuple "rd (isFib, test2, 100, 200, ?
   STRING, ?BOOL)"
Tuple {cmd = "rd", id = Identifier "isFib", argumentList = [
   String "test2",Number 100,Number 200], queryList = [Query "
   ?STRING",Query "?BOOL"]}
```

Listing 4.9: Command line demonstration of the tokenizer

4.7.5 COO5: Degree of decoupling

For an evaluation of this requirement read Section 3.5.3 of this thesis.

4.7.6 COO6: Relevance and applicability to the domain computing clouds

Computing clouds are workload agnostic. Programming language must either support a wide variety of application domains–making them essentially general purpose programming languages or they can support specific domains but must not hamper the key features and paradigms of computing clouds.

Kachele [KDH11] has published eleven requirements of typical business applications that should be supported by cloud computing platforms but found that “none of the current platforms support a majority of these requested features”. The requirements of Kachele’s study are compatible to a large extent with our objective in this thesis. As the requirements are still of significant importance these days, and the chosen cloud computing model is PaaS, we consider it as an appropriate starting point for the assessment of Haskell together and the LINDA-based coordination language that we obtained.

Table 4.3 lists and compares how Haskell and our LINDA-based coordination model match Kachele’s eleven requirements for business applications in computing clouds.

Rating: 2 The programming language needs to be extended in order to be relevant for the application domain computing clouds.[19]

[19]This finding is in line with the findings of Epstein [EBPJ11] and [Mar11] (p. 244ff.). While both aim to extend Haskell capabilities that have been designed for parallel and concurrent computing on the local system to make them fit for distributed environments such as computing clouds, we buy into Lee’s argumentation that we examine at the start of this chapter.

Table 4.3: Eleven requirements of business applications [KDH11]

Application Requirement	Explanation	Haskell	Haskell based DSL
Application-centric approach	Developers only need to focus on core application development and functional aspects	Well-standardized GPL. Large number of libraries available	Simple, high level communications model.
Application-independent approach	Applications executed on PaaS must not be limited to web services	GPL appropriate for many use cases.	Separation of concerns achieved independent of how computation is performed. Associative memory and generative communication are inter-operable.
Elasticity	Platform should be elastic and ideally preserve application state during scaling	No support for elasticity in distributed environments.	Application state is preserved in the central tuple store, elasticity depending on unit of scale. Tasks and Workers run decoupled, in parallel and at virtually any scale.
Virtual addressing	Location is irrelevant, customers must be able to reach their application from everywhere	Virtual addressing not supported. Developers need to maintain for example ’call graphs’.	Associative addressing specifies what data, message or worker is requested rather than an address.

Table 4.3: Eleven requirements of business applications [KDH11]

Application Requirement	Explanation	Haskell	Haskell based DSL
Cloud-independent programming-model	Software should be able to run on both cloud computing platforms and local systems	Possibility to have a local tuple store	
Updating and bug fixing, Native support for modularity, Adaptable design	On the fly updating to achieve high SLAs, preserve application state during updating	Uncoupling of agents in space and time. No direct communication that could be broken, state is preserved by tuple store	
Multi tenancy	Isolation and confidentiality inherent in the platform	Multiple security architectures thinkable e.g. plugins for tenant code execution in Safe Haskell [TMPJM12], application execution in light weight VMs, Rusello et al. show that confidentiality of tuples, data and state therein is possible [RDD$^+$08]	

Table 4.3: Eleven requirements of business applications [KDH11]

Application Requirement	Explanation	Haskell	Haskell based DSL
Dynamic placement	Platform chooses where exactly to deploy a worker, re-balancing in case of changes in load		
Consumption based cost tracking			

4.8 Intermediary result

Table 4.4 summarizes the ratings of the Haskell programming language in the evaluation criteria category Coordination.

Table 4.4: Rating of Haskell functions as easy access composable lightweight units that can be coordinated to achieve scalable distributed applications

Requirement	Rating	Details		
COO1 Entities suitable for coordination	2 Applications in the FPL are comprised of entities that miss one or more characteristics or they are not easily accessible.	Functions are entities suitable for decoupling and parallelizing software on the local system only.	Plugin architectures can abstract away the need for a main function	Functions do not reduce the size of the required infrastructure.
COO2 Mechanisms of coordination	3 The FPL must be extended to implement the required mechanism.	Read ASM4	Read ASM4	Read ASM4
COO3 Medium of coordination	0 Not scored			
COO4 Semantics / rules protocols	1 The FPL supports the implementation of a coordination language independently from a third party platform or runtime	Type safety, algebraic data types, generalizations	Syntax flexibility with industry grade lexer and parser.	

Table 4.4: Rating of Haskell functions as easy access composable lightweight units that can be coordinated to achieve scalable distributed applications

Requirement	**Rating**	**Details**		
COO5 Degree of Decoupling	2 Node-level shared nothing architectures can be implemented using third party libraries but they are not cost effective and not sufficient.	Read ASM3	Read ASM3	Read ASM3
COO6 Application domain	2 The FPL needs to be extended in order to be relevant for the application domain computing clouds.	The features designed to support parallel and concurrent computing on the local system need to be extended to work across physical boundaries or new means, for example a coordination language that by default supports uncoupling in space and time needs to be implemented.		

4.9 Summary

In this chapter we answer the questions:

1. What if cloud computing did not mean to wrap VMs and operating systems that need to be administered and protected around atomic units of scale, such as processes or thread?
2. What if cloud computing would have small units of scale that support distribution regardless of boundaries and scale of physical hardware?
3. What if cloud computing was not server centric?

We have investigated functions as small units of scale for computing clouds and have demonstrated domain abstractions to achieve a functional tuple space implementation based on an in-memory key-value store. We follow Lee's argumentation [Lee07] that "We should not replace established languages. We should instead build on them." and agree that under these circumstances coordination languages have special merit. Consequently, we have introduced the EDSL Cwmwl script that virtualizes (application) memory, data storage, and IPC, and detaches them from physical servers and operating systems. We have increased the expressive power of coordination languages by the use of ACP and demonstrated that undesirable tuple space access patterns, resulting for example from sequential algorithms, can be abstracted away from the resulting coordination language.

In our current implementation, Cwmwl uses a centralized tuple space that on the one hand creates the problem of a single point of failure, but on the other hand makes the Cwmwl tuple space more capable in terms of migration than a distributed implementation. The elasticity of the Cwmwl tuple space itself is left up to the capabilities of the Redis key-value store. Unlike, for example RIAK, [ria] or the Amazon Dynamo key-value store [DHJ$^+$07b], the distributable version of Redis (named "Redis cluster") is currently under development, and distribution over multiple instances (e.g. via sharding) is left up to the developer. In an industrial-grade PaaS fabric, the question what use cases the benefits of a portable tuple store outweigh the benefits of a distributed tuple store will need to be investigated.

Chapter 5

Integration with Data

Computing clouds focus on the five key concepts as defined by NIST [MG11]: self-service, broad network access, resource pooling, rapid elasticity and measured service, thereby paying no attention to the importance of data. Instead, about the same time when computing clouds became popular, data took on a life of its own, giving rise to the map reduce computing framework and data locality, NoSQL data stores and the notion of 'Big Data', resulting in two disparate paradigms: Computing Clouds and Big Data. For example, large Cloud Service Providers (CSPs) sell map reduce workflows as an application service that is separate from their IaaS or PaaS offerings.

The lack of harmonization of data and computation is holding back computing clouds from evolving further into utility clouds. Data gravity is perceived as gravity pull associated with data volume. Data of large volume develop large gravity that pulls computation to it, which, somehow, contrasts with the computation-centred principle of current computing clouds, where computational resources is centralized and fixed.

In the previous chapter, we implemented a PaaS architecture that combines the LINDA coordination language, an in-memory key-value store, with functional programming to preserve state and facilitate execution and coordination of *functions. In this chapter we discuss whether our architecture, findings and paradigms from the previous two chapters can help to bridge the gap between computing clouds and big data.*

5.1 Data

Data is an emerging asset class with the term "Big Data" being the first widely adopted mediaspeak to describe this phenomenon. Laney [Lan01] described data as having the three dimensions **V**olume, **V**elocity and **V**ariety, which gives an adequate description of the attributes and challenges found in 'Big Data'.

5.1.1 Volume

Data is perceived as big because of its impressive volume. The Map Reduce framework, with one of its core components being the Hadoop File System (HDFS), was the first step to make parallelized analytics on cheap multicore commodity hardware feasible for programmers. HDFS does not require data to be casted into any fixed schema before it is written to disk, and thus outperforms traditional RDBM in storing data [PPR$^+$09]. The actual Map Reduce query framework used on top of HDFS vastly outnumbers traditional RDBM in the number of CPU cores that can be utilized to run distributed queries over the data at hand [Rea09, OM09].

5.1.2 Variety

The variety of data reflects the range of data types and sources. For example, Hadoop Framework is designed to ingest and work schemaless with any type and format of data. Data that is free from constraints is often referred to as 'unstructured data'. It is not unusual that, at the time of ingestion, the structure of the data has not yet been specified or discovered.

The Redis key/value store goes even further and dissolves the notion of data types by providing binary safe key/value pairs that can store any data type. Data Mashups that combine data of different provenance and formats to extract new wisdom are a further expression of the

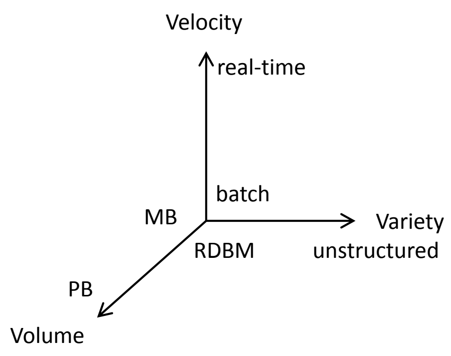


Figure 5.1: 3V Data.

importance of data variety. The internet of things, that is basically an internet of sensors, will certainly provide a plethora of interesting mashups of varying data.

5.1.3 Velocity

High velocity data as it occurs in social networking feeds and media data, such as video and audio, are the first qualities of high velocity data that come to mind. However, streaming data with various velocity is omnipresent. According to Intel [Int13], in the year 2013 the internet consisted of 640TB of data in motion per minute. Finally, the only way to analyse "offline" static data, for example on the HDFS file system or stored in an RDBM, is by transforming it into a stream and bringing it into the main memory of the computing node bit by bit. However, while traditional programming models and programming languages show no flexibility in altering the query or computation once a loop that loads data bit by bit into main memory has been started, stream processing frameworks are expected to be flexible and allow ad-hoc changes of the computation.

5.1.4 Data Gravity —the fourth dimension of data

In 2011, McCrory described in a meanwhile famous Blog Post [mcc] the qualitative characteristics of data that he called "Data Gravity". Since then these principles have resonated in the commercial communities dealing with 3V data. "Data Gravity" is a metaphor describing the economics of data. Data is better to stay where it is and not to be shipped around, no matter how big or small the amount of data may be. A finding that is supported by Jim Gray [Pat03], who stated that compared to the cost of moving data, everything else would be negligible. Consequently, McCrory stated that data must have something that is comparable to a gravitational pull that pulls services and applications to it rather than the other way around: Data Gravity.

This blends with the Map Reduce programming model where computations, for example, batch jobs written in Java or Python, are brought close to the data rather than the other way around [1]. Under the assumption that Data Gravity exists, it seems reasonable to question whether computing clouds, where computation resources, expected to be mobile under the sense of data gravity, are centralized and rationalized (illustrated in Figure 5.2), are the landmark innovation that is required to put utility computing into practice.

There following approaches bring computation to data and mitigate data gravity:

- Agent platforms, such as Java RMI[2] or CORBA.
- Code mobility as implemented in ML5 [VCH07].
- Fog Computing.

Agent platforms

The authors of this thesis define agent platforms as frameworks that allow developers to launch code/agents into a network of nodes. State and data is communicated between agents using

[1] Exemptions exist where no resource close to the required data can be scheduled. In these cases YARN transfers data over the network to the computational resource.

[2] For a preliminary discussion how Java RMI relates to distributed processing in computing clouds read section 4.7.1 of this thesis.

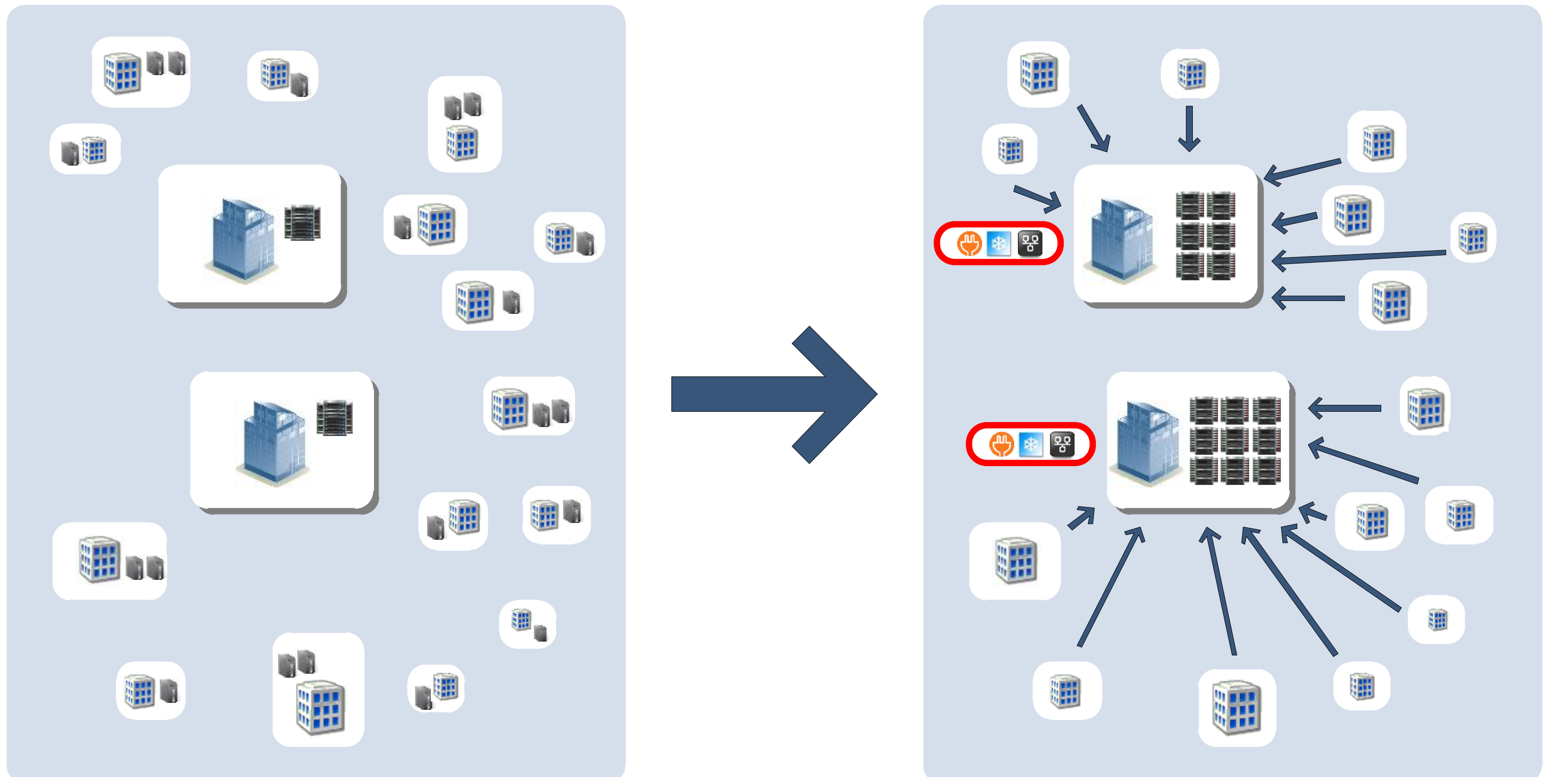


Figure 5.2: Resource pooling in computing clouds.

messages, otherwise this approach is stateless. Over the past years, a number of promising imperative agent platforms have evolved and been abandoned again, for example, Aglets, Concordia and Objectspace Voyager [PK98]. The platforms differ in, for example inter-agent communication or whether there is a central agent and service directory or not.

It must be noted that agent platforms and their general architectures are similar to function service platforms and thus to the architecture that we have been using to investigate coordination (Figure 4.4). A key difference that we observe is whether the administrative tool set that must be used to launch the agents is decoupled from the agents and possibly their programming language or not. While modern function service platforms strictly decouple the administrative tool set from the functions/agents that are being launched, traditional agent platforms such as Aglets and our cwmwl framework do not decouple agents and tool set making the framework admittedly complex.

Code mobility

The authors of this thesis define code mobility as migrating a running program (including all data and state) from one system to another. Examples are the FPLs ML5, LuaTS (based on a Tuple Space) as well as the imperative sPickle Python library. According to the observation of the authors of this thesis, there is not a single approach that has been consistently maintained and used in operational business applications—also not in imperative programming languages.

Fog Computing

Data Gravity is further amplified by the increase of the number of powerful heterogeneous mobile devices and the internet of things (IoT). IoT specialists have recognized that on the long term it may not be feasible to transport all (sensor)data from the edge to the cloud (read also Section 5.4 and Illustration 5.4) for the implementation of advanced analytics and feedback loops. IoT specialists are currently researching architectures where data is kept at the edge and analysed close to its origin. Figure NN illustrates a general IoT architecture with the three elements IoT Edge, IoT Gateways and IoT Platform (that is frequently based in a public cloud). IoT architectures that consider data gravity and bring computation close to the edge

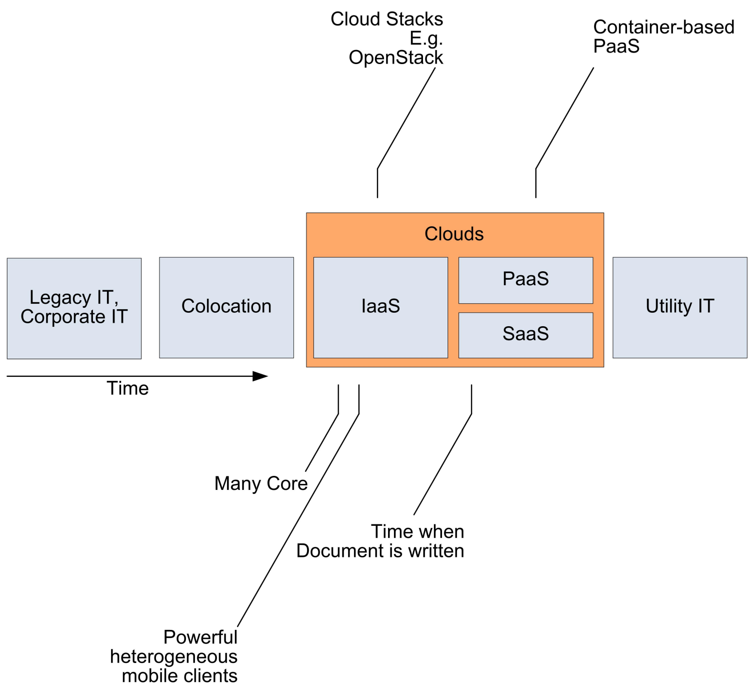


Figure 5.3: Evolution of information technology consumption.

are called "Fog Computing" [BMNZ14]. As with data gravity, fog computing is currently still in its infancy. During a literature review, we could identify [HLR+13] and [LGL+15] researching appropriate programming models as part of their research. However, the described programming models are not yet detailed enough and not supported by experimentation.

5.2 Functional Programming in Map Reduce

The Map Reduce programming model as proposed by Google [DG08b] is currently the prevailing model to analyse voluminous data sets that are offline on disks. Map Reduce is a batch oriented computational model that works in approximate parallelized polynomial time complexity: $O(N^m/k)$ [3]. Although Map Reduce excels in ad-hoc queries over unstructured offline data, it is not a good fit for recursive computations that are prevailing in Machine Learning [ZCF+10].

[3]Where N is the number of data records, m an exponent specific for the algorithm and k the number of CPU cores or computation nodes.

The most popular implementation of Map Reduce is Hadoop [apab]. Yet Another Resource Negotiator (YARN) [VMD$^+$13], frequently referred to as "Hadoop 2", is the most recent version of Hadoop that implements some trade-offs and changes to better support, for example, recursion and programming models that are neither based on the Map Reduce programming model nor interpreted into on the Map Reduce programming model, such as the initial SQL-like data platforms for Hadoop. With YARN Hadoop started to evolve from the Map Reduce programming model into a High Performance Computing (HPC) Platform and shares some of the advantages and constraints of HPC.

Map Reduce and Functional Programming are closely related because:

- Map Reduce and FPL paradigms are closely related.
- Next generation compute over HDFS is an implementation of the actor model.
- FPLs are declarative.

Map Reduce and FPL are closely related. The Hadoop Map Reduce model has some similarity to the Map Reduce model that is the mandatory basis of recursion schemes for list processing in Haskell. Lämmel [Läm08] provides an in-depth discussion of how deeply the Hadoop Map Reduce model is related to the Haskell Map Reduce programming model and gives many examples written in Haskell notation. Lämmel concludes that “MapReduce and Sawzall must be regarded as an impressive testament to the power of functional programming– to list processing in particular. Google has fully unleashed the power of list homomorphisms and friends for massive, simple and robust parallel programming. The original formulations of the models for MapReduce and Sawzall slightly confuse and hide some of the underlying concepts”.

Example: Close relation of the Hadoop Map Reduce paradigm and the Haskell functions map and fold

In the Hadoop Map Reduce paradigm Map operates on a list of values in order to produce a new list of values, by applying the same computation to each value. Reduce operates on a list of values to collapse or combine those values into a single value by applying the same computation to each value.

In Haskell, we can use the functions `map` and `fold`. Similar to the Hadoop Map Reduce paradigm, the Haskell map function operates on a list of values in order to produce a new list of values: `map (+4) [1,6,4,10]` or `map (\x -> x + 4) [1,6,4,10]` where the result will be:

```
ghci> map (\x -> x + 4) [1,6,4,10]
[5, 10, 8, 14]
```

Similar to the Hadoop Map Reduce paradigm, the Haskell `fold` function operates on a list of values to collapse or combine those values into a single value by applying the same computation to each value: `sum' :: (Num a) => [a] -> a`
`sum' xs = foldl (\acc x -> acc + x) 0 xs` using the function sum' gives the following results:

```
ghci> sum' [1,6,4,10]
21
```

Next generation compute over HDFS is implementing the actor model. Apache Spark [ZCF+10] is an in-memory engine for data processing based on an implementation of the actor model for parallelization. Apache Spark can be added to Hadoop and other distributed data stores such as HBase or Cassandra.

The actor model for parallelization can work locally and across physical nodes in distributed environments. The actor model is an inherent part of Erlang. Currently, no mature Haskell library implements the actor model. Haskell certainly has equivalents for parallelization, such as `forkIO` to create lightweight processes, the `par` monad, `MVar` or STM, but these implementations focus on parallelization on the local system and not over large distributed compute farms. For more information how the actor model relates to FPLs we refer to section ASM3: Support for node-level 'shared nothing' architectures earlier in this thesis.

FPLs are declarative FPLs are declarative programming languages. The Structured Query Language (SQL) is also a declarative programming language. Declarative languages specify what result should be produced, but not (like imperative programming languages) how to put this into practice, in theory giving the compiler the freedom to re-arrange and optimize the code and arrive at a better result than a software developer would.

But why is a declarative language important for the support of data analytics at scale? Declarative languages, such as SQL, are strongly linked to modern relational databases and data manipulation use cases. SQL allows the database to do algebraic optimizations, such as picking the sequence of joints (in complex queries) that needs the least execution time. In practice we must consider that the compiler is also written by a human only; compilers cannot look at code, understand its meaning and do the best optimization. This may change in the future when compilers are based on Artificial Intelligence. We must admit however that compiler based optimization for SQL and FPLs work very well for the majority of use cases that the implementer did foresee; not for edge cases, of course.

However, HDFS and Hadoop are written in the Java programming language and not in a FPL. While FPLs contribute a number of valuable paradigms developers that really need to get something done seem to be better off by implementing the required paradigms in a a mainstream imperative programming language, such as Java. The authors of this thesis are aware that any conclusion would be not constructive. The rating below is based on the above observations, rather than on speculation about what the reasons for this may be.

5.3 Functional Programming in Stream processing and Data flow programming

There seems to be a substantial gap between audiences as to how they understand data flow programming. While traditional research focuses on data flow programming as abstraction to model (parallel) programs as data flow graphs, there has recently been more attention to the use of the data flow programming paradigm in stream processing. While the one does not exclude the other, cloud computing and 3V data have considerably changed the perception of data and streams.

Stream processing models that operate on real-time data with approximate linearithmic time complexity O(N log N) are increasingly gaining importance for the following reasons:

- They operate on real-time and temporal data as it occurs, for example, in online advertising, elimination of online advertising fraud, sensors or social media streams [4].
- Their time complexity formula O (N log N) is not influenced by number of computing nodes while the Map Reduce framework draws its strength from its scalability over 1000s of nodes and cores.
- Their use can significantly reduce the required hardware base and energy.
- They support recursive algorithms and machine learning.
- They are not invasive and developers are not forced to change their algorithms into map and reduce phases. These phases come with an overhead due to the unnecessary addition of tasks (such as shuffle and sort) both in computing and in programming. [TSLZ12].

Johnston [JHM04] described the history of data flow programming languages from the 1970s to approximately the year 2000, and discussed in depth the inherent parallelism of data flow languages as well as their close relation to FPLs. Stonebraker [SÇZ05] states eight requirements that "a system should meet to excel at a variety of real-time stream processing applications", that are listed in Table 5.1. In spite the fact that Stonebraker had not foreseen the NoSQL movement and stipulated that SQL would be the only viable query language, nearly a decade later, most of his requirements are still applicable to processing high velocity data. Stonebraker's requirements are listed in Table 5.1, along with the features supported by FPLs and Tuple Space, to indicate that a system combining FPLs with Tuple Space, as reflected in the section "Blueprint for the next-gen computing cloud" of this paper, would meet most of Stonebraker's requirements.

The attention to data parallel programming has somewhat increased since 2007, when Microsoft research developed Dryad [IBY+07], a general purpose framework for data parallel

[4]According to IAB Internet Advertising Revenue Report conducted by PricewaterhouseCoopers (PWC) the total revenue of online advertising for the first half year 2013 was \$ 20.1 billion with a year over year growth higher than 15% [IAB13].

Table 5.1: Eight requirements that a system should meet to excel at a variety of real-time stream processing applications [SÇZ05] and how they are met by the cwmwl framework..

Requirement	FPL	Tuple-space
Keep the data moving		In memory tuple space.
SQL on streams	FPL are declarative, some aspects similar to SQL	
Handle Stream imperfections		Decoupling in space and time.
Predictable outcome	Immutability of data. Lazy evaluation. Evaluation eventually ends.	
High-availability		
Stored and Streamed Data		Supports Lambda Architecture to combine for example Map Reduce data and streaming data.
Distribution and Scalability		
Instantaneous response, Work pushing rather than work stealing		

programming. The framework exposes an API to describe acyclic DFGs that are eventually parallelized and ran by the execution engine. Many authors have subsequently implemented the Dryad API structure and execution model.

Tran [TSLZ12] proposed AROM, a framework for processing Big Data with DFGs and FPLs. AROM aiming to provide a Dryad-like API that allows jobs to be expressed as DFGs, was implemented in Scala[5] and built on top of the Akka toolkit[6]. Tran argued that FPLs provided an ideal basis to implement the required generic and reusable operators because they support higher order functions. Tran recognized the importance of highly scalable message passing services for the performance of distributed data driven systems.

The dependency between data flow systems and the underlying protocol that is used to propagate either data or control information in the system has been recognized and partially investigated. For example, Li [LTS$^+$08] described a high performance architecture that supports out of order processing of streams; Murray [MMI$^+$13] also addressed the issue in more depth and concluded that a specialized protocol would provide better performance. The findings of Tran, Li and Murray are in line with our previous research into inter-process communication for computing clouds [FW12b].

Furthermore, real time processing of high velocity data is of commercial interest where it supports, for example, Impact of workload size on throughput , online advertising and fraud detection. Yahoo S4 [NRNK10], the Storm project[7] [Cha13], Amazon Web Services Kinesis[8] and IBM Infosphere Streams[9] are examples for successful stream processing frameworks that are either commercial products or open source software developed by a commercial company.

Map Reduce and Big Data Stream analytics share many commonalities such as the shared nothing architecture. Marz [Mar13] proposed what he calls a "Lambda Architecture" to integrate both batch processing (Map Reduce) over high volume data and real-time processing of high

[5]Scala - http://www.scala-lang.org

[6]Akka - Toolkit and runtime for building highly concurrent, distributed, and fault tolerant event-driven applications. - http://akka.io

[7]The Storm Project - http://storm-project.net

[8]AWS Kinesis - http://aws.amazon.com/kinesis/

[9]IBM Infosphere Streams - http://www.ibm.com/developerworks
/bigdata/streams/

velocity data by presenting a data mashup that is the combined result from both its speed layer (real-time processing) and its batch layer (Map Reduce).

5.4 Functions in service platforms for IoT sensor data

Function service platforms, such as AWS Lambda [Inca], Google Cloud Functions [Inc16], Iron Worker [Iro16] or IBM Bluemix OpenWhisk are key architectural elements of cloud based Internet of Thing (IoT) platforms[10]. Function service platforms are also referred to as Event-driven application platform (EDAP), zero infrastructure platform services or event-driving compute service. AWS Lambda defines functions as stateless programs that have no affinity to the underlying platform. Functions maybe written in Java, Node.js or Python, the code size may not exceed 250MB and the execution time is limited to 300 seconds. Functions are triggered by events when, for example, when a new message is written to an AWS S3[11] bucket.

Figure 5.4 illustrates an architecture where a message that is written to an S3 bucket triggers an AWS Lambda functions that fans out to multiple functions, a message queue and a traditional system that is using a REST API. The message response is then written to a shared storage, for example, an RDBM or a Hadoop cluster. Applications, such as analytics platforms or portals, can then access the result sets and visualize them or, in case of the IoT, trigger a response in the real world, to name a few examples.

The fact that end customers cannot (yet) use functional programming languages to write functions for AWS Lambda and the like seems irritating at first. However, the market penetration of Erlang and Haskell is admittedly low and commercial players must offer products and features suitable for mainstream at first. However, the programming model that is required to use function service platforms is very close to what functions in for example Haskell deliver. The authors of this thesis believe that functional programming languages such as Erlang and Haskell are ideal for these use cases and thus have direct and indirect merit for the IoT that is now just at the start.

[10]An IoT platform can be build or rented, in all cases the maintainer will need functional services.

[11]Object storage offered by the AWS cloud computing platform

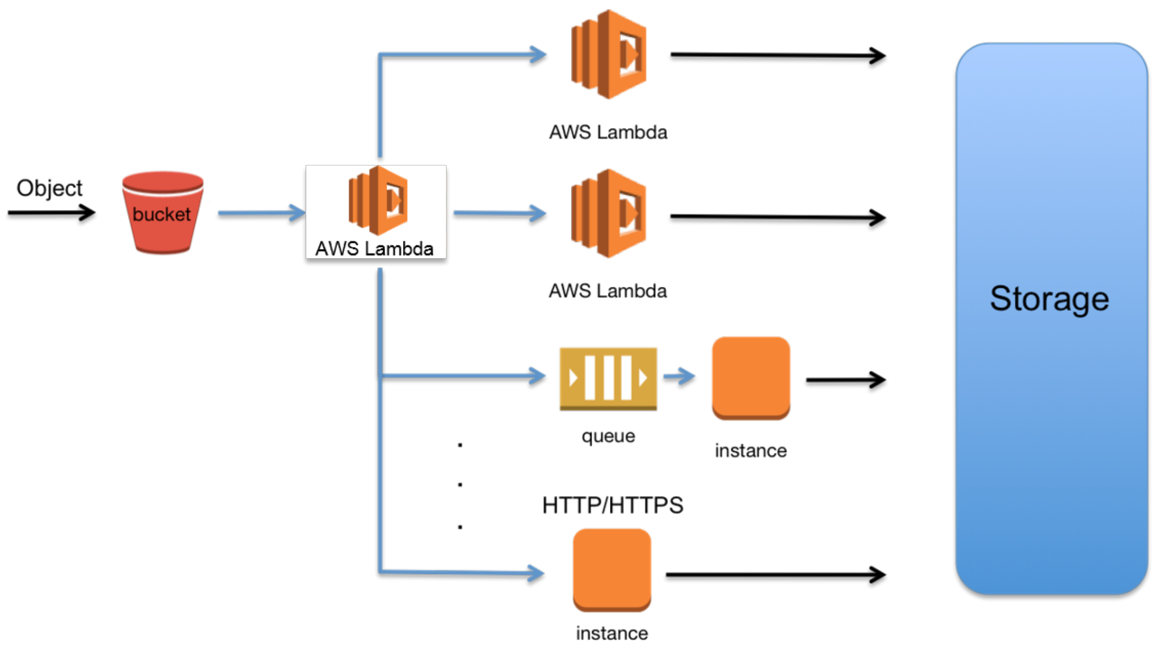


Figure 5.4: Architecture using AWS Lambda functions for data ingestion. Source: AWS.

It should also be noted that the example architecture of an application that is using AWS Lambda functions (Figure 5.4) is similar to the architecture that we have been using to investigate coordination (Figure 4.4). Our design is using a central tuple store to fan out functions and to store the results. In addition, the tuple store can track state without requiring the fan-out into a bolt on message queue.

5.5 Functions, Messaging and Coordination: Reconsidering the problem with data

Fixing data requires a large vision with a sequence of small steps that lead to it. In this section we reconsider our statements about data and apply them to the CMQ/Cwmwl architecture. Based on our research we have identified eight dogmas for the next-generation (”next-gen”) cloud computing platform that, according to Figure 5.3, may very well be similar to a future UC platform. In the following subsections we introduce qualitative evidence illustrating the integration of our architecture with data.

In order to make a significant contribution to fix data our architecture will need to

- Support for (analytics of) Data with challenging *Volume*.
- Support for a *Variety* of Data.
- Support for Data with challenging *Velocity*.
- Overcoming *Data Gravity*.

5.5.1 CMQ/Cwmwl support for Data with challenging Volume

Evidence that an architecture can support data with challenging volume is when the platform is

- Scalable. The CMQ/Cwmwl architecture is scalable because it is a distributed shared nothing architecture that is decoupled in time and space. It is inherently asynchronous.
- Resilient. The CMQ/Cwmwl architecture is resilient because the UDP based CMQ paradigm provides error resiliency on communication level. Functions are stateless and lean units of scale that can be wrapped in containers and invoked within milliseconds.
- Fast. Optimistic UDP based messaging that is faster than traditional TCP based messaging.
- Cost Effective. Cost effectiveness has not been assessed.

The remaining challenge is that the Redis based tuple store used in our test bed is centralized and not distributed. In the following criteria we will show that it is better to implement a number of smaller tuple stores at the edge rather than having a distributed resilient tuple store at the core of the architecture.

5.5.2 CMQ/Cwmwl support for unstructured Data

Evidence that an architecture can support unstructured data is when unstructured data —that is data where the format is not yet known at the time of ingestion, can be ingested without traditional transformation as applied during traditional Extract-Transform-Load (ETL) processes.

This issue has not been investigated during our research.

5.5.3 CMQ/Cwmwl support for Data with challenging Velocity

Evidence that an architecture can support data with challenging velocity is the availability of techniques that can implement analytics or filters on data from streams, such as social networking streams. By default tuple spaces, as used in the Cwmwl architecture, use associative lookup that is also used in data flow programming. Our investigation showed the FPLs and Data Flow Programming are closely related. We refer to Section 5.3 earlier in this chapter in that regard.

5.5.4 CMQ/Cwmwl and Data Gravity

Evidence that an architecture minimizes data gravity is:

- Minimization of data transfers (volume, frequency). The CMQ/Cwmwl architecture consists of a message queue that queues smaller messages until they have reached certain volume or age.
- Implementation of edge computing/fog computing. Functions are stateless units of scale that can also be invoked at the edge, given that the edge implements sufficient means of computation and coordination.

5.6 Fixing Data by Eight Dogmas

1. Associative lookup should be preferred.
   As used, for example, in Dataflow programming, Content Addressable Memory (CAM)

tables of network switches [12] and Tuple Spaces.

2. The fabric must be a dynamic, scalable distributed system.

   Achieve an inherently asynchronous framework for the required computations using a shared-nothing paradigm.

   Support elasticity and large scale operations, by providing, for instance, suitable programming language abstractions, error resiliency, modularity and flexibility.

3. Next-gen platforms must handle stored data and streamed data.

4. A global name space to virtualize data objects and apps must be provided by the next-gen platform.

5. Transmission protocols are the main.

   Data transfer and message passing should be optimistic and based on the UDP protocol [13] preserving the asynchronous character of all components and communications.

6. Declarative programming, such as in SQL and FPLs should be preferred.

7. The emulation of traditional tier-based computing needs to be removed from computing clouds and replaced with a unified fabric.

8. Next-gen cloud computing platforms need to deliver abstract services, not limited to web services.

   In order to match abstract services and data, an interface description language (IDL), similar to Thrift IDL [SAK07] or Avro IDL [apaa] may be needed.

5.6.1 Blueprint for the next-gen computing cloud

The authors of this thesis argue that a distributed in-memory Tuple Space should serve as backbone for a distributed scalable fabric that acts as an abstraction layer and virtualizes both

[12]Notably, high speed data centers already moved from being routed environments to layer 2 data center network fabrics that are based on CAM. This has also advantages in fault tolerance, the support for data center virtualization and migration of workloads.

[13]The use of UDP is explained and supported with facts and measurements in our previous research [FW12b]

data and computation. Applications and services are encapsulated in lightweight containers, such as Docker [web15], and define data flows as flows that connect and flow through the encapsulated services (hereafter 'app'). To manage the flow of computations and events, we use the principle of data gravity and distinguish between streamed data and stored data:

- **Streamed data**, like events, that need to find an encapsulated app to be processed. In this case the encapsulated app, that is data as well, has the higher gravitational pull and attracts event data.
- **Stored data**, like larger (file) objects, that have high gravitational pull and bring services temporary close to them for the time needed to be processed.

Figure 5.5 is a blueprint diagram for a data oriented next-gen utility that implements the eight dogmas. The eight dogmas are matched by seven architectural steps:

1. Distributed Shared Memory is used to implement associative lookup. Data and computations are matched and routed based on what they are rather than what the assigned IP address is.
2. An in-memory Tuple Space is used to provision a means of coordination of data and computation. Policies, such as Constraint Handling Rules (CHRs) [Frü95], are a fundamental aspect to determine what level of storage is required, such as the time (length) data should remain in the low gravity segment or what type of app can access particular data in the fabric (security)[Lin14].
3. A site global name space makes data storage transparent to the (application) developer.
4. The Tuple Space is deployed on servers and networking devices.
5. Declarative programming based on, for example, functional programming is preferred.
6. Apps are considered to be data as well and flow according to the principles of gravity. The Tuple Space makes tier-based architectures obsolete [web] [FW14a].
7. IDLs should be integrated with inter-process communication based on the User Datagram Protocol (UDP).

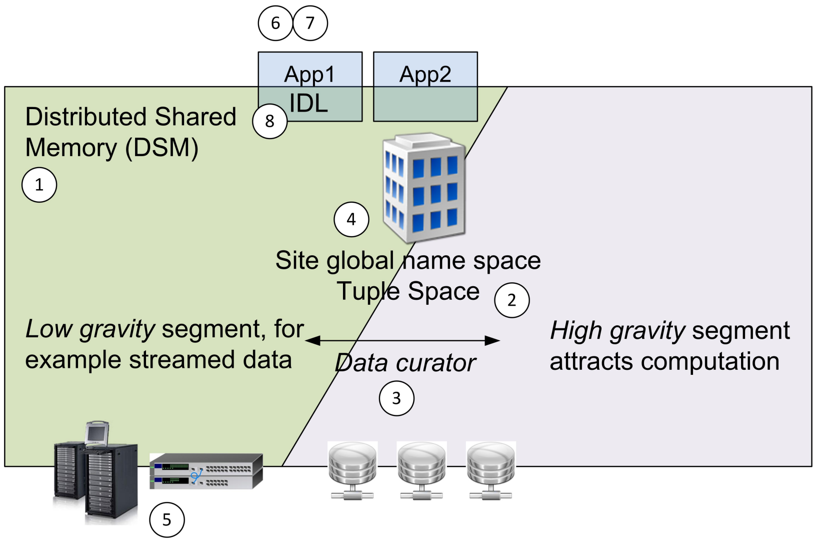

6
7
App1
IDL
App2
8
Distributed Shared Memory (DSM)
1
4
Site global name space
Tuple Space
2
Low gravity segment, for example streamed data
High gravity segment attracts computation
Data curator
3
5

Figure 5.5: Blueprint diagram for a data driven next-gen utility

5.6.2 Challenges and Opportunities

Van Lingen [Lin14] states that it is unlikely that there will be one next-gen data fabric. Instead there will probably be multiple fabrics managed by multiple companies and entities similar to modern IP networks. Next-gen data fabrics should therefore be connected to each other through gateways creating a fabrics-to-fabrics connection where needed. The fabric needs to be capable to support multiple protocols, transportation layers, and mappings to data and services; questions arise concerning the data security and privacy.

Tuple Spaces have been around for a long time with relatively moderate adoption rates. On the one hand modern in memory key-value stores such as Redis are in a way very similar to tuple spaces. On the other hand, tuple spaces have much more to offer than a simple key-value store. For example, values can be part of distributed hash tables, information for routing algorithms, references to stored data, apps and workflow scripts layers can be built on top of Tuple Spaces to provision the next-gen cloud computing platform.

Similar, Dataflow Programming and Functional Programming have also been around for a long time, but we now observe a significant uptake. Many mainstream programming languages are now extended with functional elements and it needs to be assessed separately whether this meets the spirit and requirements of functional programming that traditionally adheres strictly to its paradigms, like the immutability of data.

Eventually the innovation in architecture needs to be matched and supported by the silicon on the same level as the innovation in hypervisors. Asa result core functionalities, such as curating low gravity and high gravity data and provisioning of a unified namespace, can become part of the infrastructure rather than of the software.

The integration of data remains difficult and with the advent of Big Data we are just at the doorstep in understanding the role, dimensions and implications of data. If we expect current computing clouds to move forward and assume a new role as real utilities we must harmonize computation and data. Clearly this affects a number of research communities that are involved in distributed computing, data, networking and, not to forget, security and privacy. Clearly, the importance of data will further increase and poses tremendously interesting research questions and opportunities.

5.7 Summary

Due to our interest in data, we have included this Chapter in our evaluation that is based on literature research and the authors' experience with data. We argue that a distributed in memory Tuple Space (as proposed in Chapter 4) is effective in serving as backbone for a distributed scalable fabric that acts as an abstraction layer and virtualizes both data and computation. We introduce the principle of data gravity that reflects the nature of data to attract computation, not the other way around. The exploration of data uncovered some interesting use cases for functions, such as IoT service platforms that do not require internode message passing and coordination. Although out of scope of this thesis, the authors find that the FPL Haskell may be a much better fit for these use cases than for computing clouds.

Chapter 6

Conclusion

This research was set out to explore FPLs and computing clouds. When we started, in the year 2011, the general academic literature was sparse and we could only identify three academic authors (Armbrust [AFG$^+$10], Birman [BCvR09] and Foster [FZRL08]) that had recognized cloud computing as new paradigm that is difficult to understand for academic researchers. Birman writes, “hot research topics seem to be of limited near-term importance to the cloud builders, while some of their practical challenges seem to pose new questions to . . . researchers.” The authors on this subject achieved an academic consensus as to what cloud computing is and what new questions and paradigms need to be added in future academic research. This thesis is based on the authors’ commercial experience with computing clouds, experimental work, and literature research. The thesis sought to answer two questions:

1. What are the new paradigms and questions of cloud computing builders?
2. Do FPLs fit with the new paradigms, and do they offer special merit in alleviating the new questions and challenges?

We conclude that our experimental work and demonstrator code was geared towards compensating that Haskell has been designed with the local system in mind (for more information we refer to Sections 4.7.1 to 4.7.6 of this thesis). Although Haskell implements all the right paradigms and approaches, it is tied to the local system. Haskell is excellent for use cases that do not require the distribution of the application across the boundaries of (physical or virtual) systems but not appropriate as a whole for the development of distributed cloud based workloads that require communication with the far side and coordination of decoupled workloads. We argue that interaction of functions or any other lightweight unit of scale in computing clouds need to be dealt with in ways that are intrinsically different from objects that interact in a single system. However, Haskell may be able to qualify as a suitable vehicle in the future with future developments of formal mechanisms that embrace non-determinism in the underlying distributed environments leading to applications that are anti-fragile [Tal12] rather than applications that insist on strict determinism that can only be guaranteed on the local system or via slow blocking communication mechanisms.

6.1 Critical Assessment

The main findings are chapter specific and were summarized within the respective evaluation chapters: Chapter 3 'Asynchronous operations and messaging' and Chapter 4 'Coordination. To help with assessing Haskell a set of evaluation criteria based on the new paradigms and challenges of computing clouds was proposed and scrutinized using real-world implementations and artefacts. The evaluation criteria and implementation details highlighted that our contributions are in the connection of the new paradigms, questions, and challenges with existing academic research. Within the discourse we frequently find the need to challenge established paradigms.

Chapter 3 finds that in computing clouds, Haskell needs to support node level shared nothing architectures and provide inter-node message passing. We present the lightweight, UDP based message queue CMQ. The concept to use UDP instead of TCP is motivated by our understanding that, in Cloud Computing, omnipresent off-the-shelf technologies (both in hard and software) are encouraged, and if preventing errors from occurring becomes too costly, dealing with the errors may be a better solution. Although CMQ is a message queue oriented communication approach, CMQ is different from the conventional MOM approach because it challenges a number of assumptions under which conventional MOM is built. For instance, in conventional MOM, messages are 'always' delivered, routed, queued and frequently follow the publish/subscriber paradigm. It is often accepted that this requires an additional layer of infrastructure and software where logic is split from the application and configured in the additional layer. On the contrary, CMQ does just enough. It does not offer any guarantees, such as reliable transmission, thus is very light weight with low overhead and fast speed. Although it does not offer guarantees, it appears to be stable in the presence of errors.

At the given time, the implementation status of CMQ remains basic and it is left up to the end user or developer to detect message loss and replace (rather than resend) lost messages with newer messages if required. Further research into byzantine fault tolerance is required to illustrate the feasibility of the new paradigm even when multiple errors occur at the same time.

In Chapter 4 we answered the questions:

1. What if cloud computing did not mean to wrap VMs and operating systems that need to be administered and protected around atomic units of scale, such as processes or thread?
2. What if cloud computing would have small units of scale that support distribution regardless of boundaries and scale of physical hardware?
3. What if cloud computing was not server centric?

We follow Lee's argumentation [Lee07] that "We should not replace established languages. We should instead build on them." and agree that under these circumstances coordination languages have special merit. We find that while Haskell functions provide easy access units of scale, it must be extended with a means for coordination. We present the PaaS framework CWMWL, which is intended to rethink PaaS design and to merge brute replication and reclustering (which is the current methodology to implement PaaS) with distributed computing to improve efficiency and cost. Our foremost design goal is simplicity by achieving a novel unified platform rather than virtualizing and replicating the implementation of a load balanced web application that has existed since the end of the 1990s. Multiple times, we challenge the academic concept of deterministic computing arguing that nondeterministic means must be introduced where needed to fit the nature and requirements of computing clouds–without disrupting the essential determinism of programming languages. In both, Chapter 3 and Chapter 4 we frequently felt as if we turned back the clock 30 years but in a good way.

After literature research, we decided early on in our research to look for solutions on programming language level and engaged into development and experimentation. Especially in this chapter there is the question whether coordination needs to be part of the programming language or can better be delivered via an external coordination service. Rather than going ahead and implementing primitives for coordination, we could have taken an approach similar to Chapter 3 where we evaluate external coordination services first.

Overall, the question what needs to be part of a programming language or DSL vs. what can be consumed as external service, could have been investigated in more depth.

Due to our interest in data, we included literature research and the authors' experience with data in Chapter 5. This chapter is not backed by experimentation because the importance of data became only evident towards the end of this project. We argue that a distributed in memory Tuple Space (as proposed in Chapter 4) is effective in serving as backbone for a distributed scalable fabric that acts as an abstraction layer and virtualizes both data and computation. We introduce the principle of data gravity that reflects the nature of data to attract computation, not the other way around. The exploration of data uncovered some interesting use cases for functions: serverless computing and the IoT that frequently do not require internode message passing and coordination. Although out of scope of this thesis, the authors find that the FPL Haskell may be a much better fit for these use cases than for computing clouds.

The findings in this chapter are qualitative and, given that data gravity, edge computing and fog computing are still in their infancy we could not contribute quantitative measures suitable to assess the suitability of a programming language.

6.2 Future directions

This thesis has made a substantial step in describing the paradigms and questions of cloud computing builders that are driven by practical challenges. With the advent of stateless micro service architectures based on Docker and the IoT many paradigms have gained importance for application deployments on premise[1]. We have validated the assumption that the FPL Haskell has special merit for computing clouds and find that while this is true in theory, in practice it does not hold because Haskell's powerful (parallelization) features have been designed for a single system. Our work opens up a variety of interesting possibilities for future work. In particular there are three directions: (i) exploiting light and robust communication protocols for distributed environments, (ii) study of the integration of functions with third party communication services, such as etcd [Incc], and (iii) investigate the potential role of FPLs in serverless compute and the IoT where currently functions are used that are not based on FPLs.

Clear applications lie in the development of anti-fragile applications with stateless micro service architectures, serverless compute and in the IoT. The findings of Chapter 3 inform new protocol design in distributed environments where anti-fragile applications will need anti-fragile communication protocols. Otherwise, the fallacies of distributed computing (see Section 1.7) cannot be mitigated.

While the approaches to coordination presented in this thesis are able to coordinate functions in distributed environments, a comprehensive coordination mechanism would require much more than we have implemented and make the final product a huge platform. The authors of this thesis believe that trying to hide complexity in one-size-fits-all platform such as Java Application Servers will not be the right way to go. At this point in time the authors of this thesis believe that the integration of functions with modern third party coordination services such as Google Kubernetes or Apache Mesos, that were not available prior to this thesis, must be investigated before standing up yet another coordination framework. On the other hand, we can also envisage a composable serverless runtime operating system for functions.

The concept of 'Data Gravity' that we have adopted in Chapter 5 needs more investigation as to its overall implications on compute at the edge, as well as its benefits concerning data privacy and other data residency issues. It must be noted that the NoSQL movement has adopted the key paradigms of computing clouds. For example, it questions the CAP theorem and, where appropriate, favours eventual consistency before expensive guarantees that are hardly needed.

[1]Application developers would most likely choose the term 'application delivery' instead of 'application deployment' because Linux containers managed by Docker are frequently used for Continuous Improvement (CI) and Continuous Delivery (CD)

Also, there the clock has been turned back 30 years but in a good way.

While commercial pressure has meanwhile forced IaaS providers to abstract away their key paradigms from the enterprise customer making the overall experience similar to legacy compute environments for monolithic applications, in deployments and runtime environments of newer applications the paradigms that we describe in this thesis will undisputedly find their way into academic research agendas.

Appendix A

Time profiling

```
     Thu Aug 23 14:22 2012 Time and Allocation Profiling Report  (Final)

        timeprof_cmq +RTS -p

     total time  =        7.47 secs   (7475 ticks @ 1000 us, 1 processor)
     total alloc = 3,542,023,928 bytes  (excludes profiling overheads)

COST CENTRE    MODULE                 \%time \%alloc

collectPong    Main                    34.9   18.8
cwPop          CMQ                     15.1    0.1
append         Data.Serialize.Builder   9.4    5.6
encodeListOf.\ Data.Serialize.Put       7.9    6.6
sendPing.\     Main                     3.2    3.6
unsafeLiftIO.\ Data.Serialize.Builder   2.6    0.1
writeN         Data.Serialize.Builder   2.5    3.2
withSize.\     Data.Serialize.Builder   2.5    0.0
writeNBuffer   Data.Serialize.Builder   2.2    8.3
appendMsg      CMQ                      1.9    1.3
cwPush         CMQ                      1.8    0.5
put            Data.Serialize           1.7    1.0
putWord64be.\  Data.Serialize.Builder   1.6    0.0
execPut        Data.Serialize.Put       1.4    0.0
putWord8       Data.Serialize.Put       1.2    0.0
tell           Data.Serialize.Put       1.0    2.3
newBuffer      Data.Serialize.Builder   0.3   45.7
```

```
                                                                                   individual     inherited
COST CENTRE                            MODULE                                 no.     entries  \%time \%alloc   \%time \%alloc

MAIN                                   MAIN                                   123           0    0.1    0.0   100.0  100.0
 main                                  Main                                   257           0    0.0    0.0    94.2   93.3
  runTest                              Main                                   258           0    0.0    0.0    94.2   93.3
   sendPing                            Main                                   275           0    0.1    0.0    46.4   76.2
    sendPing.\                         Main                                   278           0    0.0    0.0    46.3   76.2
     cwPush                            CMQ                                    279           0    1.8    0.4    46.3   76.2
      transMit                         CMQ                                    451         593    0.1    0.1     2.1    1.9
       transMit.mT'                    CMQ                                    517         593    0.0    0.0     0.0    0.0
       transMit.qT'                    CMQ                                    516         593    0.0    0.0     0.0    0.0
       encode                          Data.Serialize                         455         593    0.0    0.0     1.6    1.8
        put                            Data.Serialize                         461           0    0.0    0.0     0.7    0.6
         putListOf                     Data.Serialize.Put                     462           0    0.0    0.0     0.7    0.6
          tell                         Data.Serialize.Put                     467       24906    0.0    0.0     0.0    0.0
          encodeListOf                 Data.Serialize.Put                     463           0    0.0    0.0     0.6    0.6
           encodeListOf.\              Data.Serialize.Put                     464       24906    0.2    0.3     0.6    0.6
            put                        Data.Serialize                         493       97252    0.0    0.0     0.2    0.2
             putWord8                  Data.Serialize.Put                     495           0    0.0    0.0     0.2    0.2
              tell                     Data.Serialize.Put                     499       97252    0.0    0.1     0.0    0.1
              singleton                Data.Serialize.Builder                 496       97252    0.0    0.0     0.1    0.1
               writeN                  Data.Serialize.Builder                 497       97252    0.1    0.1     0.1    0.1
                unsafeLiftIO           Data.Serialize.Builder                 506       97252    0.0    0.0     0.0    0.0
                append                 Data.Serialize.Builder                 498       97252    0.0    0.0     0.0    0.0
             put.c                     Data.Serialize                         494       97252    0.0    0.0     0.0    0.0
            execPut                    Data.Serialize.Put                     477      146471    0.1    0.0     0.1    0.0
             sndS                      Data.Serialize.Put                     478      146471    0.0    0.0     0.0    0.0
    putWord64be               Data.Serialize.Put                       472        24906    0.0    0.0     0.1    0.0
     tell                     Data.Serialize.Put                       476        24906    0.0    0.0     0.0    0.0
     putWord64be              Data.Serialize.Builder                   473        24906    0.0    0.0     0.0    0.0
      writeN                  Data.Serialize.Builder                   474        24906    0.0    0.0     0.0    0.0
       unsafeLiftIO           Data.Serialize.Builder                   486        24906    0.0    0.0     0.0    0.0
       append                 Data.Serialize.Builder                   475        24906    0.0    0.0     0.0    0.0
    append                    Data.Serialize.Builder                   466       146471    0.0    0.0     0.0    0.0
    mappend                   Data.Serialize.Builder                   465        24906    0.0    0.0     0.0    0.0
runPut                        Data.Serialize.Put                       456          593    0.0    0.0     0.9    1.2
 sndS                         Data.Serialize.Put                       460          593    0.0    0.0     0.0    0.0
 toByteString                 Data.Serialize.Builder                   457          593    0.0    0.0     0.9    1.2
  append                      Data.Serialize.Builder                   459          593    0.3    0.2     0.9    0.6
   put                        Data.Serialize                           468            0    0.0    0.0     0.6    0.4
    putListOf                 Data.Serialize.Put                       469            0    0.0    0.0     0.6    0.4
     encodeListOf             Data.Serialize.Put                       470            0    0.0    0.0     0.6    0.4
      encodeListOf.\          Data.Serialize.Put                       471            0    0.0    0.0     0.6    0.4
       put                    Data.Serialize                           500            0    0.0    0.0     0.4    0.3
        putWord8              Data.Serialize.Put                       501            0    0.0    0.0     0.4    0.3
         singleton            Data.Serialize.Builder                   502            0    0.0    0.0     0.4    0.3
          writeN              Data.Serialize.Builder                   503            0    0.0    0.0     0.4    0.3
           ensureFree         Data.Serialize.Builder                   511            0    0.0    0.0     0.1    0.0
            withSize          Data.Serialize.Builder                   512            0    0.0    0.0     0.1    0.0
             withSize.\       Data.Serialize.Builder                   513        72939    0.0    0.0     0.0    0.0
              ensureFree.\    Data.Serialize.Builder                   514        72939    0.0    0.0     0.0    0.0
           unsafeLiftIO       Data.Serialize.Builder                   507            0    0.0    0.0     0.2    0.3
            unsafeLiftIO.\    Data.Serialize.Builder                   508        97252    0.1    0.0     0.2    0.3
             flush.\          Data.Serialize.Builder                   515          593    0.0    0.0     0.0    0.0
             writeNBuffer     Data.Serialize.Builder                   509        97252    0.2    0.3     0.2    0.3
              writeNBuffer.\  Data.Serialize.Builder                   510        97252    0.0    0.0     0.0    0.0
         putWord64be            Data.Serialize.Put                 479          0   0.0    0.0     0.2    0.1
          putWord64be           Data.Serialize.Builder             480          0   0.0    0.0     0.2    0.1
           writeN               Data.Serialize.Builder             481          0   0.0    0.0     0.2    0.1
            unsafeLiftIO        Data.Serialize.Builder             487          0   0.0    0.0     0.1    0.1
             unsafeLiftIO.\     Data.Serialize.Builder             488      24906   0.0    0.0     0.1    0.1
              putWord8          Data.Serialize.Put                 504          0   0.0    0.0     0.0    0.0
               singleton        Data.Serialize.Builder             505          0   0.0    0.0     0.0    0.0
              writeNBuffer      Data.Serialize.Builder             489      24906   0.0    0.1     0.1    0.1
               writeNBuffer.\   Data.Serialize.Builder             490      24906   0.0    0.0     0.1    0.0
                putWord64be.\   Data.Serialize.Builder             491      24906   0.1    0.0     0.1    0.0
                 shiftr_w64     Data.Serialize.Builder             492     174342   0.0    0.0     0.0    0.0
            ensureFree          Data.Serialize.Builder             482          0   0.0    0.0     0.1    0.0
             withSize           Data.Serialize.Builder             483          0   0.0    0.0     0.1    0.0
              withSize.\        Data.Serialize.Builder             484      49219   0.0    0.0     0.1    0.0
               ensureFree.\     Data.Serialize.Builder             485      49219   0.0    0.0     0.0    0.0
    newBuffer                   Data.Serialize.Builder             458          0   0.0    0.5     0.0    0.5
 transMit.(...)                 CMQ                                454        593   0.0    0.0     0.0    0.0
 transMit.a                     CMQ                                453        593   0.0    0.0     0.0    0.0
 sendq                          CMQ                                452        593   0.3    0.0     0.3    0.0
put                             Data.Serialize                     301      24942   0.0    0.0     0.0    0.0
 putListOf                      Data.Serialize.Put                 302      24942   0.0    0.0     0.0    0.0
  encodeListOf                  Data.Serialize.Put                 305      24942   0.0    0.0     0.0    0.0
appendMsg                       CMQ                                287      24349   1.9    1.3    41.9   73.7
 appendMsg.env                  CMQ                                448      24348   0.1    0.0     0.4    0.1
  getQthresh                    CMQ                                450          0   0.3    0.1     0.3    0.1
 appendMsg.(...)                CMQ                                348      24349   0.2    0.0     0.2    0.0
 appendMsg.messages'            CMQ                                347      24349   0.0    0.0     0.0    0.0
 appendMsg.l                    CMQ                                289      24349   0.1    0.0    36.8   48.2
  encode                        Data.Serialize                     290      24349   0.1    0.0    36.7   48.2
```

```
put                           Data.Serialize                303          0   0.6   0.0   17.6  12.0
 putListOf                    Data.Serialize.Put            304          0   0.8   0.0   17.0  12.0
  tell                        Data.Serialize.Put            310     535565   0.2   0.4    0.2   0.4
  encodeListOf                Data.Serialize.Put            306          0   0.5   0.0   16.1  11.7
   encodeListOf.\             Data.Serialize.Put            307     535565   7.2   6.1   15.6  11.7
    put                       Data.Serialize                353    2044861   1.5   0.9    5.1   4.6
     putWord8                 Data.Serialize.Put            357          0   1.1   0.0    3.6   3.7
      tell                    Data.Serialize.Put            361    2044861   0.5   1.4    0.5   1.4
      singleton               Data.Serialize.Builder        358    2044861   0.3   0.0    2.0   2.3
       writeN                 Data.Serialize.Builder        359    2044861   1.3   2.3    1.7   2.3
        unsafeLiftIO          Data.Serialize.Builder        374    2044861   0.0   0.0    0.0   0.0
        append                Data.Serialize.Builder        360    2044861   0.4   0.0    0.4   0.0
     put.c                    Data.Serialize                354    2044861   0.0   0.0    0.0   0.0
    execPut                   Data.Serialize.Put            320    3091642   1.2   0.0    1.2   0.0
     sndS                     Data.Serialize.Put            321    3091642   0.0   0.0    0.0   0.0
    putWord64be               Data.Serialize.Put            315     535565   0.1   0.0    1.0   1.0
     tell                     Data.Serialize.Put            319     535565   0.2   0.4    0.2   0.4
     putWord64be              Data.Serialize.Builder        316     535565   0.1   0.0    0.7   0.6
      writeN                  Data.Serialize.Builder        317     535565   0.5   0.6    0.6   0.6
       unsafeLiftIO           Data.Serialize.Builder        340     535565   0.0   0.0    0.0   0.0
       append                 Data.Serialize.Builder        318     535565   0.1   0.0    0.1   0.0
    append                    Data.Serialize.Builder        309    3091642   1.1   0.0    1.1   0.0
    mappend                   Data.Serialize.Builder        308     535565   0.0   0.0    0.0   0.0
runPut                        Data.Serialize.Put            291      24349   0.0   0.0   19.1  36.1
 sndS                         Data.Serialize.Put            300      24349   0.0   0.0    0.0   0.0
 toByteString                 Data.Serialize.Builder        292      24349   0.3   0.0   19.1  36.1
  append                      Data.Serialize.Builder        299      24349   7.1   5.1   18.6  13.5
   put                        Data.Serialize                311          0   0.0   0.0   11.6   8.3
    putListOf                 Data.Serialize.Put            312          0   0.0   0.0   11.6   8.3
```

```
encodeListOf            Data.Serialize.Put                       313           0    0.0    0.0    11.6    8.3
 encodeListOf.\         Data.Serialize.Put                       314           0    0.0    0.0    11.6    8.3
  put                   Data.Serialize                           362           0    0.0    0.0     6.8    6.2
   putWord8             Data.Serialize.Put                       363           0    0.0    0.0     6.8    6.2
    singleton           Data.Serialize.Builder                   364           0    0.0    0.0     6.8    6.2
     writeN             Data.Serialize.Builder                   365           0    0.3    0.0     6.8    6.2
      ensureFree        Data.Serialize.Builder                   379           0    0.2    0.0     2.3    0.0
       withSize         Data.Serialize.Builder                   380           0    0.5    0.0     2.1    0.0
        withSize.\      Data.Serialize.Builder                   381     1533645    1.5    0.0     1.7    0.0
         ensureFree.\   Data.Serialize.Builder                   382     1533645    0.2    0.0     0.2    0.0
      unsafeLiftIO      Data.Serialize.Builder                   375           0    0.5    0.0     4.2    6.2
       unsafeLiftIO.\   Data.Serialize.Builder                   376     2044861    1.9    0.0     3.7    6.2
        flush.\         Data.Serialize.Builder                   386       24348    0.0    0.1     0.0    0.1
        writeNBuffer    Data.Serialize.Builder                   377     2044861    1.6    6.0     1.8    6.0
         writeNBuffer.\ Data.Serialize.Builder                   378     2044860    0.2    0.0     0.2    0.0
  putWord64be           Data.Serialize.Put                       322           0    0.0    0.0     4.7    2.2
   putWord64be          Data.Serialize.Builder                   323           0    0.0    0.0     4.7    2.2
    writeN              Data.Serialize.Builder                   324           0    0.1    0.0     4.7    2.2
     unsafeLiftIO       Data.Serialize.Builder                   341           0    0.2    0.0     2.9    2.2
      unsafeLiftIO.\    Data.Serialize.Builder                   342      535565    0.5    0.0     2.7    2.2
       putWord8         Data.Serialize.Put                       370           0    0.0    0.0     0.0    0.0
        singleton       Data.Serialize.Builder                   371           0    0.0    0.0     0.0    0.0
       writeNBuffer     Data.Serialize.Builder                   343      535565    0.3    1.6     2.2    2.2
        writeNBuffer.\  Data.Serialize.Builder                   344      535565    0.3    0.6     1.8    0.6
         putWord64be.\  Data.Serialize.Builder                   345      535565    1.5    0.0     1.6    0.0
          shiftr_w64    Data.Serialize.Builder                   346     3748955    0.1    0.0     0.1    0.0
     ensureFree         Data.Serialize.Builder                   334           0    0.3    0.0     1.8    0.0
      withSize          Data.Serialize.Builder                   335           0    0.3    0.0     1.5    0.0
       withSize.\       Data.Serialize.Builder                   336     1046781    0.9    0.0     1.2    0.0
```

```
               ensureFree.\    Data.Serialize.Builder                   337     1046781    0.3    0.0     0.3    0.0
    newBuffer                  Data.Serialize.Builder                   295           0    0.2   22.6     0.2   22.6
appendMsg.l’                   CMQ                                      288       24349    0.1    0.1     2.6   24.0
 encode                        Data.Serialize                           387       24348    0.1    0.0     2.5   24.0
  put                          Data.Serialize                           393           0    0.0    0.0     1.1    0.5
   putListOf                   Data.Serialize.Put                       394           0    0.0    0.0     1.1    0.5
    tell                       Data.Serialize.Put                       399       24348    0.0    0.0     0.0    0.0
    encodeListOf               Data.Serialize.Put                       395           0    0.0    0.0     1.0    0.5
     encodeListOf.\            Data.Serialize.Put                       396       24348    0.5    0.2     1.0    0.5
      put                      Data.Serialize                           425       97392    0.1    0.0     0.4    0.2
       putWord8                Data.Serialize.Put                       427           0    0.1    0.0     0.3    0.2
        tell                   Data.Serialize.Put                       431       97392    0.0    0.1     0.0    0.1
        singleton              Data.Serialize.Builder                   428       97392    0.0    0.0     0.1    0.1
         writeN                Data.Serialize.Builder                   429       97392    0.1    0.1     0.1    0.1
          unsafeLiftIO         Data.Serialize.Builder                   438       97392    0.0    0.0     0.0    0.0
          append               Data.Serialize.Builder                   430       97392    0.0    0.0     0.0    0.0
       put.c                   Data.Serialize                           426       97392    0.0    0.0     0.0    0.0
      execPut                  Data.Serialize.Put                       409      121740    0.0    0.0     0.0    0.0
       sndS                    Data.Serialize.Put                       410      121740    0.0    0.0     0.0    0.0
      putWord64be              Data.Serialize.Put                       404       24348    0.0    0.0     0.1    0.0
       tell                    Data.Serialize.Put                       408       24348    0.0    0.0     0.0    0.0
       putWord64be             Data.Serialize.Builder                   405       24348    0.0    0.0     0.1    0.0
        writeN                 Data.Serialize.Builder                   406       24348    0.0    0.0     0.0    0.0
         unsafeLiftIO          Data.Serialize.Builder                   418       24348    0.0    0.0     0.0    0.0
         append                Data.Serialize.Builder                   407       24348    0.0    0.0     0.0    0.0
      append                   Data.Serialize.Builder                   398      121740    0.0    0.0     0.0    0.0
      mappend                  Data.Serialize.Builder                   397       24348    0.0    0.0     0.0    0.0
  runPut                       Data.Serialize.Put                       388       24348    0.0    0.0     1.3   23.4
   sndS                        Data.Serialize.Put                       392       24348    0.0    0.0     0.0    0.0
```

```
toByteString                Data.Serialize.Builder            389       24348    0.2    0.0     1.2   23.4
 append                     Data.Serialize.Builder            391       24348    0.4    0.2     0.9    0.8
  put                       Data.Serialize                    400           0    0.0    0.0     0.5    0.5
   putListOf                Data.Serialize.Put                401           0    0.0    0.0     0.5    0.5
    encodeListOf            Data.Serialize.Put                402           0    0.0    0.0     0.5    0.5
     encodeListOf.\         Data.Serialize.Put                403           0    0.0    0.0     0.5    0.5
      put                   Data.Serialize                    432           0    0.0    0.0     0.3    0.4
       putWord8             Data.Serialize.Put                433           0    0.0    0.0     0.3    0.4
        singleton           Data.Serialize.Builder            434           0    0.0    0.0     0.3    0.4
         writeN             Data.Serialize.Builder            435           0    0.0    0.0     0.3    0.4
          ensureFree        Data.Serialize.Builder            443           0    0.0    0.0     0.1    0.0
           withSize         Data.Serialize.Builder            444           0    0.0    0.0     0.1    0.0
            withSize.\      Data.Serialize.Builder            445       73044    0.0    0.0     0.1    0.0
             ensureFree.\   Data.Serialize.Builder            446       73044    0.0    0.0     0.0    0.0
          unsafeLiftIO      Data.Serialize.Builder            439           0    0.0    0.0     0.1    0.4
           unsafeLiftIO.\   Data.Serialize.Builder            440       97392    0.1    0.0     0.1    0.4
            flush.\         Data.Serialize.Builder            447       24348    0.0    0.1     0.0    0.1
            writeNBuffer    Data.Serialize.Builder            441       97392    0.0    0.3     0.0    0.3
             writeNBuffer.\ Data.Serialize.Builder            442       97392    0.0    0.0     0.0    0.0
      putWord64be           Data.Serialize.Put                411           0    0.0    0.0     0.2    0.1
       putWord64be          Data.Serialize.Builder            412           0    0.0    0.0     0.2    0.1
        writeN              Data.Serialize.Builder            413           0    0.0    0.0     0.2    0.1
         unsafeLiftIO       Data.Serialize.Builder            419           0    0.0    0.0     0.2    0.1
          unsafeLiftIO.\    Data.Serialize.Builder            420       24348    0.0    0.0     0.1    0.1
           putWord8         Data.Serialize.Put                436           0    0.0    0.0     0.0    0.0
            singleton       Data.Serialize.Builder            437           0    0.0    0.0     0.0    0.0
           writeNBuffer     Data.Serialize.Builder            421       24348    0.0    0.1     0.1    0.1
            writeNBuffer.\  Data.Serialize.Builder            422       24348    0.0    0.0     0.1    0.0
             putWord64be.\  Data.Serialize.Builder            423       24348    0.1    0.0     0.1    0.0
```

```
                     shiftr_w64         Data.Serialize.Builder      424   170436   0.0   0.0   0.0   0.0
                ensureFree              Data.Serialize.Builder      414        0   0.0   0.0   0.1   0.0
                 withSize               Data.Serialize.Builder      415        0   0.0   0.0   0.1   0.0
                  withSize.\            Data.Serialize.Builder      416    48696   0.1   0.0   0.1   0.0
                   ensureFree.\         Data.Serialize.Builder      417    48696   0.0   0.0   0.0   0.0
        newBuffer                       Data.Serialize.Builder      390        0   0.1  22.6   0.1  22.6
   cwPush.m                             CMQ                         284    24350   0.0   0.0   0.3   0.1
    getTMap                             CMQ                         286        0   0.3   0.1   0.3   0.1
   insertSglton                         CMQ                         283        1   0.0   0.0   0.0   0.0
   cwPush.q                             CMQ                         280    24350   0.1   0.0   0.3   0.1
    getTPsq                             CMQ                         282        0   0.2   0.1   0.2   0.1
collectPong                             Main                        267        0  32.7  15.8  47.7  15.9
 inc                                    Main                        594    10003   0.0   0.0   0.0   0.0
 cwPop                                  CMQ                         269        0  15.1   0.1  15.1   0.1
newRq                                   CMQ                         259        0   0.0   0.0   0.0   1.2
 get                                    Data.Serialize              527        0   0.0   0.0   0.0   1.1
  getListOf                             Data.Serialize.Get          528        0   0.0   0.0   0.0   1.1
   getListOf.go                         Data.Serialize.Get          547    61009   0.0   0.1   0.0   0.1
    return                              Data.Serialize.Get          588    10374   0.0   0.0   0.0   0.0
    >>=                                 Data.Serialize.Get          548    50635   0.0   0.0   0.0   0.0
   >>=                                  Data.Serialize.Get          529    91390   0.0   0.0   0.0   1.0
    >>=.\                               Data.Serialize.Get          530   335179   0.0   0.2   0.0   1.0
     getWord8                           Data.Serialize.Get          564        0   0.0   0.0   0.0   0.6
      getPtr                            Data.Serialize.Get          565        0   0.0   0.0   0.0   0.6
       getBytes                         Data.Serialize.Get          571        0   0.0   0.0   0.0   0.6
        ensure                          Data.Serialize.Get          573        0   0.0   0.0   0.0   0.6
         ensure.\                       Data.Serialize.Get          574    40508   0.0   0.0   0.0   0.6
          >>=.\.ks'                     Data.Serialize.Get          575   293683   0.0   0.4   0.0   0.6
           getBytes.rest                Data.Serialize.Get          587    40261   0.0   0.0   0.0   0.0
      getListOf.go           Data.Serialize.Get            586          0   0.0   0.0    0.0   0.0
       return                Data.Serialize.Get            589          0   0.0   0.0    0.0   0.0
        return.\             Data.Serialize.Get            590      10374   0.0   0.0    0.0   0.0
         finalK              Data.Serialize.Get            591        247   0.0   0.0    0.0   0.0
      getPtr.k               Data.Serialize.Get            584      40508   0.0   0.0    0.0   0.0
      getBytes.consume       Data.Serialize.Get            583      40508   0.0   0.0    0.0   0.0
      return                 Data.Serialize.Get            578     202540   0.0   0.0    0.0   0.1
       return.\              Data.Serialize.Get            579     202540   0.0   0.0    0.0   0.1
        get                  Data.Serialize                585          0   0.0   0.0    0.0   0.0
        fmap                 Data.Serialize.Get            580          0   0.0   0.0    0.0   0.1
         fmap.\              Data.Serialize.Get            581          0   0.0   0.0    0.0   0.1
          fmap.\.ks'         Data.Serialize.Get            582      40508   0.0   0.1    0.0   0.1
      put                    Data.Serialize.Get            576      40508   0.0   0.0    0.0   0.0
       put.\                 Data.Serialize.Get            577      40508   0.0   0.0    0.0   0.0
  fmap                       Data.Serialize.Get            567          0   0.0   0.0    0.0   0.0
   fmap.\                    Data.Serialize.Get            568      40508   0.0   0.0    0.0   0.0
get                          Data.Serialize                556          0   0.0   0.0    0.0   0.0
 get.getByte                 Data.Serialize                559          0   0.0   0.0    0.0   0.0
getWord64be                  Data.Serialize.Get            533          0   0.0   0.0    0.0   0.1
 getBytes                    Data.Serialize.Get            536          0   0.0   0.0    0.0   0.1
  ensure                     Data.Serialize.Get            538          0   0.0   0.0    0.0   0.1
   ensure.\                  Data.Serialize.Get            539      10374   0.0   0.0    0.0   0.1
    >>=.\.ks'                Data.Serialize.Get            540      41496   0.0   0.0    0.0   0.1
     getBytes.rest           Data.Serialize.Get            553      10374   0.0   0.0    0.0   0.0
     getListOf.go            Data.Serialize.Get            549          0   0.0   0.0    0.0   0.0
     getBytes.consume        Data.Serialize.Get            546      10374   0.0   0.0    0.0   0.0
     shiftl_w64              Data.Serialize.Get            545      72618   0.0   0.0    0.0   0.0
     return                  Data.Serialize.Get            543      20748   0.0   0.0    0.0   0.0
      return.\               Data.Serialize.Get            544      20748   0.0   0.0    0.0   0.0
```

```
            put                     Data.Serialize.Get       541   10374  0.0  0.0  0.0  0.0
             put.\                  Data.Serialize.Get       542   10374  0.0  0.0  0.0  0.0
   loadTChan                        CMQ                      263       0  0.0  0.0  0.0  0.1
    write2TChan                     CMQ                      592     246  0.0  0.0  0.0  0.1
     write2TChan.\                  CMQ                      593   10086  0.0  0.0  0.0  0.0
    receiveMessage                  CMQ                      265       0  0.0  0.0  0.0  0.0
     decode                         Data.Serialize           522     247  0.0  0.0  0.0  0.0
      runGet                        Data.Serialize.Get       523     247  0.0  0.0  0.0  0.0
   loopMyQ                          CMQ                      261       0  0.0  0.0  0.0  0.0
    loopMyQ.tdelay                  CMQ                      519     248  0.0  0.0  0.0  0.0
     getDelay                       CMQ                      521       0  0.0  0.0  0.0  0.0
    loopMyQ.duetime                 CMQ                      518     248  0.0  0.0  0.0  0.0
CAF                                 CMQ                      245       0  0.0  0.0  0.0  0.0
 getDelay                           CMQ                      520       1  0.0  0.0  0.0  0.0
 getQthresh                         CMQ                      449       1  0.0  0.0  0.0  0.0
 getTMap                            CMQ                      285       1  0.0  0.0  0.0  0.0
 getTPsq                            CMQ                      281       1  0.0  0.0  0.0  0.0
 getTChan                           CMQ                      271       1  0.0  0.0  0.0  0.0
CAF                                 Main                     244       0  0.0  0.0  5.7  6.7
 get                                Data.Serialize           550       1  0.0  0.0  0.0  0.0
  getListOf                         Data.Serialize.Get       551       1  0.0  0.0  0.0  0.0
   >>=                              Data.Serialize.Get       552       1  0.0  0.0  0.0  0.0
 put                                Data.Serialize           350       1  0.0  0.0  0.0  0.0
  putListOf                         Data.Serialize.Put       351       1  0.0  0.0  0.0  0.0
   encodeListOf                     Data.Serialize.Put       352       1  0.0  0.0  0.0  0.0
 $cStrlen                           Main                     252       1  0.0  0.0  0.0  0.0
 $tStrlen                           Main                     249       1  0.0  0.0  0.0  0.0
 strlen                             Main                     247       1  0.0  0.0  0.0  0.0
  gunfold                           Main                     253       1  0.0  0.0  0.0  0.0
  toConstr                          Main                                  251      4  0.0  0.0  0.0  0.0
  gfoldl                            Main                                  250      1  0.0  0.0  0.0  0.0
  dataTypeOf                        Main                                  248      1  0.0  0.0  0.0  0.0
 main                               Main                                  246      1  0.0  0.0  5.7  6.7
  main.diff                         Main                                  595      1  0.0  0.0  0.0  0.0
  runTest                           Main                                  255      1  0.0  0.0  5.7  6.7
   sendPing                         Main                                  274      1  0.2  0.1  3.5  3.7
    sendPing.\                      Main                                  276  24350  3.2  3.6  3.3  3.6
     cwPush                         CMQ                                   277  24350  0.0  0.0  0.0  0.0
   collectPong                      Main                                  266      1  2.2  3.0  2.2  3.0
    cwPop                           CMQ                                   268      1  0.0  0.0  0.0  0.0
     cwPop.mtch                     CMQ                                   270      1  0.0  0.0  0.0  0.0
      getTChan                      CMQ                                   272      0  0.0  0.0  0.0  0.0
   newRq                            CMQ                                   256      1  0.0  0.0  0.0  0.0
    get                             Data.Serialize                        524      1  0.0  0.0  0.0  0.0
     getListOf                      Data.Serialize.Get                    525      1  0.0  0.0  0.0  0.0
      >>=                           Data.Serialize.Get                    526      1  0.0  0.0  0.0  0.0
    newRq.cmq                       CMQ                                   273      1  0.0  0.0  0.0  0.0
    loadTChan                       CMQ                                   262      1  0.0  0.0  0.0  0.0
     receiveMessage                 CMQ                                   264      1  0.0  0.0  0.0  0.0
    loopMyQ                         CMQ                                   260      1  0.0  0.0  0.0  0.0
  strlen                            Main                                  254      0  0.0  0.0  0.0  0.0
CAF                                 System.Console.CmdArgs.Default        243      0  0.0  0.0  0.0  0.0
CAF                                 System.Console.CmdArgs.Implicit.Global 237     0  0.0  0.0  0.0  0.0
CAF                                 System.Console.CmdArgs.Implicit.Reader 235     0  0.0  0.0  0.0  0.0
CAF                                 System.Console.CmdArgs.Annotate       231      0  0.0  0.0  0.0  0.0
CAF                                 System.Console.CmdArgs.Explicit       230      0  0.0  0.0  0.0  0.0
CAF                                 Data.Generics.Any.Prelude             227      0  0.0  0.0  0.0  0.0
CAF                                 System.Console.CmdArgs.Explicit.Help  224      0  0.0  0.0  0.0  0.0
CAF                                  Data.IP.Addr                        216            0    0.0    0.0     0.0    0.0
CAF                                  Network.Socket                      215            0    0.0    0.0     0.0    0.0
CAF                                  Data.Serialize                      195            0    0.0    0.0     0.0    0.0
 get                                 Data.Serialize                      560            1    0.0    0.0     0.0    0.0
 get                                 Data.Serialize                      554            1    0.0    0.0     0.0    0.0
  get.getByte                        Data.Serialize                      557            1    0.0    0.0     0.0    0.0
   >>=                               Data.Serialize.Get                  558            1    0.0    0.0     0.0    0.0
  >>=                                Data.Serialize.Get                  555            1    0.0    0.0     0.0    0.0
 put                                 Data.Serialize                      355            1    0.0    0.0     0.0    0.0
 put                                 Data.Serialize                      325            0    0.0    0.0     0.0    0.0
  putListOf                          Data.Serialize.Put                  326            0    0.0    0.0     0.0    0.0
   encodeListOf                      Data.Serialize.Put                  327            0    0.0    0.0     0.0    0.0
    encodeListOf.\                   Data.Serialize.Put                  328            0    0.0    0.0     0.0    0.0
     mempty                          Data.Serialize.Builder              383            1    0.0    0.0     0.0    0.0
      empty                          Data.Serialize.Builder              384            1    0.0    0.0     0.0    0.0
     mappend                         Data.Serialize.Builder              349            1    0.0    0.0     0.0    0.0
     putWord64be                     Data.Serialize.Put                  329            0    0.0    0.0     0.0    0.0
      putWord64be                    Data.Serialize.Builder              330            0    0.0    0.0     0.0    0.0
       writeN                        Data.Serialize.Builder              331            0    0.0    0.0     0.0    0.0
        ensureFree                   Data.Serialize.Builder              332            1    0.0    0.0     0.0    0.0
         ensureFree.\                Data.Serialize.Builder              338            0    0.0    0.0     0.0    0.0
          empty                      Data.Serialize.Builder              339            1    0.0    0.0     0.0    0.0
         withSize                    Data.Serialize.Builder              333            1    0.0    0.0     0.0    0.0
CAF                                  Data.Serialize.Put                  194            0    0.0    0.0     0.0    0.0
 putWord8                            Data.Serialize.Put                  356            1    0.0    0.0     0.0    0.0
  singleton                          Data.Serialize.Builder              366            0    0.0    0.0     0.0    0.0
   writeN                            Data.Serialize.Builder              367            0    0.0    0.0     0.0    0.0
    ensureFree                       Data.Serialize.Builder              368            1    0.0    0.0     0.0    0.0
     ensureFree.\                    Data.Serialize.Builder              372            0    0.0    0.0     0.0    0.0
```

```
      empty                          Data.Serialize.Builder                   373           1    0.0    0.0     0.0    0.0
     withSize                        Data.Serialize.Builder                   369           1    0.0    0.0     0.0    0.0
CAF                                  Data.Serialize.Get                       193           0    0.0    0.0     0.0    0.0
 getWord8                            Data.Serialize.Get                       561           1    0.0    0.0     0.0    0.0
  getPtr                             Data.Serialize.Get                       562           1    0.0    0.0     0.0    0.0
   getBytes                          Data.Serialize.Get                       569           1    0.0    0.0     0.0    0.0
    ensure                           Data.Serialize.Get                       572           1    0.0    0.0     0.0    0.0
    >>=                              Data.Serialize.Get                       570           1    0.0    0.0     0.0    0.0
   fmap                              Data.Serialize.Get                       566           1    0.0    0.0     0.0    0.0
   >>=                               Data.Serialize.Get                       563           1    0.0    0.0     0.0    0.0
 getWord64be                         Data.Serialize.Get                       531           1    0.0    0.0     0.0    0.0
  getBytes                           Data.Serialize.Get                       534           1    0.0    0.0     0.0    0.0
   ensure                            Data.Serialize.Get                       537           1    0.0    0.0     0.0    0.0
   >>=                               Data.Serialize.Get                       535           1    0.0    0.0     0.0    0.0
  >>=                                Data.Serialize.Get                       532           1    0.0    0.0     0.0    0.0
CAF                                  Data.Serialize.Builder                   192           0    0.0    0.0     0.0    0.0
 flush                               Data.Serialize.Builder                   385           1    0.0    0.0     0.0    0.0
 defaultSize                         Data.Serialize.Builder                   296           1    0.0    0.0     0.0    0.0
  defaultSize.overhead               Data.Serialize.Builder                   298           1    0.0    0.0     0.0    0.0
  defaultSize.k                      Data.Serialize.Builder                   297           1    0.0    0.0     0.0    0.0
 toByteString                        Data.Serialize.Builder                   293           0    0.0    0.0     0.0    0.0
  newBuffer                          Data.Serialize.Builder                   294           1    0.0    0.0     0.0    0.0
CAF                                  GHC.IO.Encoding                          176           0    0.0    0.0     0.0    0.0
CAF                                  GHC.IO.Handle.FD                         173           0    0.0    0.0     0.0    0.0
CAF                                  Text.Printf                              165           0    0.0    0.0     0.0    0.0
CAF                                  GHC.Event.Internal                       163           0    0.0    0.0     0.0    0.0
CAF                                  GHC.Event.Thread                         162           0    0.0    0.0     0.0    0.0
CAF                                  GHC.Conc.Sync                            160           0    0.0    0.0     0.0    0.0
CAF                                  Data.Typeable.Internal                   155           0    0.0    0.0     0.0    0.0
CAF          System.CPUTime                        154      0   0.0   0.0    0.0   0.0
CAF          GHC.Conc.Signal                       149      0   0.0   0.0    0.0   0.0
CAF          GHC.Float                             147      0   0.0   0.0    0.0   0.0
CAF          GHC.IO.Encoding.Iconv                 146      0   0.0   0.0    0.0   0.0
CAF          GHC.Integer.Logarithms.Internals      131      0   0.0   0.0    0.0   0.0
```

Bibliography

[ABLG10] Arnold Aumasson, Vincent Bonneau, Timo Leimbach, and Moritz Gödel. Economic and social impact of software & software-based services. *Cordis (Online), BE: European Commission. Available online: http://cordis. europa. eu/fp7/ict/ssai/docs/study-sw-report-final. pdf*, 2010.

[AFG$^+$10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.

[Agh86] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems (Mit Press Series in Artificial Intelligence)*. The MIT Press, Cambridge, Massachusetts, 1986.

[Ano11] Personal Data: The Emergence of a New Asset Class. *World Economic Forum*, pages 1–40, January 2011.

[apaa] Avro. `http://avro.apache.org/`. Online; accessed 26 September 2015.

[apab] hadoop. `http://hadoop.apache.org/`. Online, accessed 26 September 2015.

[Atk08] Alistair Atkinson. Tupleware: A distributed tuple space for cluster computing. In *Parallel and Distributed Computing, Applications and Technologies, 2008. PDCAT 2008. Ninth International Conference on*, pages 121–126. IEEE, 2008.

[AVW93] Joe Armstrong, Robert Virding, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, March 1993.

[BCC12] A Bialecki, M Cafarella, and D Cutting. Hadoop: a framework for running applications on large clusters built of commodity hardware. Technical report, 2012.

[BCvR09] Ken Birman, Gregory Chockler, and Robbert van Renesse. Toward a cloud computing research agenda. *SIGACT News*, 40(2):68–80, 2009.

[BDH03] LA Barroso, J Dean, and U Holzle. Web search for a planet: The Google cluster architecture. *IEEE Micro*, 23(2):22–28, March 2003.

[BJDM97] R. Bird, G. Jones, and O. De Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(05):541–547, 1997.

[BK86] JA Bergstra and Jan Willem Klop. Algebra of communicating processes. *CWI Monograph series*, 3:89–138, 1986.

[BMNZ14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. In *Big Data and Internet of Things: A Roadmap for Smart Environments*, pages 169–186. Springer, 2014.

[BR02] Roberto Baldoni and Michel Raynal. Fundamentals of distributed computing: A practical tour of vector clock systems. *IEEE Distributed Systems Online*, 3(2):12, 2002.

[Bro11] Neil Christopher Charles Brown. *Communicating Haskell Processes*. PhD thesis, Citeseer, 2011.

[BS99] Lawrie Brown and Dan Sahlin. Extending erlang for safe mobile code execution. In *International Conference on Information and Communications Security*, pages 39–53. Springer, 1999.

[Cap97] DKG Capmbell. Implementing algorithmic skeletons for generative communication with linda. *REPORT-UNIVERSITY OF YORK DEPARTMENT OF COMPUTER SCIENCE YCS*, 1997.

[Car05] N G Carr. The end of corporate computing. *MIT Sloan Management Review*, 46(3):67–73, 2005.

[Car09] Nicholas Carr. *The Big Switch: Rewiring the World, from Edison to Google*. W. W. Norton & Company, reprint edition, January 2009.

[CBP$^+$10] N Chohan, C Bunch, S Pang, C Krintz, N Mostafa, S Soman, and R Wolski. Appscale: Scalable and open appengine application development and deployment. *Cloud Computing*, pages 57–70, 2010.

[CCDN$^+$12] Nicolo M Calcavecchia, Bogdan A Caprarescu, Elisabetta Di Nitto, Daniel J Dubois, and Dana Petcu. Depas: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94(8-10):701–730, 2012.

[CGW91] WJ Cullyer, SJ Goodenough, and BA Wichmann. The choice of computer languages for use in safety-critical systems. *Software Engineering Journal*, 6(2):51–58, 1991.

[Cha13] Thibaud Chardonnens. Big data analytics on high velocity streams. 2013.

[CJLL02] FH Carvalho Jr, Ricardo Massa Ferreira Lima, and Rafael Dueire Lins. Coordinating functional processes with haskell#. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 393–400. ACM, 2002.

[Cla10] Tim Clark. Quantifying the benefits of the rightscale cloud management platform. *Rightscale [en línea] Available at: http://www. rightscale. com/info_center/white-papers. php (Accessed August 12, 2013)*, 2010.

[clu] Clustrx. `http://massivesolutions.co.uk/clustrx.html`. Online; accessed 14 February 2013.

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel computing*, 30(3):389–406, 2004.

[Dew25] John Dewey. Logic: The theory of inquiry (1938). *The later works*, 1953:1–549, 1925.

[DG08a] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DG08b] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[DHJ$^+$07a] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SOSP*, volume 7, pages 205–220, 2007.

[DHJ$^+$07b] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 205–220. ACM, 2007.

[EBPJ11] Jeff Epstein, Andrew P Black, and Simon Peyton-Jones. Towards Haskell in the cloud. In *Proceedings of the 4th ACM symposium on Haskell*, pages 118–129, New York, NY, USA, 2011. ACM.

[etc15] etcd – A high-available key value store for shared configuration and service discovery. `https://coreos.com/etcd/`, 2015. Online; accessed Sep 26, 2015.

[FBB$^+$05] W Feng, P Balaji, C Baron, L N Bhuyan, and D K Panda. Performance characterization of a 10-Gigabit Ethernet TOE. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, pages 58–63, August 2005.

[Fie09] Glenn Fiedler. Reliability and Flow Control. `http://gafferongames.com/networking-for-game-programmers/reliability-and-flow-control/`, 2009. accessed May 2012.

[Frü95] Thom Frühwirth. Constraint handling rules. In *Constraint programming: Basics and trends*, pages 90–107. Springer, 1995.

[FTS$^+$] M. J. Frisch, G. W. Trucks, H. B. Schlegel, G. E. Scuseria, M. A. Robb, J. R. Cheeseman, J. A. Montgomery, Jr., T. Vreven, K. N. Kudin, J. C. Burant, J. M. Millam, S. S. Iyengar, J. Tomasi, V. Barone, B. Mennucci, M. Cossi, G. Scalmani, N. Rega, G. A. Petersson, H. Nakatsuji, M. Hada, M. Ehara, K. Toyota, R. Fukuda, J. Hasegawa, M. Ishida, T. Nakajima, Y. Honda, O. Kitao, H. Nakai, M. Klene, X. Li, J. E. Knox, H. P. Hratchian, J. B. Cross, V. Bakken, C. Adamo, J. Jaramillo, R. Gomperts, R. E. Stratmann, O. Yazyev, A. J. Austin, R. Cammi, C. Pomelli, J. W. Ochterski, P. Y. Ayala, K. Morokuma, G. A. Voth, P. Salvador, J. J. Dannenberg, V. G. Zakrzewski, S. Dapprich, A. D. Daniels, M. C. Strain, O. Farkas, D. K. Malick, A. D. Rabuck, K. Raghavachari, J. B. Foresman, J. V. Ortiz, Q. Cui, A. G. Baboul, S. Clifford, J. Cioslowski, B. B. Stefanov, G. Liu, A. Liashenko, P. Piskorz, I. Komaromi, R. L. Martin, D. J. Fox, T. Keith, M. A. Al-Laham, C. Y. Peng, A. Nanayakkara, M. Challacombe, P. M. W. Gill, B. Johnson, W. Chen, M. W. Wong, C. Gonzalez, and J. A. Pople. Gaussian 03, Revision C.02. Gaussian, Inc., Wallingford, CT, 2004.

[Fur] Sadayuki Furuhashi. Messagepack: It's like json. but fast and small, 2014. *`http://msgpack.org`*.

[FW12a] J Fritsch and C Walker. CMQ-A lightweight, asynchronous high-performance messaging queue for the cloud. *Journal of Cloud ...*, 2012.

[FW12b] Joerg Fritsch and Coral Walker. Cmq-a lightweight, asynchronous high-performance messaging queue for the cloud. *Journal of Cloud Computing*, 1(1):1–13, 2012.

[FW14a] Joerg Fritsch and Coral Walker. Cwmwl, a linda-based paas fabric for the cloud. *Journal of Communications*, 9(4):286–298, 2014.

[FW14b] Joerg Fritsch and Coral Walker. The problem with data. In *7th International Conference on Utility and Cloud Computing (UCC 2014)*. IEEE/ACM, 2014.

[FWR$^+$05] Daniel Fiedler, Kristen Walcott, Thomas Richardson, Gregory M Kapfhammer, Ahmed Amer, and Panos K Chrysanthis. Towards the measurement of tuple space performance. *ACM SIGMETRICS Performance Evaluation Review*, 33(3):51–62, 2005.

[FZRL08] I Foster, Yong Zhao, I Raicu, and S Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, November 2008.

[GC92] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):96–, February 1992.

[Gel85] David Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):80–112, 1985.

[GG07] Y. Gu and R.L. Grossman. UDT: UDP-based data transfer for high-speed wide area networks. *Computer Networks*, 51(7):1777–1799, 2007.

[GHW12a] Kartik Gopalan, Michael Hines, and Jian Wang. Centralized adaptive network memory engine, October 16 2012. US Patent 8,291,034.

[GHW12b] Kartik Gopalan, Michael Hines, and Jian Wang. Distributed adaptive network memory engine, October 2 2012. US Patent 8,280,976.

[Gig] Gigaspaces XAP. `http://www.gigaspaces.com`. Online; accessed 26 September 2015.

[Gil74] KAHN Gilles. The semantics of a simple language for parallel programming. *In Information Processing*, 74:471–475, 1974.

[goo] google.com. Google app engine - google developers.

[hal15] The haskell lightweight virtual machine (halvm): Ghc running on xen. `http://halvm.org`, 2015. Online; accessed 26 September 2015.

[Has12] HaskelWiki - Multicore. `https://wiki.haskell.org/Haskell_for_multicores#Examples_2`, 2012. Online; accessed 5 June 2015.

[Hay10] Jonathan Mark Hayman. *Petri net semantics*. PhD thesis, University of Cambridge, 2010.

[Hen04] M Henning. Massively multiplayer middleware. *Queue*, 1(10):38, 2004.

[Hen06] Cal Henderson. *Building scalable web sites*. " O'Reilly Media, Inc.", 2006.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[Hew10] Carl Hewitt. Actor Model of Computation: Scalable Robust Information Systems. *arXiv.org*, cs.PL, August 2010.

[Hin01] Ralf Hinze. A simple implementation technique for priority search queues. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, pages 110–121, New York, NY, USA, 2001. ACM.

[HKDS12] Franz J Hauck, Steffen Kächele, Jörg Domaschka, and Christian Spann. The cosca paas platform: on the way to flexible and dependable cloud computing. In *Proceedings of the 1st European Workshop on Dependable Cloud Computing*, page 1. ACM, 2012.

[HKJR10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, volume 8, page 9, 2010.

[HKZ$^+$11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[HLR$^+$13] Kirak Hong, David Lillethun, Umakishore Ramachandran, Beate Ottenwälder, and Boris Koldehofe. Mobile fog: A programming model for large-scale applications on the internet of things. In *Proceedings of the second ACM SIGCOMM workshop on Mobile cloud computing*, pages 15–20. ACM, 2013.

[HMJH08] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. *Commun. ACM*, 51(8):91–100, 2008.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60. ACM, 2005.

[How95] James Howatt. A project-based approach to programming language evaluation. *ACM SIGPLAn Notices*, 30(7):37–40, 1995.

[Huc99] F Huch. Erlang-style distributed haskell. *In Draft Proceedings of the 11th International Workshop on implementation of functional languages, September 7th 10th 1999*, 1999.

[IAB13] Price Water House Coopers IAB. Internet advertising revenue report 2013 first six months' results, 2013.

[IBY$^+$07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[Inca] Amazon Web Services Inc. `http://aws.amazon.com/lambda/`. Online; accessed 10 October 2015.

[Incb] Amazon Web Services Inc. High performance computing (hpc) on aws. `http://aws.amazon.com/hpc-applications/`. Online; accessed 20 April 2013.

[Incc] CoreOS Inc. Distributed reliable key-value store for the most critical data of a distributed system.

[Inc16] Google Inc. `https://cloud.google.com/functions/`, 2016. Online; accessed 23 April 2016.

[INT06] INTEL GmbH. Intel® MPI Benchmarks, June 2006.

[Int13] Intel. What happens in an internet minute? `http://www.intel.com/content/www/us/en/communications/internet-minute-infographic.html`, 2013. Online; accessed 26 September 2015.

[Iro16] Iron.io. `https://www.iron.io/platform/ironworker/`, 2016. Online; accessed 23 April 2016.

[JF14] Coral Walker Joerg Fritsch. Cwmwl, a linda-based paas fabric for the cloud. *Journal of Communications*, 9(4):286–298, 2014.

[JHM04] Wesley M Johnston, JR Hanna, and Richard J Millar. Advances in dataflow programming languages. *ACM Computing Surveys (CSUR)*, 36(1):1–34, 2004.

[JS04] Simon Peyton Jones and Mark Shields. Lexically scoped type variables. *Submitted to ICFP*, 2004.

[KDH11] Steffen Kächele, Jörg Domaschka, and Franz J Hauck. COSCA: an easy-to-use component-based PaaS cloud system for common applications. In *CloudCP '11: Proceedings of the First International Workshop on Cloud Computing Platforms*, page 4. ACM Request Permissions, April 2011.

[KH] NR Keetha and K He. HiPerFS: A Framework for High Performance Financial Services using Advanced Message Queuing. *cs.columbia.edu*.

[KHSS10] Ali Khajeh-Hosseini, Ian Sommerville, and Ilango Sriram. Research challenges for enterprise cloud computing. *arXiv preprint arXiv:1001.3257*, 2010.

[Kol84] David A Kolb. Experiential learning englewood cliffs. *NJ: Prentice-Hall. Lord, RG, & Emrich, CG (2001). Thinking outside the box by looking inside the box: Extending the cognitive revolution in leadership research. Leadership Quarterly*, 11(4):551–579, 1984.

[Kri10] Navaneeth Krishnan. Building Java Apps on the Google App Engine. `http://java-appengine.blogspot.com/2010/10/google-cloud-vs-amazon-cloud.html`, 2010. Online, accessed May 2012.

[KWW94] SC Kendall, J Waldo, and A Wollrath. *A Note on Distributed Computing, Sun Microsystems*. Inc., 1994.

[Läm08] Ralf Lämmel. Google's mapreduce programming model—revisited. *Science of computer programming*, 70(1):1–30, 2008.

[Lan01] Doug Laney. 3d data management: Controlling data volume. *Velocity, and Variety, Application Delivery Strategies published by META Group Inc*, 2001.

[Lee] Edward A Author Lee. The Problem with Threads. *Electrical Engineering and Computer Sciences University of California at Berkeley*.

[Lee06] Edward A Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

[Lee07] EA Lee. Are new languages necessary for multicore? *2007 International Symposium on Code Generation and Optimization*, 2007.

[LGL$^+$15] Tom H Luan, Longxiang Gao, Zhi Li, Yang Xiang, and Limin Sun. Fog computing: Focusing on mobile users at the edge. *arXiv preprint arXiv:1502.01815*, 2015.

[Lin14] Frank Van Lingen. Data driven platforms to support iot, sdn, and cloud. `http://blogs.cisco.com/perspectives/data-driven-platforms-to-support-iot-sdn-and-cloud/`, January 2014. Online; accessed 26 September 2015.

[Lip11] Miran Lipovača. *Learn You a Haskell for Great Good!* A Beginner's Guide. No Starch Pr, April 2011.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. 2001.

[Loo12] Rita Loogen. Eden–parallel functional programming with haskell. In *Central European Functional Programming School*, pages 142–206. Springer, 2012.

[Los10] Tim Lossen. Redis – memory as the new disk. In *NOSQL Europe Conference, April 20-22 2010.*, 2010.

[LTS+08] Jin Li, Kristin Tufte, Vladislav Shkapenyuk, Vassilis Papadimos, Theodore Johnson, and David Maier. Out-of-order processing: a new architecture for high-performance stream systems. *Proceedings of the VLDB Endowment*, 1(1):274–288, 2008.

[Mai07] Geoffrey Mainland. Why it's nice to be quoted: quasiquoting for haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, Haskell '07, pages 73–82, New York, NY, USA, 2007. ACM.

[Mar11] S Marlow. Parallel and concurrent programming in haskell. 2011.

[Mar12] Simon Marlow. Parallel and concurrent programming in haskell. In *Central European Functional Programming School*, pages 339–401. Springer, 2012.

[Mar13] Nathan Marz. *Big Data: Principles and best practices of scalable realtime data systems*. O'Reilly Media, 2013.

[mcc] Data gravity - in the clouds.

[MFD10] M. Marzolla, S. Ferretti, and G. D'Angelo. Dynamic scalability for next generation gaming infrastructures. *Proc. 4th ACM/ICST International Conference on Simulation Tools and Techniques (SIMUTools 2011)*, pages 1–8, 2010.

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[MH05] M Mernik and J Heering. When and how to develop domain-specific languages. *ACM Computing Surveys (CSUR)*, 2005.

[MH10] Derek G Murray and Steven Hand. Scripting the cloud with skywriting. In *HotCloud'10: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. USENIX Association, June 2010.

[MMI+13] Derek G Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martin Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[MMS$^+$10] A Madhavapeddy, R Mortier, R Sohan, T Gazagnaire, S Hand, T Deegan, D McAuley, and J Crowcroft. Turning down the LAMP: software specialisation for the cloud. 2010.

[MSMO97] M Mathis, J Semke, J Mahdavi, and T. Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[Mur11] DG Murray. Non-deterministic parallelism considered useful. *HotOS XIII*, 2011.

[Net] Networking for Game Programmers. `http://gafferongames.com/networking-for-game-programmers/`. Online; accessed May 2012.

[Nor11] John Norstad. A mapreduce algorithm for matrix multiplication. `http://www.norstad.org/matrix-multiply/`, 2011. Online; accessed 10 August 2013.

[NPFI09] V Nae, R Prodan, T Fahringer, and A Iosup. The impact of virtualization on the performance of Massively Multiplayer Online Games. *Network and Systems Support for Games (NetGames), 2009 8th Annual Workshop on*, pages 1–6, October 2009.

[NRNK10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177. IEEE, 2010.

[OGS08] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real world haskell: Code you can believe in.* " O'Reilly Media, Inc.", 2008.

[Oka99] Chris Okasaki. *Purely Functional Data Structures*. Cambridge Univ Pr, June 1999.

[OM09] Owen O'Malley and Arun Murthy. Hadoop sorts a petabyte in 16.25 hours and a terabyte in 62 seconds. `http://developer.yahoo.com/blogs/hadoop/hadoop-sorts-petabyte-16-25-hours-terabyte-62-422.html`, 2009. Online; accessed 26 September 2015.

[O'S15] Bryan O'Sullivan. The criterion package. `http://hackage.haskell.org/package/criterion`, 2015. Online; accessed 26 September 2015.

[OT10] Bryan O'Sullivan and Johan Tibell. Scalable i/o event handling for ghc. In *ACM Sigplan Notices*, volume 45, pages 103–108. ACM, 2010.

[PA98] George A Papadopoulos and Farhad Arbab. Coordination models and languages. *Advances in computers*, 46:329–400, 1998.

[Pat03] Dave Patterson. A conversation with jim gray. *ACM Queue*, 1(4):53–56, 2003.

[Pat10] David Patterson. The trouble with multicore: Chipmakers are busy designing microprocessors that most programmers can't handle. *IEEE Spectrum*, 2010.

[PD10] Daniel Peng and Frank Dabek. Large-scale incremental processing using distributed transactions and notifications. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[Pes06] Yaniv Pessach. UDP DELIVERS: Take Total Control Of Your Networking With .NET And UDP. *Microsoft MSDN Magazine*, pages 56–65, 2006.

[PK98] Vu Anh Pham and Ahmed Karmouch. Mobile software agents: an overview. *Communications Magazine, IEEE*, 36(7):26–37, 1998.

[PPR$^+$09] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[PSSC04] André Pang, Don Stewart, Sean Seefried, and Manuel MT Chakravarty. Plugging haskell in. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 10–21. ACM, 2004.

[RC90] Gruia-Catalin Roman and H Conrad Cunningham. Mixed programming metaphors in a shared dataspace model of concurrency. *Software Engineering, IEEE Transactions on*, 16(12):1361–1373, 1990.

[RDD$^+$08] Giovanni Russello, Changyu Dong, Naranker Dulay, Michel Chaudron, and Maarten Van Steen. Encrypted shared data spaces. In *Coordination Models and Languages*, pages 264–279. Springer, 2008.

[Rea09] Monash Reasearch. ebay,s two enormous data warehouses. `http://www.dbms2.com/2009/04/30/ebays-two-enormous-data-warehouses/`, 2009. Online; accessed 26 September 2015.

[RGAB10] M J Rashti, R E Grant, A Afsahi, and P Balaji. iWARP redefined: Scalable connectionless communication over high-speed Ethernet. In *High Performance Computing (HiPC), 2010 International Conference on*, pages 1–10, December 2010.

[RGO06] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. *URL http://www. rgoarchitects. com/Files/fallacies. pdf*, page 20, 2006.

[RH16] Inc Red Hat. Ansible is simple it automation. `https://www.ansible.com/`, 2016. Online; accessed 14 May 2016.

[ria] Riak homepage. `http://docs.basho.com/`. Online; Accessed 15 February 2013.

[RMI$^+$04] G. Regnier, S. Makineni, I. Illikkal, R. Iyer, D. Minturn, R. Huggahalli, D. Newell, L. Cline, and A. Foong. TCP onloading for data center servers. *Computer*, 37(11):48–58, 2004.

[RS] J. Ramdas and J. Srinivas. Extend Java EE containers with cloud characteristics.

[RTLQ09] Y. Ren, H. Tang, J. Li, and H. Qian. Performance comparison of UDP-based protocols over fast long distance network. *Information Technology Journal*, 8(4):600–604, 2009.

[SAK07] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook White Paper*, 5, 2007.

[SÇZ05] Michael Stonebraker, Ugur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM SIGMOD Record*, 34(4):42–47, 2005.

[ser15] Serf by HashiCorp. `http://serfdom.io`, 2015. Online; accessed Sep 26, 2015.

[Sha06] Nati Shalom. Space-based architecture and the end of tier-based computing [pdf document]. retrieved february 14, 2013. `http://www.gigaspaces.com/WhitePapers`, 2006.

[Sin89] Mukesh Singhal. Deadlock detection in distributed systems. *Computer*, 22(11):37–48, 1989.

[Sin11] Satnam Singh. Computing without processors. *Communications of the ACM*, 54(8):46–54, 2011.

[Sis87] Jonathan E Sisk. *Pick Basic: a programmer's guide*. Tab Books, 1987.

[SK12] Viktor Sovietov and M. Kharchenko. Erlang on XEN. `http://www.erlangonxen.org`, 2012. Online; accessed 26 September 2015.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10. IEEE, 2010.

[SN] Salvatore Sanfilippo and Pieter Noordhuis. Redis. `http://redis.io`. Online; accessed 15 February 2013.

[Spe91] Rick Spence. *Clipper programming guide*. Microtrend Books, 1991.

[spo15] In praise of ”boring” technology. `https://labs.spotify.com/2013/02/25/in-praise-of-boring-technology/more-104`, 2015. Online; accessed 26 September 2015.

[Sto86] Michael Stonebraker. The case for shared nothing. *IEEE Database Eng. Bull.*, 9(1):4–9, 1986.

[Sun10] Mark Sung. From Cloud Computing To Cloud Computing. *Network and Systems Support for Games (NetGames), 2010 9th Annual Workshop*, pages 1–48, 2010.

[SW08] Peter Seveik and Rebecca Wetzel. Improving Effective WAN Throughput for Large Data Flows. *NetForecast Report 5095*, pages 1–8, November 2008.

[Tal12] Nassim Nicholas Taleb. *Antifragile: Things that gain from disorder*, volume 3. Random House Incorporated, 2012.

[TDJ13] Samira Tasharofi, Peter Dinges, and Ralph E Johnson. Why do scala developers mix the actor model with other concurrency models? In *ECOOP 2013–Object-Oriented Programming*, pages 302–326. Springer, 2013.

[TMPJM12] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. Safe haskell. In *Proceedings of the 2012 symposium on Haskell symposium*, pages 137–148. ACM, 2012.

[TSLZ12] Nam-Luc Tran, Sabri Skhiri, Arthur Lesuisse, and Esteban Zimányi. Arom: Processing big data with data flow graphs and functional programming. In *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pages 875–882. IEEE, 2012.

[Unta] Akka (toolkit and runtime for building highly concurrent, distributed and fault tolerant event-driven applications on the jvm). `http://akka.io`. Online; accessed May 2012.

[Untb] cloudstack. `http://cloudstack.org`. Online; accessed May 2012.

[Untc] ØMQ. `http://www.zeromq.org/`. Online; accessed May 2012.

[Untd] The epass package. `http://hackage.haskell.org/package/epass`. Online; accessed May 2012.

[Unte] The netfilter.org project. `http://www.netfilter.org/`. Online; accessed May 2012.

[Unt09a] Light Weight Event System. `http://www.lwes.org/`, 2009. Online; accessed May 2012.

[Unt09b] Stackoverflow. `http://stackoverflow.com/questions/1435359/why-can-you-only-prepend-to-lists-in-functional-languages`, 2009. Online; accessed May 2012.

[Unt12a] Data.Sequence. `http://hackage.haskell.org/packages/archive/containers/0.4.2.1/doc/html/Data-Sequence.html`, 2012. Online; accessed May 2012.

[Unt12b] MessagePack. `http://msgpack.org/`, 2012. Online; accessed May 2012.

[Unt12c] MessagePack Blog. `http://msgpack.wordpress.com/`, 2012. Online; accessed May 2012.

[Unt12d] Programming Language Popularity. `http://www.langpop.com`, 2012. Online; accessed May2012.

[VCH07] Tom Murphy Vii, Karl Crary, and Robert Harper. Type-safe distributed programming with ml5. In *Trustworthy Global Computing*, pages 108–123. Springer, 2007.

[vdGSW97] Roel van der Goot, Jonathan Schaeffer, and Gregory V Wilson. Safer tuple spaces. In *Coordination Languages and Models*, pages 289–301. Springer, 1997.

[Vin07] Steve Vinoski. Concurrency with erlang. *Internet Computing, IEEE*, 11(5):90–93, 2007.

[VMD$^+$13] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[Vog08] Werner Vogels. Keynote: Uncertainty, The Next Web Conference. April 2008.

[Vog09] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[VRMB11] L M Vaquero, L Rodero-Merino, and R Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[WCC+09] Chen-Chi Wu, Kuan-Ta Chen, Chih-Ming Chen, Polly Huang, and Chin-Laung Lei. On the challenge and design of transport protocols for MMORPGs. *Multimedia Tools and Applications*, 45(1-3):7–32, 2009.

[web] Gigaspaces xap. `http://www.gigaspaces.com`. Online; accessed 2014.

[web15] Docker, build ship and run any app anywhere. `https://www.docker.com`, 2015. Online; accessed 26 September 2015.

[Wel05] George Wells. Coordination languages: Back to the future with linda. In *Proceedings of the Second International Workshop on Coordination and Adaption Techniques for Software Entities (WCAT05)*, pages 87–98, 2005.

[Wik15a] Wikipedia. Asynchronous i/o — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Asynchronous_I/O`, 2015. Online; accessed 22 July 2015.

[Wik15b] Wikipedia. dbase. `https://en.wikipedia.org/wiki/DBase`, 2015. Online; accessed 20 October 2015.

[Wik15c] Wikipedia. Fallacies of distributed computing — wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Fallacies_of_distributed_computing`, 2015. Online; accessed June 5, 2015.

[Wik15d] Wikipedia. Kernel density estimation. `https://en.wikipedia.org/wiki/Kernel_density_estimation`, 2015. Online; accessed 17 August 2015.

[WN10] Guohui Wang Guohui Wang and TSE Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. *IEEE INFOCOM. Proceedings*, pages 1–9, 2010.

[www15] The history of foxpro. `http://www.foxprohistory.org/`, 2015. Online; accessed 20 October 2015.

[YIFB08] Y Yu, M Isard, D Fetterly, and M Budiu. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th . . .* , 2008.

[ZCF+10] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[ZHC$^+$12] Wenbo Zhang, Xiang Huang, Ningjiang Chen, Wei Wang, and Hua Zhong. Paas-oriented performance modeling for cloud computing. In *Computer Software and Applications Conference (COMPSAC), 2012 IEEE 36th Annual*, pages 395–404. IEEE, 2012.