Programming Model and Protocols for Reconfigurable Distributed Systems

COSMIN IONEL ARAD

ROYAL INSTITUTE
OF TECHNOLOGY

Doctoral Thesis in
Electronic and Computer Systems
Stockholm, Sweden 2013

Programming Model and Protocols for Reconfigurable Distributed Systems

ROYAL INSTITUTE
OF TECHNOLOGY

Programming Model and Protocols for Reconfigurable Distributed Systems

COSMIN IONEL ARAD

Doctoral Thesis
Stockholm, Sweden 2013

TRITA-ICT/ECS AVH 13:07
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/07-SE
ISBN 978-91-7501-694-8

KTH – Royal Institute of Technology
School of Information and
Communication Technology
Electrum 229, SE-164 40 Kista, Sweden

SICS Dissertation Series 62
ISSN 1101-1335
ISRN SICS-D-62-SE

Swedish Institute of Computer Science
Computer Systems Laboratory
Box 1263, SE-164 29 Kista, Sweden

Printed and bound by Universitetsservice US-AB

Abstract

Distributed systems are everywhere. From large datacenters to mobile devices, an ever richer assortment of applications and services relies on distributed systems, infrastructure, and protocols. Despite their ubiquity, testing and debugging distributed systems remains notoriously hard. Moreover, aside from inherent design challenges posed by partial failure, concurrency, or asynchrony, there remain significant challenges in the implementation of distributed systems. These programming challenges stem from the increasing complexity of the concurrent activities and reactive behaviors in a distributed system on the one hand, and the need to effectively leverage the parallelism offered by modern multi-core hardware, on the other hand.

This thesis contributes Kompics, a programming model designed to alleviate some of these challenges. Kompics is a component model and programming framework for building distributed systems by composing message-passing concurrent components. Systems built with Kompics leverage multi-core machines out of the box, and they can be dynamically reconfigured to support hot software upgrades. A simulation framework enables deterministic execution replay for debugging, testing, and reproducible behavior evaluation for large-scale Kompics distributed systems. The same system code is used for both simulation and production deployment, greatly simplifying the system development, testing, and debugging cycle.

We highlight the architectural patterns and abstractions facilitated by Kompics through a case study of a non-trivial distributed key-value storage system. CATS is a scalable, fault-tolerant, elastic, and self-managing key-value store which trades off service availability for guarantees of atomic data consistency and tolerance to network partitions. We present the composition architecture for the numerous protocols employed by the CATS system, as well as our methodology for testing the correctness of key CATS algorithms using the Kompics simulation framework.

Results from a comprehensive performance evaluation attest that CATS achieves its claimed properties and delivers a level of performance competitive with similar systems which provide only weaker consistency guarantees. More importantly, this testifies that Kompics admits efficient system implementations. Its use as a teaching framework as well as its use for rapid prototyping, development, and evaluation of a myriad of scalable distributed systems, both within and outside our research group, confirm the practicality of Kompics.

Părinţilor mei

Acknowledgements

I am truly honoured to have been a mentee of my advisor, Professor Seif Haridi. Seif's inspiring enthusiasm and curiosity, his broad knowledge and ample technical expertise, his kind guidance, and his sound principles of deep understanding and systematic investigation, gladly shared every step of the way, have all made my PhD journey a fun and profound learning experience. *Thank you, dear sir!*

I am extremely grateful to Doctor Jim Dowling, who has been an early and tireless champion of this work. Jim's phenomenal energy and helpful suggestions were a constant source of motivation and encouragement. *Go raibh maith agat, Jim!*

I was gratified by my fellow students who graciously volunteered to contribute enhancements and improvements to Kompics. Markus Kilås wrote a NetBeans IDE plug-in enabling point-and-click component design and code generation for an early, yet unnamed version of Kompics. Frej Drejhammar fixed a subtle scheduling bug and indulged in captivating discussions about intelligent software testing and many other exciting technical topics. Niklas Ekström ported Kompics to Python. *Tack så mycket till alla!* Lars Kroll ported Kompics to Scala and wrote an elegant DSL; he also contributed a scheduling optimization and wrote an Eclipse IDE plug-in for static checking and browsing Kompics architectures. *Danke schön!*

I thank Tallat M. Shafaat for many happy shared enterprises during the PhD years, for pointing out good food, and especially for our excellent collaboration on CATS. Our work on CATS was extended by Muhammad Ehsan ul Haque, who added multiple data persistence engines, crash-recovery support, and efficient data transfers. *Bahut shukriya!* Hamidreza Afzali implemented range queries and Alexandru Ormenişan contributed a neat data indexing and query API. *Merci!*

I am much obliged to Professor Vladimir Vlassov for his valuable suggestions and constructive critiques of this work. I was so fortunate to have my education broadened by Professor Rassul Ayani in modeling and simulation, by Professor

Christian Schulte in constraint programming, and by Professor Dilian Gurov in formal methods. I got helpful input from Professor Johan Montelius and Professor Robert Rönngren. Many thanks to the head of the Software and Computer Systems department, Thomas Sjöland, and to the entire administrative staff for all the help they provided me with throughout my time at KTH. *Tack så väldigt mycket!*

I have had both the luck and the pleasure of many delightful and enlightening conversations with Per Brand, Lars Rasmusson, Karl-Filip Faxén, Sverker Janson, Roland Yap, Sameh El-Ansary, Erik Aurell, Joe Armstrong, György Dán, Ian Marsh, Šarūnas Girdzijauskas, Konstantin Popov, Martin Nilsson, Björn Grönvall, Mikael Nehlsen, Adam Dunkels, and Victoria Knopf, in the pleasant and resourceful ambiance at SICS. In particular, I thank Ali Ghodsi for his advice on research and writing, and for inspiring me with his rigor, clarity, and determination. *Tusen tack!*

I am thankful to all my colleagues and friends, who provided good company over the years, including Ahmad Al-Shishtawy, Amir Payberah, Fatemeh Rahimian, Mikael Högqvist, Roberto Roverso, John Ardelius, Joel Höglund, Martin Neumann, Raul Jimenez, Flutra Osmani, Daniela Bordencea, Hamid Mizani, Alex Averbuch, Shahab Mokarizadeh, Nima Dokoohaki, Ozair Kafray, and Salman Niazi. *Cheers!*

I gratefully acknowledge funding from the Swedish Research Council, who supported the Kompics work with grant 2009-4299, as well as funding from the European Commission within the SelfMan and the EverGrow research projects.

I was granted the privilege of refreshing context switches from my thesis work by Ken Goldman at Google and Mihai Budiu at Microsoft Research. *Thanks a lot!*

I am truly indebted to my earlier teachers and mentors who were largely responsible for launching me into the PhD orbit. Maria Dărăbanţ showed me the beauty of English. Maria Enache revealed the elegance of Mathematics and imparted me with her passion for problem solving. Octavian Purdilă shared his fascination for operating systems and computer networks, as well as his excellence, design taste, and curiosity for tinkering with software systems. *Mulţumesc frumos!*

I owe my deepest gratitude to my beloved Emma, who has been extremely supporting, unbelievably understanding, and relentlessly patient with my work-related investigations and my late-night coding and eager debugging excursions. I bow down to you my dearest friend, for being a wonderful companion through these academic adventures and onwards. *Vilpittömällä rakkaudella ja kiitos paljon!*

Finally, I want to convey my very special thanks to my parents, Viorica and Ionel, and to my sister Cristina, who have been expressing genuine interest in my progress on this dissertation. Thank you for your love and support, for giving me wings to embark on this journey, to carry on and bring it to fruition, and for instilling in me the conscience of a job well done. This work is dedicated to you. *Vă mulţumesc cu drag!*

Cosmin Ionel Arad
Stockholm, April 2013

Contents

List of Figures

List of Source Code Listings

Chapter 1

Introduction

A large and increasing fraction of the world's computer systems are distributed. Distribution is employed to achieve *scalability*, *fault-tolerance*, or it is just an artifact of the *geographical separation* between the system participants. Distributed systems have become commonplace, operating across a wide variety of environments from large data-centers to mobile devices, and offering an ever richer combination of services and applications to more and more users.

All distributed systems share inherent challenges in their design and implementation. Often quoted challenges stem from concurrency, partial failure, node dynamism, or asynchrony. We argue that today, there is an underacknowledged challenge that restrains the development of distributed systems. The *increasing complexity* of the concurrent activities and reactive behaviors in a distributed system is unmanageable by today's programming models and abstraction mechanisms.

Any first-year computer science student can quickly and correctly implement a sorting algorithm in a general purpose programming language. At the same time, the implementation of a distributed consensus algorithm can be time consuming and error prone, even for an experienced programmer who has all the required expertise. Both sorting and distributed consensus

are basic building blocks for systems, so why do we witness this state of affairs? Because currently, programming distributed systems is done at a too low level of abstraction. Existing programming languages and models are well suited for programming local, sequential abstractions, like sorting. However, they are ill-equipped with mechanisms for programming high-level distributed abstractions, like consensus.

Testing and debugging distributed systems is also notoriously hard. Despite previous work [97, 223] that focused on performance testing through scalable and accurate network emulation, correctness testing and debugging distributed systems is largely a still unsolved problem. The dire state of tool support for building and testing distributed systems, which leaves researchers and practitioners to face the complexity challenges head on, has been acknowledged by world-renowned experts in the field [47]. In describing the experience of building Chubby [45], Google's lock service based on the Paxos [130, 131] consensus algorithm, Tushar D. Chandra, recipient of the Edsger W. Dijkstra Prize in Distributed Computing, writes:

> "The fault-tolerance computing community has not developed the tools to make it easy to implement their algorithms.
>
> The fault-tolerance computing community has not paid enough attention to testing, a key ingredient for building fault-tolerant systems."

Having identified these shortcomings of the distributed computing field, the expert concludes with a call to action, asserting the importance of finding solutions to these challenging open problems:

> "It appears that the fault-tolerant distributed computing community has not developed the tools and know-how to close the gaps between theory and practice with the same vigor as for instance the compiler community. Our experience suggests that these gaps are non-trivial and that they merit attention by the research community."

This lays the foundation for the motivation of this thesis. Our overarching goal is to make it easy to implement distributed algorithms and systems, and to make it easy to test and debug them.

1.1 Motivation

Modern hardware is *increasingly parallel*. In order to effectively leverage the hardware parallelism offered by modern multi-core processors, concurrent software is needed. There exist two major software concurrency models: shared-state concurrency and message-passing concurrency. (We view dataflow concurrency as a special case of message-passing concurrency where dataflow variables act as implicit communication channels.) It appears that there is broad consensus among concurrent programming researchers and practitioners, that the message-passing concurrency model is superior to the shared-state concurrency model.

Message-passing concurrency has proved not only to scale well on multi-core hardware architectures [31] but also to provide a simple and *compositional* concurrent programming model, free from the quirks and idiosyncrasies of locks and threads. As demonstrated by the actor model [2], epitomized by programming languages like Erlang [20, 21], and message-passing frameworks like Kilim [206] or Akka [216], message-passing concurrency is both very easy to program and it makes it easy to reason about concurrent program correctness. Additionally, Erlang supports the construction of software that can be safely upgraded in place without stopping the system. This is a crucial prerequisite for enabling dynamic system evolution for mission-critical, always-on systems.

While Erlang and actor-based message-passing frameworks provide compositional concurrency, multi-core scalability, and actor isolation, they do little to help deal with increasing software complexity. The crux of the problem is that despite offering modular abstraction [169], these models do not restrict communication between processes to occur only through module interfaces. By allowing processes to communicate with any other processes in the software architecture, and not only with their architectural neighbors, these models violate the Law of Demeter [138, 137], and thus fail to realize its benefits of good software maintainability and adaptability. This proliferation of implicit process references leads to tight coupling between modules and, despite Erlang's support for online code upgrades, it ends up becoming a hindrance to dynamic software reconfiguration.

Protocol composition frameworks like Horus [224, 225], Ensemble [101], Bast [84], or Appia [156], were specifically designed for building distributed

systems by layering modular protocols. Multilayered software architectures can systematically implement the Law of Demeter, and as such, they may enjoy the full benefits of loose coupling [34, 207]. This approach certainly simplifies the task of programming distributed systems, however, these frameworks are often designed with a particular protocol domain in mind, and enabling protocol composition solely by layering, limits their generality. As we show in this dissertation, *nested* hierarchical composition enables richer, more useful architectural patterns.

More general programming abstractions, like nested hierarchical components, are supported by modern component models like OpenCom [63], Fractal [44], or Oz/K [139], which also provide dynamic system reconfiguration, an important feature for long-running or always-on, mission-critical systems, and for evolving or self-adaptive systems. However, the style of component interaction, based on synchronous interface invocation or atomic rendezvous, precludes compositional concurrency in these models, making them unfit for present-day multi-core hardware architectures.

To summarize, our motivation is to contribute models, techniques, and tools, to make it easy to implement, evaluate, and test distributed systems, in an attempt to bridge the gap between the theory and the practice of distributed computing. In an endeavour to accommodate the modern trends of increasing hardware parallelism and increasing software complexity, we seek to make modular distributed system implementations, tackling their complexity through hierarchical nested composition, and enabling them to effortlessly leverage multi-core processors for parallel execution, while being dynamically reconfigurable.

1.2 Design Philosophy

With Kompics we propose a message-passing, concurrent, and hierarchically nested component model with support for dynamic reconfiguration. We also propose a systematic methodology for designing, programming, composing, deploying, testing, debugging, and evaluating distributed systems. Our key principles in the design of Kompics are as follows:

- First, we tackle the increasing complexity of modern distributed systems through *modular abstraction* and *nested hierarchical composition*.

This facilitates modeling entire subsystems as first-class composite components, not only isolating them and hiding their implementation details [169], but also enabling distributed system designs based on the concept of virtual nodes [208], or, executing an entire distributed system within a single OS process, for testing and debugging.

- Second, we choose a *message-passing concurrency* model. Message-passing concurrency is preferable to shared-state concurrency because it scales better on multi-core processors; it makes it easier to reason about correctness; it simplifies programming, largely avoiding the inefficiencies and synchronization complexities of locks; and most importantly, because it is compositional.

- Third, we decouple components from each other to enable *dynamic reconfiguration* and system evolution for critical, always-on systems. Publish-subscribe component interaction enables both architectural decoupling (components are unaware of their communication counterparts) and temporal decoupling (asynchronous communication) as well as runtime dependency injection.

- Fourth, we decouple component code from its executor to enable different execution modes. The same system code can then be executed in *distributed production deployment*, in *local interactive testing* enabling quick incremental development, and in *deterministic repeatable simulation* for correctness testing and replay debugging.

1.3 Thesis Contributions

This thesis aims to raise the level of abstraction in programming distributed systems. We provide constructs, mechanisms, architectural patterns, as well as programming, concurrency, and execution models that enable programmers to construct and compose reusable and modular distributed abstractions. We believe this is an important contribution because it lowers the cost and accelerates the development and evaluation of more reliable distributed systems.

With Kompics we contribute a programming model and a set of techniques designed to simplify the development of reconfigurable distributed

systems. The practicality of the Kompics programming model is underscored by a number of salient features that follow naturally from its design philosophy. The Kompics framework supports a comprehensive set of methods for testing distributed systems: local interactive testing support, quick iterative and incremental development and testing support, local or distributed stress testing support, as well as protocol correctness testing through complex simulation experiment scenarios and safety and liveness predicates validation.

A remarkable characteristic enabled by this model is the ability to execute the *same* system implementation in either production deployment mode or in repeatable simulation mode for testing and stepped debugging. In production deployment mode, Kompics systems are automatically executed in parallel on multi-core machines, seamlessly leveraging hardware parallelism and largely circumventing multi-core programming challenges [178, 52, 196]. Using the same system code for simulation and deployment avoids the need to maintain two different implementations, which would otherwise add both development overhead and potential for errors through divergence in the different code bases.

Kompics offers a systematic methodology for designing, programming, composing, deploying, testing, debugging, and evaluating distributed systems. These characteristics of the framework, together with a rich library of provided protocols and abstractions, ultimately led to its usage for prototyping, evaluating, and developing a plethora of distributed systems, both within and outside of our research group.

Some examples of distributed systems built with Kompics include a peer-to-peer *video-on-demand* system [37], a secure and fault-tolerant distributed *storage* system [111], NAT-aware *peer-sampling* protocols [73, 172], peer-to-peer *live media streaming* systems [170, 174, 171, 173, 176], locality-aware scalable *publish-subscribe* systems [187], scalable *NAT-traversal* protocols [164], distributed hash-table *replication schemes* [200], gossip protocols for *distribution estimation* [175], an *elasticity controller* simulator [162, 161], studies of multi-consistency-model *key-value stores* [7, 41], mechanisms for *robust self-management* [6, 22], and a *reliable UDP* protocol [157]. The broad variety of these applications is a testament to the usefulness of Kompics.

Furthermore, for more than five years, Kompics has been successfully used as a teaching framework in two Master's level courses on distributed

systems: a course on advanced distributed algorithms and abstractions, and a course on large-scale and dynamic peer-to-peer systems. Kompics enabled students to compose various distributed abstractions and to experiment with large-scale overlays and content-distribution networks, both in simulation and real distributed deployments. Students were able both to deliver running implementations of complex distributed systems, and to gain insights into the dynamic behavior of those systems.

Within this dissertation, we offer both a qualitative and a quantitative evaluation of Kompics. The qualitative evaluation focuses on the programming idioms, protocol composition patterns, and architectural designs that Kompics facilitates, and their implications on the development, testing, and debugging of distributed systems. The quantitative evaluation comprises a number of microbenchmarks of the Kompics runtime, as well as end-to-end performance measurements of CATS, a non-trivial distributed system that we built using Kompics. CATS is a scalable and consistent key-value store which trades off service availability for guarantees of atomic data consistency and tolerance to network partitions. We present CATS as a case study of using Kompics for building and testing distributed systems.

Within CATS, we introduce *consistent quorums* as an approach to guarantee linearizability [106] in a decentralized, self-organizing, dynamic system spontaneously reconfigured by consistent hashing [120], and prone to inaccurate failure suspicions [49] and network partitions [66].

We showcase consistent quorums in the design and implementation of CATS, a distributed key-value store where every data item is an atomic register [129] with linearizable *put* and *get* operations, and which is hosted by a dynamically reconfigurable replication group [56].

We evaluate the cost of consistent quorums and the cost of achieving atomic data consistency in CATS. We give evidence that consistent quorums admit system designs which are scalable, elastic, self-organizing, fault-tolerant, consistent, and partition-tolerant, on the one hand, as well as system implementations with practical performance and modest overhead, on the other hand.

CATS delivers sub-millisecond operation latencies under light load, single-digit millisecond operation latencies at 50% load, and it sustains a throughput of one thousand operations per second, per server, while scaling linearly to hundreds of servers. This level of performance is competitive

with that of systems with a similar architecture but which provide only weaker consistency guarantees [125, 60].

1.4 Source Material

The material in this dissertation has been previously published in the following internationally peer-reviewed articles:

- Cosmin Arad, Jim Dowling, and Seif Haridi. *Message-Passing Concurrency for Scalable, Stateful, Reconfigurable Middleware*. In Proceedings of the Thirteenth ACM/IFIP/USENIX International Conference on Middleware, volume 7662 of Lecture Notes in Computer Science, Springer [16]. **Middleware 2012**, Montreal, Canada, December 2012.

- Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. *Brief Announcement: Atomic Consistency and Partition Tolerance in Scalable Key-Value Stores*. In Proceedings of the Twenty-sixth International Symposium on Distributed Computing, volume 7611 of Lecture Notes in Computer Science, Springer [17]. **DISC 2012**, Salvador, Brazil, October 2012.

- Cosmin Arad, Jim Dowling, and Seif Haridi. *Developing, Simulating, and Deploying Peer-to-Peer Systems using the Kompics Component Model*. In Proceedings of the Fourth International Conference on COMmunication System softWAre and MiddlewaRE, ACM Digital Library [14]. **COMSWARE 2009**, Dublin, Ireland, June 2009.

- Cosmin Arad, Jim Dowling, and Seif Haridi. *Building and Evaluating P2P Systems using the Kompics Component Framework*. In Proceedings of the Ninth International Conference on Peer-to-Peer Computing, IEEE Communications Society [15]. **P2P 2009**, Seattle, WA, USA, September 2009.

- Cosmin Arad and Seif Haridi. *Practical Protocol Composition, Encapsulation and Sharing in Kompics*. In Proceedings of the Second International Conference on Self-Adaptive and Self-Organizing Systems Workshops, IEEE Computer Society [13]. **SASO Workshops 2008**, Venice, Italy, October 2008.

- Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. *CATS: Atomic Consistency and Partition Tolerance in Scalable and Self-Organizing Key-Value Stores*. Currently under submission. Also available as SICS Technical Report T2012:04 [18].

The author of this dissertation was the main contributor to the design of Kompics as well as the principal developer of the reference Kompics implementation in Java. The open source code repository for the Kompics platform, as well as further documentation, videos, and tutorials, were made publicly available at `http://kompics.sics.se/`.

The work on the design, implementation, and evaluation of CATS was partially done in collaboration with Tallat M. Shafaat, who contributed a partition-tolerant topology maintenance algorithm as well as a garbage collection mechanism, and he duly reported parts of the work and our results in his doctoral dissertation. The open source code repository for CATS, together with an interactive demonstration of the deployed system, was made publicly available at `http://cats.sics.se/`.

1.5 Organization

This dissertation is structured in two parts. In Part I we introduce the Kompics component model and programming framework.

- Chapter 2 describes the first-class concepts in Kompics and the operations upon these. It also presents the asynchronous publish-subscribe style of communication between components, as well as aspects pertaining to component initialization and life cycle management, fault isolation, and dynamic reconfiguration.

- Chapter 3 presents the basic distributed systems programming patterns enabled by Kompics and it illustrates how higher-level distributed computing abstractions can be built from lower-level abstractions. Finally, it shows a few examples of peer-to-peer protocols and services implemented in Kompics.

- Chapter 4 provides implementation details related to the component execution model and multi-core scheduling, scalable network com-

munication, and enabling deterministic single-threaded simulation. It also discusses the implementation of Kompics in various modern programming languages and aspects of programming in the large.

- Chapter 5 compares Kompics to related work in the areas of protocol composition frameworks, concurrent programming models and process calculi, reconfigurable component models and software architecture description languages, and frameworks for simulation and replay debugging of large-scale distributed systems.

In Part II we present CATS, a non-trivial distributed system that we built using Kompics, in order to showcase the architectural patterns and the system development cycle support provided by the Kompics framework.

- Chapter 6 motivates the work on CATS by overviewing the landscape of existing scalable storage systems and arguing for the need for scalable and consistent fault-tolerant data stores for mission-critical applications. It also reviews the principal replicated data consistency models, quorum-based replication systems, and the impossibility of simultaneous consistency, availability, and partition tolerance.

- Chapter 7 introduces *consistent quorums*, a novel technique which enables distributed algorithms designed for a static group of processes and relying on majority quorums, to continue to operate correctly in process groups with dynamically reconfigurable group membership. This was instrumental in adapting a static atomic register protocol to operate at arbitrary large scales, within coherent dynamic groups.

- Chapter 8 illustrates the software architecture of the CATS system as a composition of protocols and service abstractions, and it discusses various system design choices. It also demonstrates the Kompics methodology of interactive testing, which supports incremental development, and protocol correctness testing and debugging based on whole-system repeatable simulation.

- Chapter 9 evaluates the performance of the CATS system implemented in Kompics Java, showing both that the component model

admits efficient system implementations and that our consistent quorums technique achieves atomic consistency at modest overheads.

- Chapter 10 discusses alternative consistency models that can be easily provided on CATS' foundation of scalable reconfigurable group membership, as well as alternative efficient implementations of these models. It also compares CATS with related work in the areas of scalable key-value stores and consistent meta-data storage systems.

Chapter 11 concludes this dissertation by highlighting the benefits and limitations of both Kompics and CATS, sharing the lessons learnt, and pointing to future research directions.

Part I

Building Distributed Systems from Message-Passing Concurrent Components

Chapter 2

Component Model

Kompics is a component model [211] targeted at building distributed systems by composing protocols programmed as event-driven components. Kompics components are reactive state machines that execute concurrently and communicate by passing data-carrying typed events, through typed bidirectional ports which are connected by channels. This chapter introduces the conceptual entities of our component model and its programming constructs, its concurrent message-passing execution model and publish-subscribe component communication style, as well as constructs enabling dynamic reconfiguration, component life cycle and fault management.

2.1 Concepts in Kompics

The fundamental Kompics entities are *events*, *ports*, *components*, *event handlers*, *subscriptions*, and *channels*. We introduce them here and show examples of their definitions with snippets of Java code. The Kompics component model is programming language independent, however, we use Java to illustrate a formal definition of its concepts. In Section 4.7 we show examples of Kompics entities written in other programming languages like Scala and Python to which Kompics has been ported.

2.1.1 Events

Events are passive and immutable *typed* objects having any number of typed attributes. The type of an attribute can be any valid data type in the implementation programming language. New event types can be defined by sub-classing existing ones.

Code 2.1 illustrates a simple example event type definition in Java. For clarity, we omit the constructor, and any getters, setters, access modifiers, and import statements. The Message event contains two attributes: a source and a destination Address, which is a data type containing an IP address, a TCP or UDP port number, and an integer virtual node identifier.

Code 2.1 A simple event type

```
1 class Message extends Event {
2     Address source;
3     Address destination;
4 }
```

Code 2.2 shows an example of a derived event type. In our Java implementation of Kompics, all event types are descendants of a root type, Event. We write DataMessage $\subseteq$ Message to denote that DataMessage is a subtype of Message. In diagrams, we represent an event using the 

graphical notation, where Event is the event's type, e.g., Message.

Code 2.2 A derived event type

```
1 class DataMessage extends Message {
2     Data data;
3     int sequenceNumber;
4 }
```

2.1.2 Ports

Ports are *bidirectional* event-based component interfaces. A port is a gate through which a component communicates asynchronously with other components in its environment, by sending and receiving events. A port allows a specific set of event types to pass in each direction. We label the two directions of a port as *positive* (+) and *negative* (−). The *type* of a port

specifies the set of event types that can traverse the port in the positive direction and the set of event types that can traverse the port in the negative direction. Concretely, a port type definition consists of two sets of event types: a "positive" set and a "negative" set. We regard ports as service interfaces implemented or required by components, and conceptually, we view negative events as service *request* events and positive events as service *indication* events. There is no sub-typing relationship for port types.

Code 2.3 shows a simple example of a port type definition in Java. The code block in the inner braces represents an "instance initializer". The *positive* and *negative* methods populate the respective sets of event types. In our Java implementation, each port type is a singleton.

Code 2.3 A Network port definition

```
1 class Network extends PortType {
2     {
3         positive(Message.class);            // indication
4         negative(Message.class);            // request
5     }
6 }
```

In this example we defined a Network port type which allows events of type Message, or any subtype thereof, to pass in both ('+' and '−') directions. The Timer port type defined in Code 2.4 allows request events of type ScheduleTimeout and CancelTimeout to pass in the '−' direction, and indication events of type Timeout to pass in the '+' direction.

Code 2.4 A Timer port definition

```
1 class Timer extends PortType {
2     {
3         indication(Timeout.class);          // positive
4         request(ScheduleTimeout.class);     // negative
5         request(CancelTimeout.class);       // negative
6     }
7 }
```

Conceptually, a port type can be seen as a service or protocol abstraction with an event-based interface. It accepts *request* events and delivers *indica-*

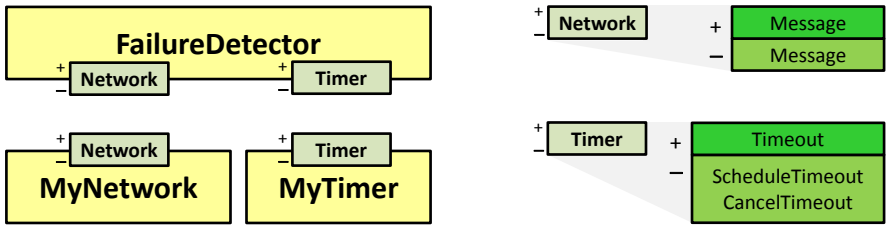

Figure 2.1. The MyNetwork component has a *provided* Network port. MyTimer has a *provided* Timer port. The FailureDetector has a *required* Network port and a *required* Timer port. In diagrams, a *provided* port is figured on the top border, and a *required* port on the bottom border of a component.

tion or *response* events. By convention, we associate requests with the '−' direction and responses or indications with the '+' direction. In the example of Code 2.3, a Timer abstraction accepts ScheduleTimeout requests and delivers Timeout indications. Code 2.4 defines a Network abstraction which accepts Message events at a sending node (*source*) and delivers Message events at a receiving node (*destination*) in a distributed system.

A component that *implements* a protocol or service will *provide* a port of the type that represents the implemented abstraction. Through this provided port, the component will receive the request events and it will trigger the indication events specified by the port's type. In other words, for a provided port, the '−' direction is incoming into the component and the '+' direction is outgoing from the component.

In Figure 2.1, the MyNetwork component provides a Network port and the MyTimer component provides a Timer port. In diagrams, we represent a port using the +/− Port graphical notation, where Port is the type of the port, e.g., Network. We represent components using the Component graphical notation. The right side of the figure contains a legend illustrating the request and indication events of the Network and Timer port types.

When a component *uses* a lower level abstraction in its implementation, it will *require* a port of the type that represents the abstraction. Through a required port, a component sends out the request events and receives the indication/response events specified by the required port's type. In other words, for required ports, the '−' direction is outgoing from the component and the '+' direction is incoming into the component.

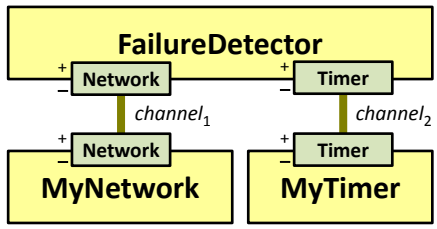

Figure 2.2. *channel*$_1$ connects the provided Network port of the MyNetwork component with the required Network port of the FailureDetector component. *channel*$_2$ connects the provided Timer port of the MyTimer component with the required Timer port of the FailureDetector component.

2.1.3 Channels

Channels are first-class bindings between component ports. A channel connects two *complementary* ports of the *same* type. For example, in Figure 2.2, *channel*$_1$ connects the provided Network port of the MyNetwork component with the required Network port of the FailureDetector component. This allows, Message events sent by the FailureDetector to be received and handled by the MyNetwork component.

Channels forward events in both directions in FIFO order, i.e., events are delivered at each destination component in the same order in which they were triggered at a source component. In diagrams, we represent channels using the 

graphical notation. We omit the channel name when it is not relevant.

Event filters can be associated with each direction of a channel, instructing the channel to forward only particular events which match the filter. We discuss channel event filters in more detail in Section 2.4. To enable the dynamic reconfiguration of the software architecture, event forwarding through channels can be paused and resumed. We discuss these channel operations enabling dynamic reconfiguration in Section 2.9.

2.1.4 Event Handlers

An event handler is a first-class procedure of a component. A handler accepts events of a particular type, and any subtypes thereof, and it is executed *reactively* when the component receives such events. During

its execution, a handler may trigger new events and mutate the component's local state. The Kompics execution model guarantees that the event handlers of one component instance are *mutually exclusive*, i.e., they are executed sequentially. This alleviates the need for synchronization between different event handlers of the same component accessing the component's mutable state, which greatly simplifies their programming.

Code 2.5 illustrates an example event handler definition in Java. Upon receiving a Message event, the *handleMsg* event handler increments a local message counter and prints a message to the standard output console.

Code 2.5 A simple event handler

```
1 Handler<Message> handleMsg = new Handler<Message>() {
2   public void handle(Message message) {
3     messages++;  // ← component-local state update
4     System.out.println("Received from " + message.source);
5   }
6 };
```

In diagrams, we use the $h\langle\text{Event}\rangle$ graphical notation to represent an event handler, where h is the handler's name and Event is the type of events accepted by the handler, e.g., Message.

2.1.5 Subscriptions

A subscription binds an event handler to a component port, enabling the event handler to handle events that arrive at the component on that port. A subscription is allowed only if the type of events accepted by the handler, say E, is allowed to pass by the port's type definition. In other words, if $\mathcal{E}$ is the set of event types that the port allows to pass in the direction of the event handler, then either $\text{E} \in \mathcal{E}$, or E must be a subtype of a member of $\mathcal{E}$.

Figure 2.3 illustrates the *handleMsg* event handler from our previous example, being subscribed to the component's required Network port. In diagrams, we represent a subscription using the $\longrightarrow$ graphical notation.

In this example, the subscription of *handleMsg* to the Network port is allowed because Message is in the positive set of Network. As a result of this subscription, *handleMsg* will handle all events of type Message or of any subtype of Message, received on this Network port.

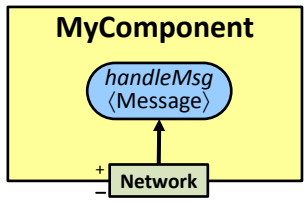

Figure 2.3. The *handleMsg* event handler is *subscribed* to the required Network port of MyComponent. As a result, *handleMsg* will be executed whenever MyComponent receives a Message event on this port, taking the event as an argument.

2.1.6 Components

Components are event-driven state machines that execute *concurrently* and communicate *asynchronously* by message passing. In the implementation programming language, components are objects consisting of any number of local state variables and event handlers. Components are modules that export and import event-based interfaces, i.e., provided and required ports. Each component is instantiated from a component definition.

Code 2.6 shows the Java component definition corresponding to the component illustrated in Figure 2.3. Line 2 specifies that the component

Code 2.6 A simple component definition

```
1 class MyComponent extends ComponentDefinition {
2   Positive<Network> network = requires(Network.class);
3   int messages;          // ← local state, ↖required port
4   public MyComponent()   // ← component constructor
5     System.out.println("MyComponent created.");
6     messages = 0;
7     subscribe(handleMsg, network);
8   }
9   Handler<Message> handleMsg = new Handler<Message>() {
10    public void handle(Message msg) {
11      messages++;  // ← component-local state update
12      System.out.println("Received from " + msg.source);
13    }
14  };
15 }
```

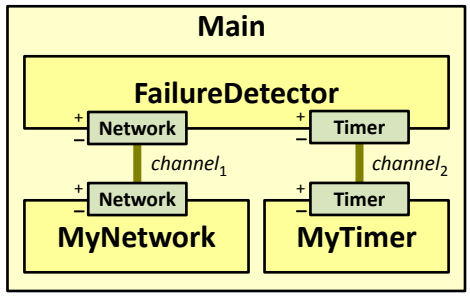

Figure 2.4. The Main component encapsulates a FailureDetector component, a MyNetwork component, and a MyTimer component.

has a required Network port. The *requires* method returns a reference to a required port, *network*, which is used in the constructor to subscribe the *handleMsg* handler to this port (see line 7). The type of the required port is Positive⟨Network⟩ because, for required ports the positive direction is incoming into the component. Both a component's ports and its event handlers are first-class entities which allows for their dynamic manipulation.

Components can encapsulate subcomponents to hide implementation details [169], reuse functionality, and manage system complexity. Composite components enable the control and dynamic reconfiguration of entire component ensembles as if they were simple components. Composite components form a containment hierarchy rooted at a Main component. An example is shown in Figure 2.4. Main is the first component created when the run-time system starts and it recursively creates all other subcomponents. Since there exist no components outside of it, Main has no ports.

Code 2.7 illustrates the Main component specification in Java. In our Java implementation of Kompics, the Main component is also a Java main class; lines 13–15 show the *main* method. When executed, this will invoke the Kompics run-time system, instructing it to bootstrap, i.e., to instantiate the root component using Main as a component specification (see line 14).

In lines 5–7, Main creates its subcomponents and saves references to them. In line 8, it connects MyNetwork's provided Network port to the required Network port of the FailureDetector. As a result, $channel_1$ is created and saved. Unless needed for dynamic reconfiguration (see Section 2.9), channel references need not be saved.

Code 2.7 A root component definition in an executable program

```
1 class Main extends ComponentDefinition {
2   Component net, timer, fd;          // ← subcomponents
3   Channel channel1, channel2;        // ← channels
4   public Main() {                    // ↙ constructor
5     net = create(MyNetwork.class);
6     timer = create(MyTimer.class);
7     fd = create(FailureDetector.class);
8     channel1 = connect(net.provided(Network.class),
9                        fd.required(Network.class));
10    channel2 = connect(timer.provided(Timer.class),
11                       fd.required(Timer.class));
12  }
13  public static void main(String[] args) {
14    Kompics.bootstrap(Main.class);
15  }
16 }
```

Kompics components are *loosely coupled* [34]. A component does not know the type, availability, or identity of any components with which it communicates. Instead, a component only "communicates", i.e., it subscribes its handlers and it triggers events on its own ports or the ports of its subcomponents. It is up to the component's environment, i.e., its parent component, to wire up the communication channels.

Explicit component dependencies, in the form of required ports, enable dependency injection [83], which facilitates testing, and also contribute to facilitating the dynamic reconfiguration of the component architecture, a fundamental feature for evolving, long-lived systems. Because component communication occurs only through adjacent ports, Kompics satisfies *architectural integrity* [8], an important prerequisite for dynamic reconfiguration.

2.2 Kompics Operations

While presenting the Kompics concepts we have already introduced some of the basic operations on these concepts, such as *subscribe*, *create*, and *connect*. These operations have counterparts that undo their actions, namely *unsubscribe*, *destroy*, and *disconnect*, and these have the expected semantics.

Code 2.8 Commands enabling dynamic reconfiguration

```
class Main extends ComponentDefinition {
  Component net, timer, fd;          // ← subcomponents
  Channel channel1, channel2;        // ← channels
  public undo() {                    // ↙ some method
    disconnect(net.provided(Network.class),
               fd.required(Network.class));
    disconnect(timer.provided(Timer.class),
               fd.required(Timer.class));
    destroy(net);
    destroy(timer);
    destroy(fd);
  }
}
```

Code 2.8 illustrates the *destroy* and *disconnect* operations in the context of our previous example. A fundamental command in Kompics is *trigger*, which is used to asynchronously send an event through a port. In the next example, illustrated in Code 2.9 and Figure 2.5, MyComponent handles a MyMessage event due to its subscription to its required Network port. Upon handling the first message, MyComponent triggers a MyMessage reply on its Network port and then it unsubscribes its *myMsgH* event handler, thus handling no further messages. In diagrams, we denote that an event handler

Code 2.9 An example component handling a single network message

```
class MyComponent extends ComponentDefinition {
  Positive<Network> network = requires(Network.class);
  public MyComponent() { // ← component constructor
    subscribe(myMsgH, network);
  }
  Handler<MyMessage> myMsgH = new Handler<MyMessage>() {
    public void handle(MyMessage m) {
      trigger(new MyMessage(m.destination, m.source),
              network);
      unsubscribe(myMsgH, network); // ← reply only once
    }
  };
}
```

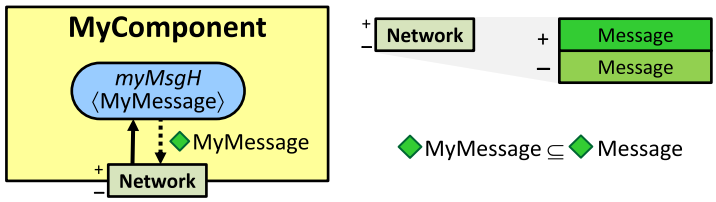

Figure 2.5. MyComponent handles one MyMessage event and triggers a MyMessage reply on its required Network port.

may trigger an event on a particular port, using the Event graphical notation. We discuss more Kompics operations in Sections 2.8 and 2.9.

2.3 Publish-Subscribe Message Passing

Components are unaware of other components in their environment. A component can communicate, i.e., handle received events and trigger events, only through the ports visible within its scope. The ports visible in a component's scope are its own ports and the ports of its immediate subcomponents. Ports and channels forward triggered events toward other connected components, as long as the types of events triggered are allowed to pass by the respective port type specifications. Therefore, component communication is constrained by the connections between components as configured by their respective enclosing parent components.

Communication between components works according to a message-passing publish-subscribe model. An event published on one side of a port is forwarded to all channels connected to other side the port. We illustrate the Kompics publish-subscribe component communication with some examples. In Figure 2.6, every MessageA event triggered by MyNetwork on its provided Network port is delivered both at Component1 and Component2, by channel_1 and channel_2. In Figure 2.7, however, MessageA events triggered by MyNetwork are only going to be delivered at Component1 while MessageB events triggered by MyNetwork are only going to be delivered at Component2. In Figure 2.8, whenever MyNetwork triggers a MessageA event on its Network port, this event is delivered to MyComponent where it is handled by $\textit{handler}_1$. Conversely, whenever MyNetwork triggers a MessageB event on its

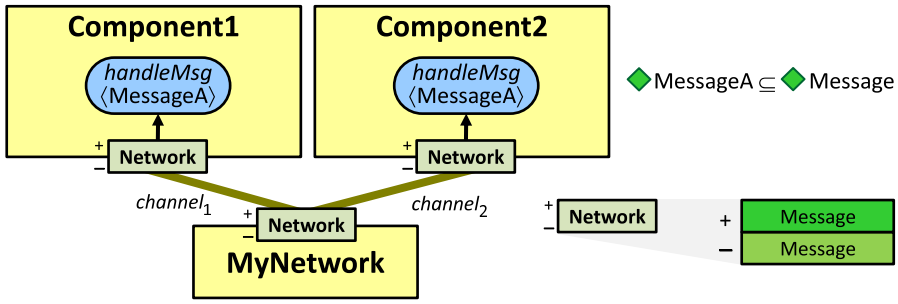

Figure 2.6. When MyNetwork triggers a MessageA on its provided Network port, this event is forwarded by both $channel_1$ and $channel_2$ to the required Network ports of Component1 and Component2, respectively.

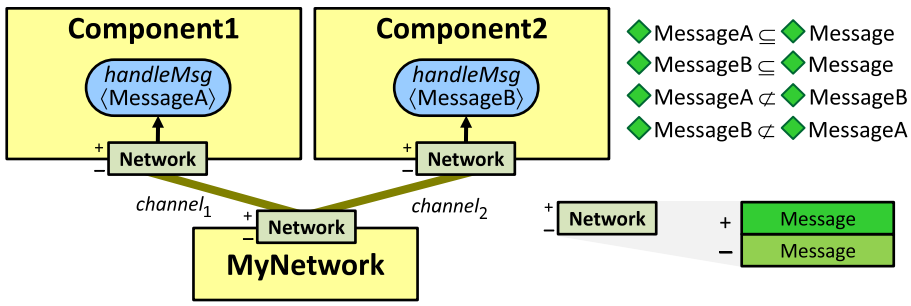

Figure 2.7. When MyNetwork triggers a MessageA event on its provided Network port, this event is forwarded only by $channel_1$ to the required Network port of Component1. MessageB events triggered by MyNetwork on its Network port, are forwarded only by $channel_2$ to the Network port of Component2.

Network port, this event is delivered to MyComponent where it is handled by $handler_2$. An event triggered (published) on a port is forwarded to other components by all channels connected to the other side of the port as in Figure 2.6. As an optimization, the run-time system should not forward events on channels that would not lead to any compatible subscribed handlers. An event received on a port is handled by all compatible handlers subscribed to that port as in Figure 2.9. Here, whenever MyNetwork triggers a MessageA event on its Network port, this event is delivered to MyComponent where it is handled sequentially by both $handler_1$ and $handler_2$, in the same order in which these two handlers were subscribed to the Network port.

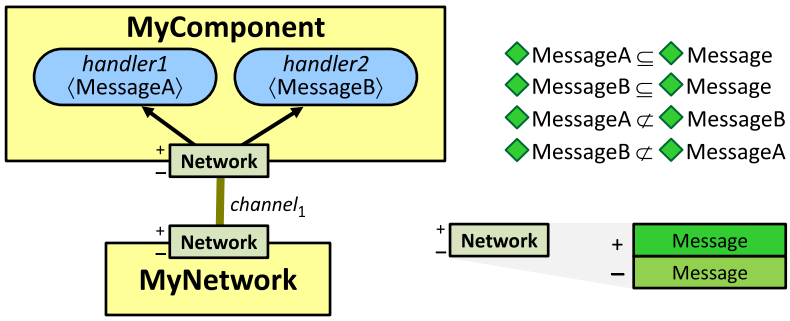

Figure 2.8. MessageA events triggered by MyNetwork on its Network port, are delivered to the Network port of MyComponent and handled by $handler_1$. MessageB events triggered by MyNetwork on its Network port, are delivered to the Network port of MyComponent and handled by $handler_2$.

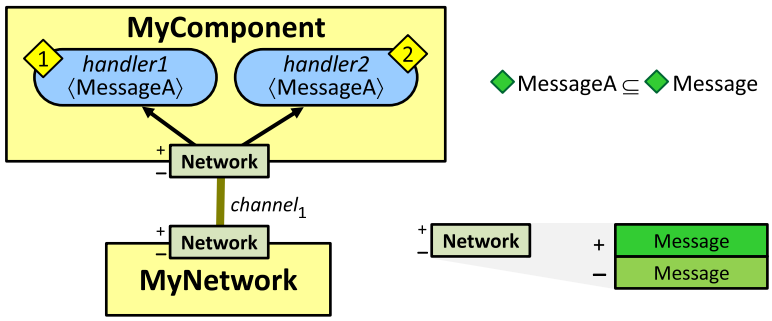

Figure 2.9. When MyNetwork triggers a MessageA event on its Network port, this event is delivered to the Network port of MyComponent and handled by both $handler_1$ and $handler_2$, sequentially (figured with yellow diamonds), in the order in which the two handlers were subscribed to the Network port.

2.4 Channel Event Filtering

Component reusability means that a component implementation can be used in different contexts without being changed. Component reuse may take the form of either creating multiple instances of the same component definition, or sharing the services provided by one component instance, among multiple other components. Sharing may avoid duplication of work and thus increase efficiency. For example, a failure detection service may be

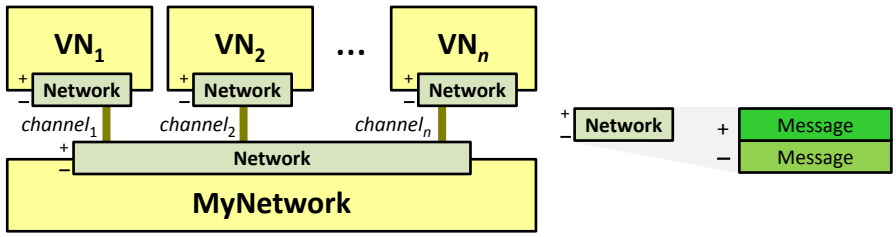

Figure 2.10. Each channel filters events in the '+' direction, only forwarding messages with a destination address matching the corresponding virtual node.

used by multiple protocols. Using a shared failure detector implementation, rather than one for each protocol, would save network bandwidth when multiple protocols need to monitor the failure of the same remote nodes.

Another basic example of component sharing is when multiple protocols on the same node share a single network component, which is in charge of managing network connections to remote nodes and message serialization and deserialization. The shared network component publishes received messages on its provided Network port, and each protocol subscribes to the message types it needs to handle. This type-based publish-subscribe mechanism works fine in this context but it becomes problematic when we want to package the protocols of one distributed system node into a composite component and execute multiple nodes within the same OS process, which enables whole-system repeatable simulation.

Similar to whole-system simulation is support for virtual nodes [208], whereby multiple nodes of the distributed system are executed on a single machine to facilitate load balancing and fast recovery [51]. An example is illustrated in Figure 2.10, where multiple virtual nodes, identical in structure but with different node identifiers, share the same network component and potentially the same IP address and port number. Destination addresses for different virtual nodes may differ only in their virtual node identifier.

In order to maintain the reusability of virtual node components, while at the same time avoiding the case where every virtual node handles every received message, we introduced channel filtering by event attributes.

Event attribute filters can be associated with a channel to instruct the channel to forward only events with certain attribute values in a particular direction. If channel x contains no filters, x forwards all events in both

directions. If x contains some filters, then it only forwards the events matching the respective filters. In our virtual nodes example of Figure 2.10, each channel filters messages sent in the positive direction by their destination address virtual node identifier. Messages need not be filtered in the negative direction. The channel event filters are specified by the parent component upon connecting the respective Network ports.

Attribute filtering enables complete component reusability by allowing the same component implementation to be used in different contexts and filtering its input events in its enclosing scope. This is quite appropriate given that this is precisely the scope in which the architect decides how the component is used. Attribute filtering also reduces the potential for errors by freeing the programmer from having to explicitly write code that rejects events that are not intended for a particular component. With n components sharing the same service, this avoids $O(n)$ operations. In our reference implementation of Kompics, both type-based and attribute-based filtering are implemented using a constant-time hash table lookup, enabling event filtering in $O(1)$ operations, and thus scalable publish-subscribe.

2.5 Request-Response Interaction

If we again consider the example of a failure detector service shared by multiple protocols, we notice that it is possible that different protocols may request the failure monitoring of different remote nodes. When a failure is detected and a corresponding notification is published on the provided service port, this is delivered to all client protocols, even to those that did not request de monitoring of the currently detected node.

More generally, given a *server* component providing a service with an interface based on requests and responses (e.g., Timer, Failure Detector), and multiple instances of a *client* component, using the service, given the publish-subscribe semantics described in Section 2.3, when one of the clients issues a request and the server handles it and issues a response, all clients receive the response. For this situation, Kompics provides two special types of events: Request and Response, which should be used in any port type definition which represents a request-response service potentially shared by multiple independent clients.

When a Request event is triggered by a client, as the event passes through different channels and ports in the architecture, it saves them on an internal stack. When the server component generates a Response event, it initializes it with the Request's stack. As the Response event is passed through the architecture, the run-time system pops its stack one element at a time to see where to deliver it next. This mechanism, ensures that only the client which initiated a Request will receive the corresponding Response.

2.6 Component Initialization and Life Cycle

Every component provides a special Control port used for initialization, life cycle, and fault management. Figure 2.11 illustrates the Control port type and a component that declares an Init, a Start, and a Stop handler. Typically, for each component definition that requires state initialization, we define a specific initialization event, as a subtype of Init, which contains component-specific configuration parameters.

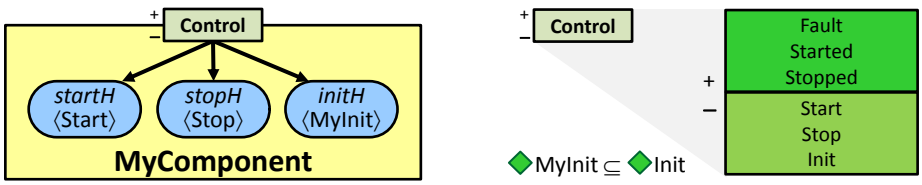

Figure 2.11. Every Kompics component provides a Control port by default. To this Control port, the component can subscribe Start, Stop, and Init handlers. In general, we do not illustrate the control port in component diagrams.

An Init event is guaranteed to be the first event handled by a component. When a component subscribes an Init event handler to its Control port in its constructor, the run-time system will only schedule the component for execution upon receiving an Init event.

Start and Stop events allow a component which handles them to take some actions when the component is activated or passivated. A component is created *passive*. In the passive state, a component can receive events but it will not execute them. (Received events are stored in a port queue.) When activated, a component will enter the *active* state executing any enqueued events. Handling life cycle events (illustrated in Code 2.10) is optional.

Code 2.10 Handling component initialization and life cycle events

```
class MyComponent extends ComponentDefinition {
  int myParameter;
  public MyComponent() { // ← component constructor
    subscribe(handleStart, control); // ←similar for Stop
    subscribe(handleInit, control);
  }
  Handler<MyInit> handleInit = new Handler<MyInit>() {
    public void handle(MyInit init) {
      myParameter = init.myParameter;
    }
  };
  Handler<Start> handleStart = new Handler<Start>() {
    public void handle(Start event) {
      System.out.println("started");
    }
  };
}
```

To activate a component, an enclosing composite component triggers Start event on the Control port of the subcomponent as shown in Code 2.11. Similarly, parent components can initialize or passivate their children by triggering Init or respectively Stop events, on their control ports.

Code 2.11 Triggering component initialization and life cycle events

```
  trigger(new MyInit(42), myComponent.control());
  trigger(new Start(), myComponent.control());
  trigger(new Stop(), myComponent.control());
```

When a composite component is activated (or passivated), its subcomponents are recursively activated (or passivated). The *bootstrap* construct, introduced in the Main component example (see Code 2.7), both creates and starts the Main component, recursively creating and starting all components.

When a composite component needs to perform dynamic reconfiguration on some of its subcomponents, it passivates them first (see Section 2.9). Once the parent component has triggered a Stop event on a subcomponent's Control port it does not mean that the subcomponent has already been passivated since handling the Stop event happens asynchronously.

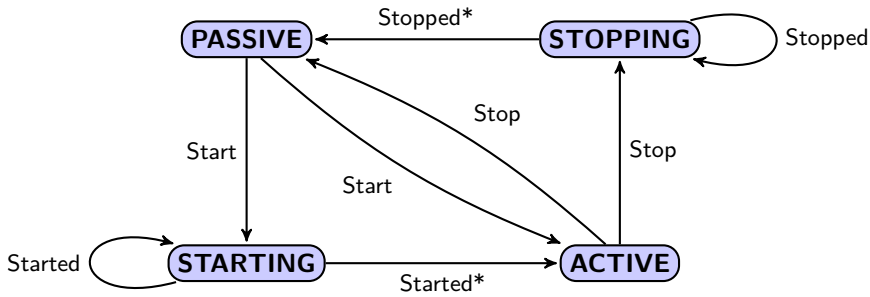

Figure 2.12. Kompics component life cycle state diagram.

Moreover, the subcomponent may itself have multiple subcomponents. A composite component becomes passive only once all its subcomponents have been passivated. When entering the *passive* state a subcomponent triggers a Stopped event on its Control port informing its parent of its passivation. As such, a composite component that needs to passivate first enters the *stopping* state, it triggers Stop events on the Control port of each of its subcomponents, and then it waits to handle a Stopped event from each subcomponent. Having received Stopped events from every subcomponent, the composite component enters the *passive* state and informs its parent by sending a Stop event. The life cycle diagram for Kompics components is illustrated in Figure 2.12. Starting a composite component happens in a similar fashion, first entering a *starting* state, and once all subcomponents become *active* the parent enters the *active* state itself. Start, Stop, Started, and Stopped event handlers which implement this behavior are provided by the run-time system. User-provided handlers for these events are optional.

From the point of view of handling regular events, a component is still considered active in the *stopping* state and it is still considered passive in the *starting* state. A component can only be destroyed when *passive*.

2.7 Fault Management

Kompics enforces a fault isolation and management mechanism inspired by Erlang [21]. A software fault or exception thrown and not caught within an event handler is caught by the run-time system, wrapped into a Fault event and triggered on the Control port, as shown in Figure 2.13.

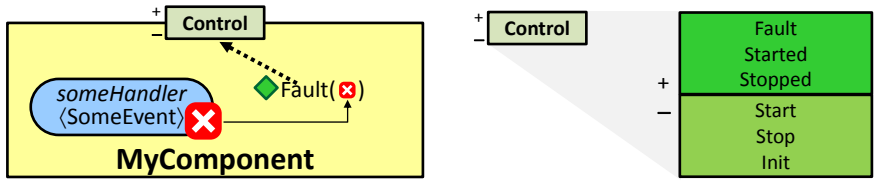

Figure 2.13. Uncaught exceptions thrown in event handlers are caught by the run-time system, wrapped in a Fault event and triggered on the Control port.

A composite component may subscribe a Fault handler to the Control ports of its subcomponents. The composite component can then replace the faulty subcomponent with a new instance, through dynamic reconfiguration, or take other appropriate actions. If a Fault is not handled in a parent component it, is further propagated to the parent's parent and so on until it reaches the Main component. If not handled anywhere, ultimately, a system fault handler is executed which logs the exception to standard error and halts the execution.

2.8 Non-blocking Receive

A component executes events received on a particular port, in the order in which they were received. This means that event execution order is dictated by the order in which other components trigger the events, and not by the programmer. It is sometimes necessary that a component waits to receive a particular event before continuing to execute other events. This behaviour is needed, for example, to implement a blocking remote call (RPC) into another component, whereby a client component triggers a request event and needs to wait for the response event before executing other events.

We want to allow the implementation of the Kompics model in environments where lightweight threads are not available, like Java, whereby components would be scheduled for execution over a fixed number of worker processing threads. For this reason, we decided not to provide a blocking *receive* primitive, whereby a component would block its executor thread in the middle of executing an event handler waiting to receive a particular event. Instead, we provide an *expect* primitive which does not block the thread executing the component, but which installs a one-time event

pattern within the component. The next event executed by the component is one that matches the installed pattern. If there is no such event already enqueued at the component, the component will wait for one, without executing any non-matching events. Once a matching event is received, the one-time event pattern is uninstalled.

The *expect* primitive is a non-blocking component synchronization mechanism. It allows a component to "block" awaiting a specific event. However, the waiting component only "blocks" after it finishes executing the current handler, not during its execution. This design allows for an implementation where heavyweight threads execute components and no continuation needs to be saved. It also means that a sequence of operations including a "blocking call" has to be programmed using two handlers. The first handler contains the operations before the "call", ending with triggering the request event and expecting the response event. The second handler handles the response event and contains the operations after the "call" returns. Any continuation state necessary for the execution of the second handler can be either local state in the component or included in the response event. The *expect* primitive has the following implications:

- *expect* breaks the FIFO property of ports and channels since the expected event is not necessarily the next event received;
- *expect* potentially reduces the parallelism of the execution since some events ready to be executed are actually delayed if they don't match the expect filter;
- *expect* makes it possible to program deadlocks since cycles of expectations may occur. For example, component a expects an event e_1 that would be triggered by component b, which expects an event e_2 that would be triggered by a, if a wasn't expecting event e_1.
- in traditional RPC for object-oriented systems, a method may contain several RPC calls. A Kompics component may have at most one installed pattern of expected events. In an event handler with multiple *expect* calls, the last *expect* call wins, therefore, a single method making multiple RPC calls should be split over several Kompics event handlers, each handler having effectively one *expect* call per RPC call.

2.9 Dynamic Reconfiguration

Kompics enables the dynamic reconfiguration of the component architecture without dropping any of the triggered events. In addition to the ability to dynamically create and destroy components, connect and disconnect ports, subscribe and unsubscribe handlers, Kompics supports four channel commands which enable safe dynamic reconfiguration: *hold*, *resume*, *plug*, and *unplug*. The *hold* command puts a channel on hold. The channel stops forwarding events and starts queuing them in both directions. The *resume* command has the opposite effect, resuming the channel. When a channel resumes, it first forwards all enqueued events, in both directions, and then keeps forwarding events as usual. The *unplug* command, unplugs one end of a channel from the port where it is connected, and the *plug* command plugs back the unconnected end to a (possibly different) port.

We highlight here the most common type of reconfiguration operation: swapping a component instance with a new instance. To replace a component c_1 with a new component c_2 (with similar ports), c_1's parent, p, puts on hold and unplugs all channels connected to c_1's ports; then, p passivates c_1, creates c_2 and plugs the unplugged channels into the respective ports of c_2 and resumes them; c_2 is initialized with the state exported by c_1 and then c_2 is activated. Finally, p destroys c_1.

Chapter 3

Programming Patterns and Distributed Abstractions

Having introduced the fundamental concepts of the Kompics component model let us now take a look at some of the programming idioms, patterns, and abstractions supported in Kompics. We start by discussing basic idioms like message passing between remote nodes of a distributed system, timer management and remote service invocation, and event interception patterns. We then illustrate how one can build higher-level abstractions from lower-level ones, with a few examples of fault-tolerant distributed computing abstractions. Finally, we present a framework of peer-to-peer services and protocols that were implemented using Kompics.

3.1 Distributed Message Passing

The Network abstraction is used for sending messages between remote nodes in a distributed system. Typically, for each component implementing a distributed protocol, a programmer defines component-specific protocol messages as subtypes of the Message event.

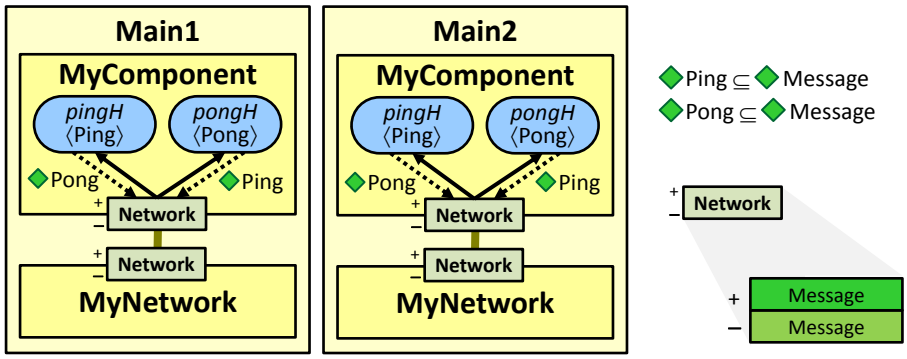

Figure 3.1. Two processes exchange Ping and Pong messages over an IP network. The MyNetwork component in each process is manages network connections to other processes and also handles message serialization and deserialization.

Figure 3.1 shows two processes sending Ping and Pong messages to each other as part of a protocol implemented by MyComponent. When designing MyComponent, the programmer knew it had to handle Ping and Pong messages, therefore these message types were also defined so that the *pingH* and *pongH* event handlers of MyComponent could be defined and subscribed to handle those events. Being subtypes of Message, both Ping and Pong have *source* and *destination* Address attributes. When MyComponent in Main1 wants to send a ping to MyComponent in Main2, it creates a Ping message using its own address as source, and Main2's address as destination, and triggers it on its required Network port. This Ping event is handled by MyNetwork in Main1, which marshals it and sends it to MyNetwork in Main2, which unmarshals it and triggers it on its Network port. The Ping event is delivered to MyComponent in Main2 where it is handled by *pingH*. The *pingH* handler in Main2 responds with a Pong event which is sent back to the Main1 process in a similar fashion.

The MyNetwork component in each process is configured with a network address to listen on for incoming connections. MyNetwork automatically manages network connections between processes. Each message type has an optional *transport* attribute which can be UDP or TCP (default). MyNetwork will send each message on a connection of the specified type, if one is currently open. Otherwise, MyNetwork first opens a connection to the destination process, and then it uses that to transmit the message.

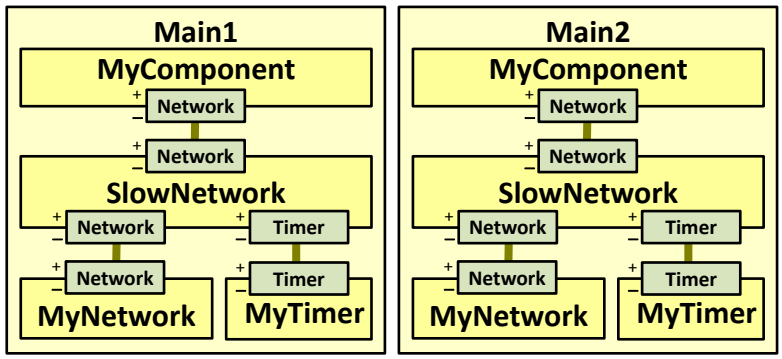

Figure 3.2. A SlowNetwork component was interposed between MyComponent and MyNetwork to emulate network latency. The SlowNetwork delays every sent message, according to a network model, before forwarding it to the MyNetwork.

3.2 Event Interception

Event interception is a fundamental pattern supported in Kompics. It allows a system architect to extend the functionality of a system without changing it. For example, let us take a look at the architecture in Figure 2.6, and let us assume that initially we only have Component1 which processes MessageA events. Without making any changes to Component1 or to any other part of the system, Component2 can later be added, e.g., by means of dynamic reconfiguration, in order to perform some non-functional task, such as keeping statistics on how many MessageA events were processed.

Event interception can also be used to interpose a component between two components connected by a channel, in order to perform complex filtering of events, to delay events, or to implement some form of admission control for events. Let us again consider the Ping-Pong example from Figure 3.1. In Figure 3.2 we modified the architecture by interposing a SlowNetwork between MyComponent and MyNetwork. The SlowNetwork delays every message sent by MyComponent by some random delay. In essence we emulate a slower network, which could be useful for testing the protocol in an otherwise fast LAN, by subjecting it to a congested network scenario. SlowNetwork could be configured to emulate specific fine-grained network conditions, which allows the user to experiment with the (unmodified) Ping-Pong protocol on a network with special properties.

3.3 Timer Management

Kompics alarms and timeouts are provided as a service abstraction through the Timer port. This guideline allows the usage of different implementations of the Timer abstraction in different execution environments. We illustrated the Timer port in Figure 2.1. It accepts two request events, ScheduleTimeout and CancelTimeout, and it delivers a Timeout indication event.

In Figure 3.3 we illustrate a component that uses a Timer abstraction. Typically, when designing a component such as MyComponent, one would also design specific timeout events, e.g., MyTimeout, as a subtype of the Timeout event. Multiple timeout event types can be defined for different timing purposes, so a component can have different event handlers for different timeouts. Code 3.1 illustrates how a timeout is scheduled.

Code 3.1 Scheduling a timeout alarm

```
1 class MyComponent extends ComponentDefinition {
2   Positive<Timer> timer = requires(Timer.class);
3   UUID timeoutId; // ← used for canceling
4   Handler<Start> startHandler = new Handler<Start>() {
5     public void handle(Start event) {
6       long delay = 5000; // milliseconds
7       ScheduleTimeout st = new ScheduleTimeout(delay);
8       st.setTimeoutEvent(new MyTimeout(st));
9       timeoutId = st.getTimeoutId();
10      trigger(st, timer);
11    }
12  };
13 }
```

To cancel a previously scheduled timeout, a component will issue a CancelTimeout request on its required Timer port. The CancelTimeout event needs to contain the unique identifier of the scheduled timeout that should be cancelled. Code 3.2 shows how to cancel a timeout.

Code 3.2 Canceling a timeout

```
1  CancelTimeout ct = new CancelTimeout(timeoutId);
2  trigger(ct, timer);
```

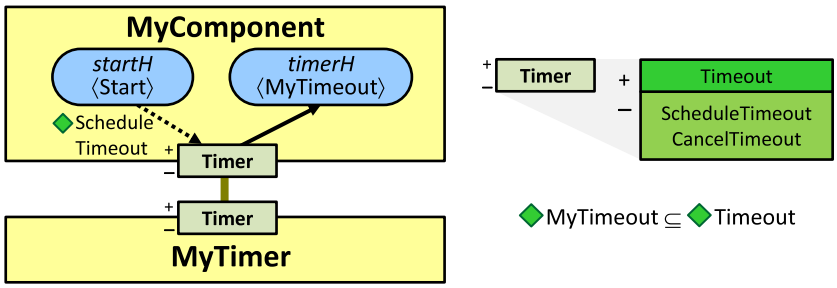

Figure 3.3. MyComponent uses the Timer abstraction provided by MyTimer.

3.4 Remote Service Invocation

A common idiom in many distributed systems is sending a request to a remote node and waiting for a response up to a timeout. This entails scheduling a timeout to be handled in case the response never arrives, e.g., in case the remote node crashes or the message is lost, and canceling the timeout when the response does arrive.

A recommended practice is for the client to send the unique timeout identifier in the request message, which is then echoed by the server it in the response message. This way, when the client node gets the response, it knows which timer to cancel. Another recommended practice is to keep a set of all outstanding requests or just their timeout identifiers. This helps with handling either the response or the timeout exclusively as follows: upon handling either the response or the timeout, the request is removed from the outstanding set. Neither the response nor the timeout is handled if the request is not outstanding anymore.

3.5 Distributed Computing Abstractions

In Section 3.1 we described message passing between two nodes in a distributed system, an example of a simple point-to-point communication abstraction. We used Kompics to implement and compose a wide array of higher-level fault-tolerant distributed computing abstractions [94, 46]. For example, we implemented different types of broadcast communication abstractions with various guarantees of reliable message delivery and

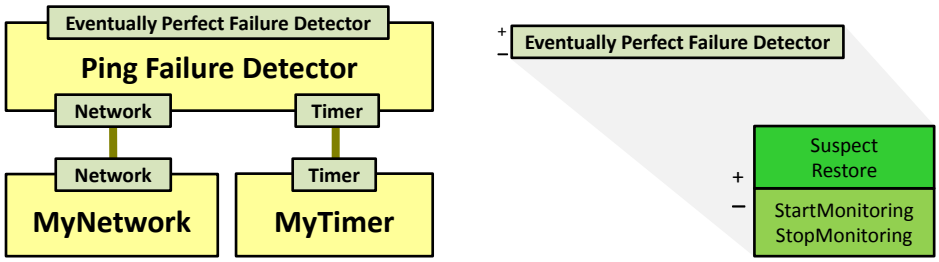

Figure 3.4. An eventually perfect failure detector abstraction.

ordering, failure detection abstractions, leader election, consensus, distributed shared memory abstractions with various consistency models such as sequential consistency, atomic registers, or regular registers, replicated state machines, etc. We highlight some of these abstractions and their implementation through protocol composition in the following sections.

3.5.1 Failure Detection

An eventually perfect failure detector abstraction detects the crashes of other nodes in a distributed system. The detector is called eventually perfect because it is allowed to make inaccurate crash detections whereby it falsely suspects other nodes to have crashed, however it should ultimately converge to an accurate behavior [49]. Therefore, the Eventually Perfect Failure Detector service abstraction shown in Figure 3.4 provides two indication events: Suspect and Restore, through which it notifies higher-level protocols that a particular node is suspected to have crashed, or that a previous suspicion is revised, respectively. The abstraction accepts two request events from higher-level protocol, namely to start and to stop monitoring a given node.

One possible implementation is to periodically send Ping messages to each monitored node and await Pong responses within a given timeout. A node is suspected to have crashed if a Pong is not received before the timeout expires. Whenever a Pong is received from a suspected node, the timeout value is increased and the suspicion is revised. Figure 3.4 illustrates a protocol composition whereby an eventually perfect failure detector abstraction is implemented using a Network and a Timer abstraction provided by the MyNetwork and MyTimer components respectively.

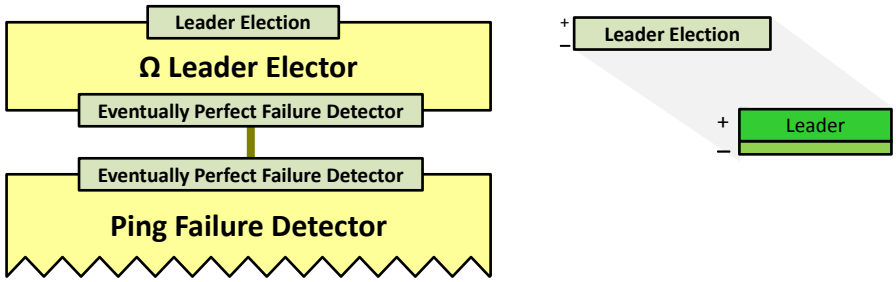

Figure 3.5. A Leader Election abstraction implemented using a failure detector.

3.5.2 Leader Election

A leader election abstraction enables choosing one node to be selected as a unique representative of a group of nodes in a distributed system. This abstraction is useful in situations where a single process should coordinate some steps of a distributed algorithm. In order to provide fault tolerance, a new leader is elected whenever the current leader crashes.

A leader election abstraction typically provides a Leader indication event which notifies higher-level protocols whenever a new leader is elected, informing them which node is the new leader. Figure 3.5 illustrates a protocol composition whereby a leader election abstraction is implemented by leveraging a failure detector abstraction. Indeed, leader election is closely related to failure detection and it is sometimes viewed as a failure detector: instead of detecting which processes have failed, it rather identifies one process that has not failed. Ω is a leader election abstraction which was shown to be the weakest failure detector to solve the consensus problem [48].

3.5.3 Broadcast

Broadcast communication abstractions allow the dissemination of information among a group of nodes in a distributed system. A typical broadcast abstraction offers a Broadcast request event through which a higher- level protocol solicits the dissemination of a given message, and a Deliver indication event through which the abstraction delivers a received message to higher-level protocols at all nodes in the group. There exist various flavours of broadcast abstractions and they differ in their fault-tolerance guarantees.

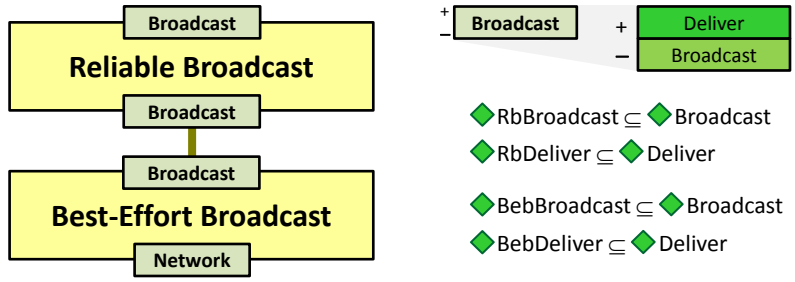

Figure 3.6. Two Broadcast abstractions built on top of a Network abstraction.

For example, an *unreliable broadcast* abstraction offers no guarantees on message delivery, whereby a message which is broadcast may be delivered at some nodes in the group but not at others. A *probabilistic broadcast* abstraction only guarantees message delivery with high probability, but it makes no deterministic guarantees. A *best-effort broadcast* abstraction guarantees that a message is delivered at all non-crashing nodes provided that the broadcasting node does not crash. A *reliable broadcast* abstraction guarantees that a message is delivered either at all or at none of the non-crashing nodes, regardless of whether the broadcasting node crashes or not during the execution of the protocol. In other words, if any of the non-crashing nodes delivers the message, then all other non-crashing nodes are guaranteed to deliver the message. A *uniform reliable broadcast* abstraction guarantees that if any node – crashing or not – delivers a message, then the message is eventually going to be delivered at all of the non-crashing nodes, regardless of whether the broadcasting node crashes or not.

Figure 3.6 illustrates a protocol composition whereby two broadcast abstractions are implemented on top of a network abstraction. Broadcast implementations may use helper messages and potentially message retransmission in order to satisfy some of the guarantees discussed above.

When we take into consideration multiple messages being broadcast, reliable broadcast abstractions may further differ in the guarantees they give on the *ordering* of message deliveries [98]. With regular reliable broadcast, different nodes may deliver different messages in different and completely independent orders. A *source-FIFO broadcast* abstraction guarantees that all messages originating from the same node are delivered at all nodes in

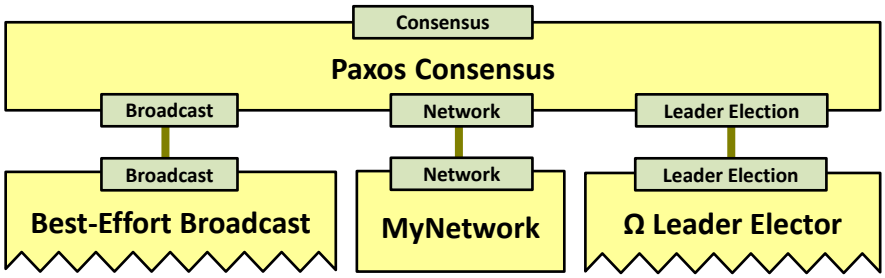

Figure 3.7. A Consensus protocol implemented using Ω and Best-Effort Broadcast.

the group in the same order in which they were broadcast by the source. A *causal-order broadcast* abstraction guarantees that message delivery order at each node in the group is consistent with the causal order [127] of Broadcast and Deliver events. A *total-order broadcast* abstraction, sometimes called atomic broadcast [68, 57], guarantees that all messages are delivered in the same order at all nodes in the group. This message delivery order does not need to be consistent with the causal order nor with the order in which messages were broadcast. It can be any order as long as it is the same at all nodes. Total-order broadcast was shown to be equivalent to the consensus problem [49].

3.5.4 Consensus

The consensus problem is probably the single most important problem in distributed computing. Any algorithm that helps multiple processes in a distributed system to maintain common state or to decide on a future action, in a model where some processes may fail, involves solving a consensus problem [177]. Processes use consensus to agree on a common value out of values they initially propose. A consensus abstraction is specified in terms of two events, Propose and Decide. Each process has an initial value that it proposes for consensus through a Propose request. All non-crashing processes have to decide on the same value through a Decide indication.

Figure 3.7 illustrates a protocol composition whereby a consensus abstraction is implemented by the Paxos algorithm [130, 131] which uses the Ω eventually accurate leader election abstraction and best-effort broadcast.

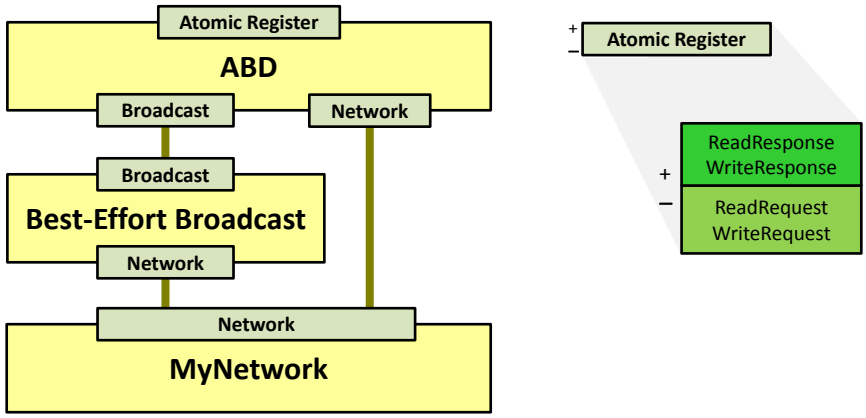

Figure 3.8. An Atomic Register distributed shared memory abstraction.

It was shown that in an asynchronous distributed system [24, 214] – one in which there is no bound on message transmission delay or on the relative speeds of different processes – the consensus problem is not solvable with a deterministic algorithm even if a single process may crash [81]. Therefore, consensus algorithms like Paxos [130, 131] or viewstamped replication [166] rely on partial synchrony [74].

3.5.5 Distributed Shared Memory

Distributed shared memory registers are abstractions for fault-tolerant data storage. They replicate data at a group of processes in a distributed system, effectively emulating a global shared memory using message-passing protocols to implement read and write operations.

A register abstraction is specified in terms of Read and Write request events and their associated indication responses. There exist various flavours of registers, e.g., *safe*, *regular*, *atomic* [129], which mainly differ in the consistency guarantees they provide in spite of data replication, process failures, and concurrent operations. Register abstractions also differ in the number of client processes allowed to perform reads and writes.

Figure 3.8 illustrates a protocol composition whereby an atomic register is implemented by the ABD algorithm [23] using a best-effort broadcast abstraction and a Network abstraction for point-to-point communication.

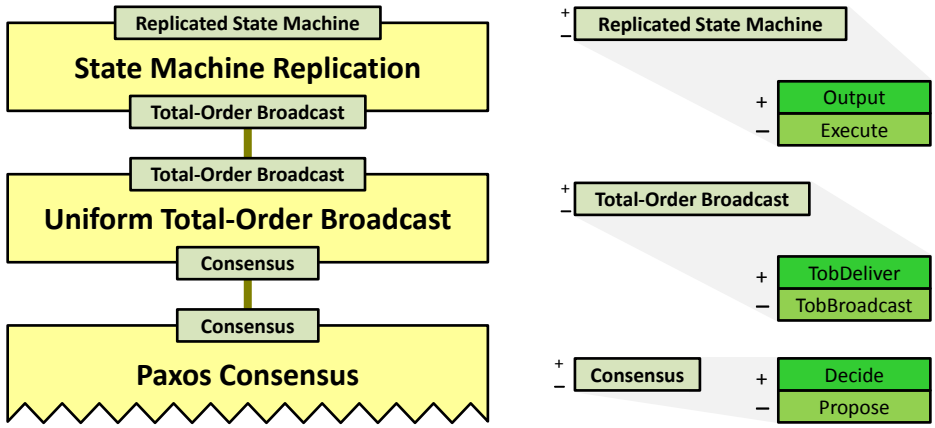

Figure 3.9. A Replicated State Machine abstraction using Total-Order Broadcast.

3.5.6 State Machine Replication

State machine replication (SMR) is a technique for building reliable and highly available distributed services [198]. A service, expressed as a state machine, consists of variables that encode its state, and commands that transform its state and may produce some output. To achieve fault tolerance, the service is replicated by a group of processes which coordinate to make sure they execute all commands, i.e., service requests, in the same order.

A replicated state machine abstraction is specified in terms of two events: an Execute request event used by a client to invoke the execution of a command on the state machine, and an Output indication event produced by the state machine as a result of executing the requested command.

All replicas are identical deterministic state machines and since they begin in the same initial state and perform all operations sequentially and in the same order, their state remains consistent.

Figure 3.9 illustrates a protocol composition, whereby a Replicated State Machine abstraction, is implemented by leveraging a Total-Order Broadcast abstraction. In turn, the uniform total-order broadcast implementation relies on Consensus. While atomic registers can be implemented in asynchronous systems, replicated state machines require partial synchrony, since they rely on consensus to agree on the command execution order. Atomic registers can trivially be implemented using a Replicated State Machine.

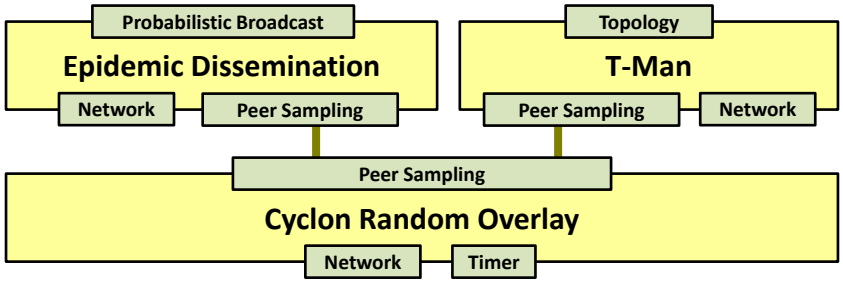

Figure 3.10. Gossip-based protocols for epidemic information dissemination and topology construction, implemented using a Peer Sampling service abstraction.

3.6 Peer-to-Peer Protocol Framework

We used Kompics to implement a set of generic and reusable peer-to-peer protocol components for building overlay network systems and content-distribution networks. Two characteristics that set these protocols and services apart from the abstractions discussed in the previous section, are their large scale and intense node dynamism, or churn [191, 210]. Peer-to-peer (P2P) protocols are typically deployed at Internet scale, operating over thousands to millions of machines scattered over wide geographical areas.

3.6.1 Random Overlays and Peer Sampling

The general paradigm of building scalable distributed systems based on the gossip communication model [9] has important applications which include information dissemination [69, 76], data aggregation [113], node clustering, ranking, and overlay topology management [114, 160]. At the heart of many such protocols lies a fundamental distributed abstraction: the *peer sampling* service [112, 115]. The aim of this service is to provide every node with a stream of peers to exchange information with, and a best effort is made to sample peers uniformly at random from the entire population while maintaining a small number of neighbor connections.

Figure 3.10 illustrates a protocol composition whereby a Peer Sampling abstraction, implemented by the Cyclon random overlay [227], is leveraged by the implementation of two higher-level abstractions. One is a probabilistic broadcast abstraction [38, 77] implemented by an epidemic information

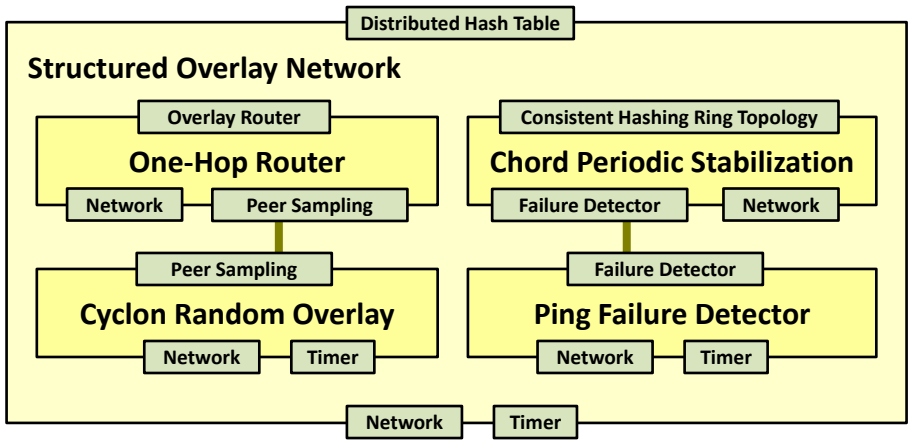

Figure 3.11. Protocols for structured overlay networks and distributed hash tables.

dissemination protocol [69, 76]. The other is a topology maintenance abstraction implemented by protocols like T-Man [114] or T-Chord [160] which can construct a distributed hash table [208, 192, 189, 233, 87] topology from a random graph of peers.

3.6.2 Structured Overlays and Distributed Hash Tables

Figure 3.11 illustrates a Kompics protocol composition for implementing a structured overlay network (SON) which provides a distributed hash table (DHT) service. Internally, a consistent hashing [120] ring topology is maintained by the Chord periodic stabilization protocol [208], which relies on a Failure Detector abstraction to maintain the topology in reaction to failure detection notifications. Also, an efficient location and routing protocol [96] is implemented using the Peer Sampling service. Kompics has been used – by students – to implement and experiment with other DHTs like Kademlia [151], which has a slightly different topology than Chord.

A DHT provides a lookup service similar to a hash table, where a set of (key, value) pairs is partitioned across peers, and any peer can efficiently retrieve the value associated with a given key. A DHT abstraction accepts Put and Get request events and issues corresponding responses. In Part II we present CATS, a DHT where every key-value pair is an Atomic Register.

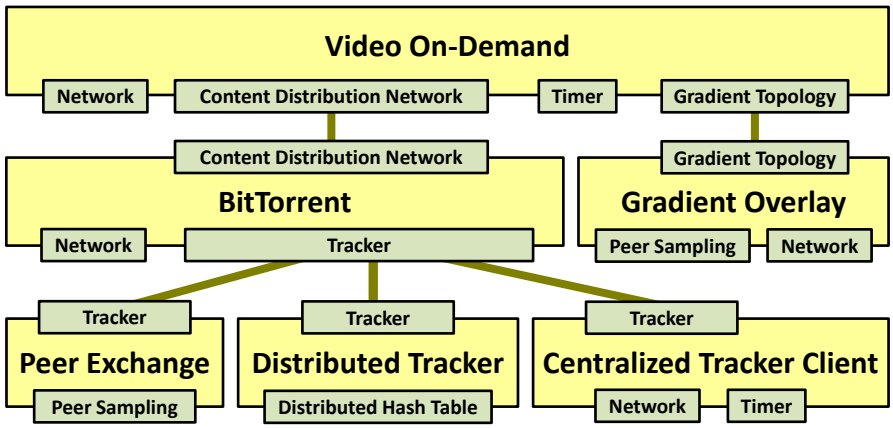

Figure 3.12. The BitTorrent protocol relying on multiple Tracker abstractions is used together with the Gradient Overlay to provide a video on demand service.

3.6.3 Content Distribution Networks and NAT Traversal

Kompics has been used for building content distribution networks (CDNs) like BitTorrent [58, 181], a P2P video on demand (VOD) system [37], and a number of P2P live media streaming protocols [171, 174, 170, 173, 176].

Figure 3.12 shows a protocol composition where the BitTorrent protocol relies on three different implementations of a Tracker abstraction. First, a Peer Exchange (PEX) protocol provides a Tracker service by relying on a Peer Sampling service. Second, a distributed tracker leverages a DHT service, namely Kademlia [151]. Third, a regular centralized tracker [58] is accessed through a client component. BitTorrent uses the Tracker abstraction to periodically find new peers in a CDN swarm, in the hope of discovering peers with better upload capacities, able to sustain faster data transfers. The Peer Sampling service is also used by a gradient overlay [194, 195] which ranks VOD peers according to their playback position in a video file.

Kompics has also been used to build NAT traversal infrastructures [164] which enable communication between private nodes – nodes behind NATs or firewalls – either by hole punching using protocols like STUN, or by relaying messages via public nodes which support direct connectivity. Interestingly, NAT traversal infrastructures often rely on structured overlay networks (SONs), e.g., to efficiently locate STUN servers.

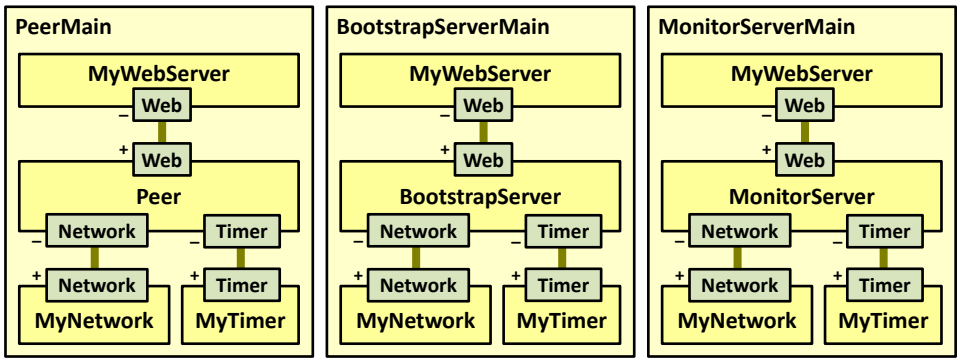

Figure 3.13. A peer process, a bootstrap server, and a monitoring server, all exposing a user-friendly web-based interface for troubleshooting peer-to-peer systems. The Peer composite component in each peer process encapsulates, besides other protocol components, a BootstrapClient component and a MonitorClient component which communicate periodically with their server counterparts. The Peer component also contains a WebApplication subcomponent handling WebRequests.

3.6.4 Peer-to-Peer Bootstrap and Monitoring

Peer-to-peer systems typically need a bootstrapping procedure to assist newly arrived nodes in finding nodes already in the system in order to execute any join protocols. To this end, the Kompics P2P framework contains a BootstrapServer component which maintains a list of online peers. Every peer embeds a BootstrapClient component which provides it with a Bootstrap service. When the peer starts, it issues a BootstrapRequest to the client which retrieves from the server a list of alive peers and delivers it through a BootstrapResponse to the local peer. The new peer then runs a join protocol using one or more of the returned nodes and after joining, it sends a BootstrapDone event to the BootstrapClient, which, from this point on, will send periodic keep-alive messages to the server letting it know this node is still alive. After a while, the BootstrapServer evicts from its list nodes which stopped sending keep-alive messages.

Another reusable service provided by the Kompics P2P framework, is a monitoring service. A MonitorClient component at each node periodically inspects the status of various local components, and may also aggregate various operational statistics. The client periodically sends reports to a

monitoring server that can aggregate the status of nodes and present a global view of the system through a web interface. The bootstrap and monitoring servers are illustrated in Figure 3.13, within executable main components, together with a peer process. The composite Peer component in the peer process encapsulates the BootstrapClient and the MonitorClient.

The Jetty web server [219] library is embedded in the MyWebServer component which wraps every HTTP request into a WebRequest event and triggers it on its required Web port. Both servers provide the Web abstraction, accepting WebRequests and delivering WebResponses containing HTML pages with the active node list and the global monitoring view, respectively. The local state of each peer can also be inspected on the web. To this end, the Peer component also contains a WebApplication subcomponent to which the peer delegates all WebRequests received on its provided Web port.

The subcomponents of the Peer component are omitted from Figure 3.13, however we give a complete example of a peer protocol composition in Figure 8.1 in the context of our case study of the CATS key-value store.

Chapter 4

Implementation Aspects and Development Cycle Support

Having presented the Kompics component model and some of the idioms, distributed programming abstractions, and protocol composition patterns that it enables, we now turn to discuss some implementation aspects.

The reference Kompics implementation was done in Java and released as an open-source project [19], available online at `http://kompics.sics.se`. The source code for the Java run-time system, component library, and the peer-to-peer protocol framework, together with further documentation, videos, and tutorials are all available from the project website.

In this chapter we present the component execution model and two different pluggable component schedulers for multi-core parallel execution and deterministic single-threaded simulation. We discuss support for incremental development, and we show how the same implementation of a distributed system can be subject to stress testing or executed in production deployment mode or in repeatable simulation mode for correctness testing and debugging. We also discuss the implementation of Kompics in other modern programming languages and aspects of programming in the large.

4.1 Component Execution and Scheduling

Kompics components are reactive state machines. In general, components do not have control threads of their own, however, the Kompics execution model admits an implementation with one lightweight thread per component like in Erlang [20] or Oz [193]. Since Java has only heavyweight threads, we use a pool of worker threads for executing components.

The Kompics run-time system spawns a number of worker threads that execute event handlers on behalf of components. Typically, the number of workers is equal to the number of processing cores or processors available on the machine. Each worker maintains a private work queue containing components which are considered ready for execution because they have received some events. Each component maintains a queue of received events for each of its ports. Workers manage component execution by transitioning a component to one of three states:

- *busy* – a worker is currently executing one of its event handlers;
- *ready* – one or more of its port event queues are not empty and the component is not *busy*; or
- *idle* – all its port event queues are empty and it is not *busy*.

Component execution proceeds as follows. If a worker has no components in its ready queue, it steals work from another worker. We describe the details of work stealing in the next subsection. When work becomes available at a worker, the worker picks the first ready component, say c, from its work queue. The worker then transitions c to the *busy* state. The component c now selects one of its ports with a non-empty event queue, say p, in a round-robin fashion, and then takes the first event, e, from the event queue of port p. Round-robin selection of ports ensures the fairness of event execution for events received on different ports. Next, c's event handlers that are subscribed to port p for events of the same type or a supertype of event e are executed in the order in which they were subscribed to p. After the handler execution terminates, if all the port event queues for c are empty, the worker transitions c to the *idle* state. Otherwise the worker transitions c to the *ready* state, and it places c on the tail of its work queue.

When a component is in the *idle* state and some worker places an event on one of its ports, the worker transitions the component from the *idle* state to the *ready* state and places it on its own work queue; *idle* components are not scanned by workers, so they contribute no scheduling overhead.

Workers process one component at a time and the same component cannot be processed by multiple workers at the same time. As each worker has a private queue with ready components, different workers can execute event handlers for different component instances in parallel. This improves concurrency, since there is no need for mutual exclusion between the event handlers of different component instances. However, different event handlers of the same component instance are still guaranteed to be executed sequentially and non-preemptively by workers. This eliminates the need for programmers to synchronise access to local component state variables between different event handlers, which reduces programming complexity.

The Kompics run-time system supports pluggable schedulers and allows users to provide their own component schedulers. Decoupling component behaviour from component execution enables the ability to use different component schedulers to execute the same, unaltered, component-based system in different execution modes such as parallel multi-core execution and deterministic simulation. In the next two subsections we highlight the default multi-core scheduler based on work-stealing and the default single-threaded deterministic scheduler used for repeatable simulations of entire distributed systems.

4.1.1 Multi-Core Work-Stealing Scheduler

Workers may run out of ready components to execute, in which case they engage in *work stealing* [40, 39]. Work stealing involves a *thief*, a worker with no ready components, contacting a *victim*, in our case, the worker with the highest number of ready components. The thief steals from the victim a batch of half of its ready components. Stolen components are moved from the victim's work queue to the thief's work queue. From our experiments, batching shows a considerable performance improvement over stealing a small number of ready components.

For efficient concurrent execution, the work queue is a lock-free [105] non-blocking queue [153], meaning that the victim and multiple work-

ers can concurrently consume ready components from the queue. Lock-freedom, however, does not imply starvation-freedom [104], which means that during concurrent operations some workers may not make progress. In our case, this is not a practical concern since system-wide throughput is guaranteed and the work stealing terminates in a finite number of steps, therefore, under fair scheduling, all workers will make progress eventually.

A stronger progress condition, which implies starvation-freedom, is wait-freedom [103]. Practical wait-free queues have been introduced [122] recently. We leave their implementation in Kompics to future work.

It is possible that a worker thread blocks while executing an event handler on behalf of a component, which may happen, e.g., when the handler invokes an I/O operation. It may also happen, that an event handler invokes a long-running computation. In such cases, a benefit of work stealing is that the worker's ready components can be stolen by other workers and executed, preventing a blocked or a slow component from indefinitely delaying the execution of other components.

Evaluation

We evaluated the performance of the Kompics work-stealing scheduler against Erlang, the gold standard for concurrent programming [206], using the highly concurrent application, the Game of Life [85]. We modeled cells in the Game of Life as components and we setup eight connections between cells, where connections are modeled as ports connected by channels. We ran the Game of Life program with 100×100 cells for 1,000 generations on a Sun Niagara machine with 2 GB of main memory and six processors, each containing four hardware threads. Kompics ran on Sun's standard edition Java Runtime Environment (JRE) version 6. The goal of the experiment is to compare the speedup [12] of the Kompics version with the speedup of the Erlang version as the number of available processing units is increased.

Figure 4.1 shows the results as we increase the number of workers to take advantage of all 24 hardware processing units. Kompics has a slightly lower speedup compared to Erlang, but note that with the exception of two data points, it is never more than 10% lower. These two outliers were observed at 21 and 22 workers, and were due to increased Java garbage collection (GC) time relative to the experiment running time. For 21 and

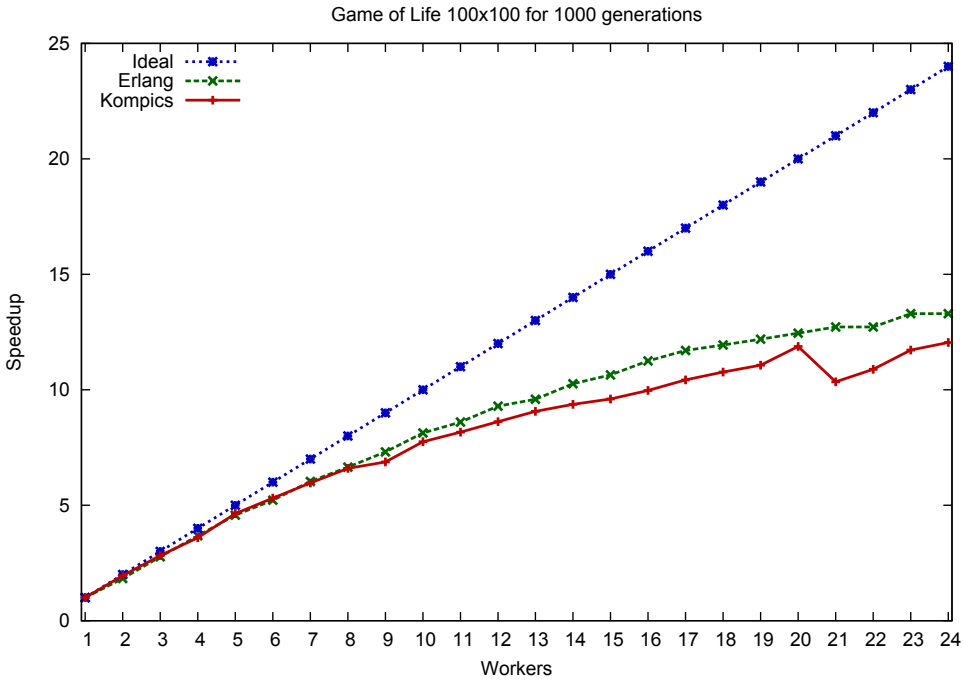

Figure 4.1. Comparing the speedup for an implementation of the Game of Life in both Kompics and Erlang executed on a Sun Niagara machine with six processors each having four hardware threads.

22 workers, the time spent in garbage collection was approximately 16% of the total running time. This is substantially higher than our expected value of 8%, estimated from the garbage collection times observed at neighboring data points. This same garbage collection behaviour was observed repeatedly over different experiment runs. The extra time spent in garbage collection is due to an extra major collection [117, 118] being performed by the Java virtual machine. Note that with a higher number of workers, the total execution time is shorter, resulting in the garbage collection time having a higher relative impact on the results.

Overall, the results show good scalability for the Kompics multi-core work-stealing scheduler, demonstrating that Kompics offers the potential for building scalable, highly concurrent applications for the Java platform.

4.1.2 Discrete-Event Simulation Scheduler

We have also designed a single-threaded scheduler for deterministic component execution which can be leveraged for repeatable simulation. The simulation scheduler executes – in a deterministic manner – all components that are *ready* to execute until there is no more work to be done. At that point, the simulation scheduler yields control to a discrete-event simulator [28] which is in charge of maintaining virtual simulation time and a future event list [116]. When the run-time system starts, the simulation scheduler is configured to use a particular component as the discrete-event simulator (DES) to yield control to. We have implemented a generic discrete-event simulator which we were able to reuse for the whole-system simulation of all P2P systems we have developed, some of which we have described earlier in Section 3.6.

We have taken the following approach to using Kompics for executing real implementations of entire peer-to-peer systems in simulation mode. First, we encapsulate all the protocols implemented by one peer as subcomponents of a composite Peer component, which only requires a Timer and a Network abstraction. Second, we implement a system-specific simulator component that manages multiple peers as its subcomponents and further requires a Timer and a Network abstraction. In a sense, our system-specific simulator component *delegates* the Timer and Network requirements of its Peer subcomponents, to its enclosing environment. And third, a generic and reusable discrete-event simulator is implemented as a component which provides the Timer and Network abstractions for the rest of the system.

Given a system-specific simulation scenario (see Subsection 4.3.2), the generic discrete-event simulator component commands the system-specific simulator to create and destroy peers or to initiate various system-specific operations on the existing peers. This component architecture, illustrated in Figure 4.2, allowed us to execute in simulation entire peer-to-peer networks of tens of thousands of nodes, using the same system implementation designated for production deployment. We were able to achieve this by virtue of reusable component abstractions, hierarchical nested composition, and the dynamic reconfiguration of the component architecture.

We give the full details of the whole-system repeatable simulation mechanism, together with its requirements and limitations, in Section 4.3.

Table 4.1. Time compression effects observed when simulating a peer-to-peer system with various numbers of peers for 4,275 seconds of simulated time.

Simulated peers	Time compression factor	Wall clock time (seconds)
64	475.00	9
128	237.50	18
256	118.75	36
512	59.38	72
1,024	28.31	151
2,048	11.74	364
4,096	4.96	862
8,192	2.01	2,127

Evaluation

We used a P2P simulation architecture, like the one in Figure 4.2, to evaluate the effectiveness of using simulation for studying the dynamic behavior of large-scale P2P systems. As a result of simulation time compression effects, computation time can be traded for simulating larger system sizes.

We ran experiments with the Cyclon overlay network [227] and we were able to simulate a system of 16,384 nodes in a single 64-bit JVM with a heap size of 4 GB. The ratio between the real time taken to run the experiment and the virtual simulated time was roughly one, when simulating the execution of 16,384 peers in one JVM. For smaller system sizes we observed a much higher simulated time compression effect, as illustrated in Table 4.1.

4.2 Scalable Network Communication

Nodes in a Kompics distributed system communicate with each other by sending messages through a Nework service abstraction. The MyNetwork component, shown in all examples of Chapter 2 and Chapter 3, implements the Nework abstraction by marshalling and sending out messages to other nodes over an IP network. It also unmarshalls messages received from remote nodes and delivers them locally to higher-level protocols. The network component is also in charge with establishing and managing network connections to other nodes in the distributed system.

For production deployment mode, the Kompics component library provides three different network component implementations embedding the Apache MINA [185], the Grizzly [218], and the Netty [163] network libraries respectively. Each of these libraries provides an asynchronous event-driven framework designed for building high performance and high scalability network applications using the Java NIO APIs. Because they leverage Java's support for non-blocking I/O, each of these libraries can process a large number of network connections using a small number of I/O processing threads, which enables scalable network communication.

Each of our three network components implements automatic connection management for both TCP and UDP transport protocols and supports pluggable message marshallers; including a built-in object serializer which is useful during prototyping, while message-specific marshallers, for compact binary or text protocols, can be written for production systems. Message serialization is part of a configurable protocol stack where additional message transformation layers can be enabled, e.g., for message compression, encryption, fragmentation. etc. We use the Kryo [124] library for fast and efficient message serialization and deserialization, and Zlib [70] for compression. The choices of which serialization library to use, or enabling message compression, are configurable by Kompics users.

4.3 Whole-System Repeatable Simulation Support

We now show how the same implementation of a distributed system, which is designated for production deployment, is also executable in simulation mode for stepped debugging, protocol correctness testing, or for repeatable studies of the dynamic behaviour of large-scale peer-to-peer systems. Figure 4.2 illustrates a typical component architecture for simulation mode. Here, a generic P2pSimulator interprets an experiment scenario – described in Subsection 4.3.2 – and issues command events to a system-specific simulator component, MySimulator, through its MyExperiment port. An issued command – which is part of the experiment scenario specification – may instruct the MySimulator to create and start a new peer, to stop and destroy an existing peer, or to command an existing peer to execute a system-specific operation by issuing a request through its MyPeerPort.

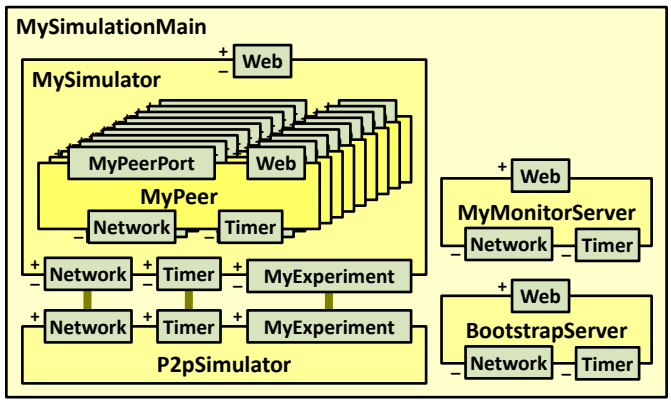

Figure 4.2. Component architecture for whole-system repeatable simulation. All peers and servers execute within a single OS process in virtual simulated time.

The P2pSimulator component provides the Network and Timer abstractions and also implements a generic and reusable discrete-event simulator (DES). This whole component architecture is executed in simulation mode, i.e., using a single-threaded component scheduler which executes all ready components in a deterministic order, and when it runs out of work, it passes control to the P2pSimulator to advance the virtual time and continue the simulation [28], typically by delivering a message to one of the peers.

In order to circumvent nondeterministic execution, when running in simulation mode, the Java bytecode of the system is instrumented to intercept all calls for the current time and return the virtual simulation time. Therefore, without any changes to a system's source code, the system can be executed deterministically in simulated time. JRE code for random number generators (RNG) is also instrumented to use the same RNG seed and preserve determinism. Attempts to create threads are also intercepted and the simulation halts with an error informing the user that deterministic execution cannot be guaranteed. Kompics protocol components are typically reactive and don't spawn threads of their own, so they lend themselves well to simulation. In the whole Kompics component library, the only components that spawn threads of their own are MyTimer, which embeds a Java timer thread, MyNetwork, which embeds a Java NIO network framework, and MyWebServer, which embeds the Jetty web server [219].

The advantage of using bytecode instrumentation for whole-system simulation is that in order to execute a system in simulation there is no need to change any of its source code. The implication of this fact is that we can simulate not only the code of the system under development, but also any third-party libraries that it might use. The only limitation of this approach is when a third-party library invokes native code. Allowing the execution to "escape" the managed environment of the JVM into native code means we loose the guarantee of deterministic execution.

Intercepting calls for the current system time or for thread creation in order to guarantee deterministic execution, could in theory be achieved through the use of custom class-loaders [141]. In practice however, there are technicalities which makes that approach too difficult. Intercepting method calls to `java.lang.System.currentTimeMillis()` requires a custom definition of the `java.lang.System` class which was challenging to provide owing to all its native static methods. Therefore, we resorted to bytecode instrumentation, for which we used the Javassist toolkit [54, 55].

4.3.1 Modeling Network Latency and Bandwidth

A custom network latency model can be used for each simulation experiment. The discrete-event simulator can be configured with a NetworkModel implementation. Code 4.1 shows the interface implemented by a particular network model. For every message sent between two peers, the simulator asks the model what the latency of the message should be, and it delays delivering the message accordingly. The network model generates a message latency based on the source and destination addresses of the message.

Code 4.1 Java interface of a network latency model

```
1 public interface NetworkModel {
2     public long getLatencyMs(Message message);
3 }
```

The Kompics framework provides three NetworkModel implementations. The first provided network model generates uniformly random latencies. The uniform distribution can be parameterized with an interval from which latencies are drawn uniformly. The second provided network model

generates latencies from an exponential distribution. This exponential distribution is also parameterizable with the desired mean latency. The third provided network model generates latencies using the King data set [95] which contains latencies measured between a set of DNS servers.

The Kompics simulation framework also provides a network model which permits the specification of the bandwidth capacity of the network links between peers. We used this network bandwidth model for simulations of the BitTorrent [58] protocol, and it was instrumental in accurately modeling bandwidth queuing delay and network congestion, for content distribution through chunk data transfers. Upload and download bandwidth capacities are specified for every peer. When a message, carrying a data block of a specified size, is sent from one source peer to a destination peer, the message is first subject to the bandwidth queuing delay corresponding to the upload link of the source peer and then it is subject to the bandwidth queuing delay corresponding to the download link of the destination peer. Optionally, the message can be subject to additional delay according to a network latency model such as the ones we described above.

Each link is modeled as a queue of messages. When a new message arrives at the link, a queuing delay is computed for the message, based on the link's capacity, the size of the message, and the total size of the messages currently enqueued in the link. The computed queuing delay determines the exit time of the message, i.e., the time when it will be dequeued from the link and sent forward. Network congestion is accurately modeled since the bandwidth queuing delay for each message is computed as a function of all other messages that are traversing a particular link at the same time.

4.3.2 Specifying Experimentation Scenarios

We designed a Java domain-specific language (DSL) for expressing experiment scenarios for P2P systems. Such experiment scenarios can be interpreted, e.g., by a discrete-event simulator (DES) like our P2pSimulator. We now give a brief description of our DSL with a simple example scenario.

A scenario is a parallel and/or sequential composition of *stochastic processes*. We call a stochastic process, a finite random sequence of events, with a specified distribution of inter-arrival times. Code 4.2 shows an example stochastic process. This will generate a sequence of 1,000 *join*

Code 4.2 Stochastic process for bootstrapping a peer-to-peer system

```
1 StochasticProcess boot = new StochasticProcess() {
2     {
3         // exponentially distributed, μ = 2s
4         eventInterArrivalTime(exponential(2000));
5         // 1000 joins with uniform IDs from 0..2^16
6         raise(1000, join, uniform(0, 65536));
7     }
8 };
```

Code 4.3 Defining a simulation operation with one parameter

```
1 Operation1<Join, BigInteger> join =
2         new Operation1<Join, BigInteger>() {
3     public Join generate(BigInteger nodeKey){
4         return new Join(new NumericRingKey(nodeKey));
5     }
6 };
```

operations, with an inter-arrival time between two consecutive operations extracted from an exponential distribution with a mean of two seconds. The *join* operation is a system-specific operation with one parameter. In this case, the parameter is the Chord [208] identifier of the joining peer, extracted from an uniform distribution of $[0..2^{16}]$. Code 4.3 shows how the *join* operation is defined. It takes one BigInteger argument (extracted from a distribution) and generates a Join event (triggered by the P2pSimulator on MyPeerPort). In Code 4.4 we define a *churn* process which will generate a sequence of 1,000 churn events (500 joins randomly interleaved with 500 failures), with an exponential inter-arrival time with a mean of 500 ms.

Code 4.4 Stochastic process regulating churn in a peer-to-peer system

```
1 StochasticProcess churn = new StochasticProcess() {
2     {
3         // exponentially distributed, μ = 500ms
4         eventInterArrivalTime(exponential(500));
5         raise(500, join, uniform(16));  // 500 joins
6         raise(500, fail, uniform(16));  // 500 failures
7     }
8 };
```

Code 4.5 Stochastic process regulating lookup operations

```
1 StochasticProcess lookups = new StochasticProcess() {
2     {
3         // normally distributed, μ = 50ms, σ = 10ms
4         eventInterArrivalTime(normal(50, 10));
5         raise(5000, lookup, uniform(16), uniform(14));
6     }
7 };
```

In Code 4.5 we define a process to issues some Lookup events. The *lookup* operation takes two BigInteger parameters, extracted from a (here, uniform) distribution, and generates a Lookup event that tells MySimulator to issue a lookup for key *key* at the peer with identifier *node*.

As shown in Code 4.6, a random peer in range $0..2^{16}$ will issue a lookup for a random key in rage $0..2^{14}$. Five thousand lookups are issued in total, with an exponentially distributed inter-arrival time with a mean of 50 ms.

Code 4.6 Defining a simulation operation with two parameters

```
1 Operation2<Lookup, BigInteger, BigInteger> lookup
2       = new Operation2<Lookup, BigInteger, BigInteger>() {
3   public Lookup generate(BigInteger node,BigInteger key) {
4       return new Lookup(new NumericRingKey(node),
5                         new NumericRingKey(key));
6   }
7 };
```

We have defined three stochastic processes: *boot*, *churn*, and *lookups*. Putting it all together, Code 4.7 shows how we can compose them to define and execute an experiment scenario using our Java DSL. The experiment scenario starts with the *boot* process. Two seconds (of simulated time) after this process terminates, the *churn* process starts. Three seconds after *churn* starts, the *lookups* process starts, now working in parallel with *churn*. The experiment terminates one second after all lookups are done.

Note that Code 4.7 contains an executable Java main-class. It creates a *scenario1* object, sets an RNG seed, and calls the *simulate* method passing the simulation architecture of the studied system as an argument (line 16).

Code 4.7 A complete experiment scenario definition

```
class SimulationExperiment {
    static Scenario scenario1 = new Scenario() {
        StochasticProcess boot = ...  // see above
        StochasticProcess churn = ...
        StochasticProcess lookups = ...
        boot.start(); // scenario starts with boot process
        // sequential composition
        churn.startAfterTerminationOf(2000, boot);
        // parallel composition
        lookups.startAfterStartOf(3000, churn);
        // join synchronization
        terminateAfterTerminationOf(1000, lookups);
    }
    public static void main(String[] args) {
        scenario1.setSeed(rngSeed);
        scenario1.simulate(SimulationMain.class);
    }
}
```

4.4 Testing and Debugging Distributed Systems

We leverage whole-system repeatable simulation for testing the correctness of distributed protocols. Given that the execution of a particular experiment scenario is deterministic, and it occurs in a single JVM, we can encapsulate the simulation of a complete scenario in a unit test. This approach allows us to define an entire test suite for a particular protocol, comprising a set of unit tests, one for each simulation scenario. Typically, we define one scenario for each kind of execution we want to subject a protocol to. For example, one can define scenarios with various combinations of concurrent churn events like nodes joining the system or failing. The test could then validate that certain reconfiguration protocols in the system are executed correctly, e.g., by satisfying their safety invariants and terminating.

We applied this approach in the context of developing CATS, our case-study key-value store. In Section 8.4 we show how we leveraged repeatable whole-system simulation to test the correctness of the CATS reconfiguration protocols. We devised 135 distinct scenarios that covered all types of churn situations that the system should handle correctly for a replication degree

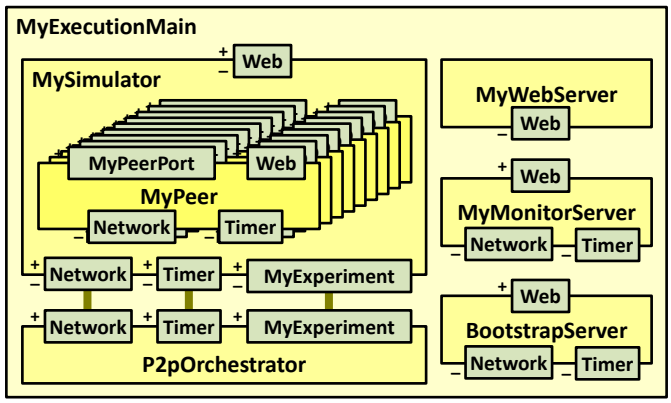

Figure 4.3. Component architecture for whole-system interactive stress testing. All peers and servers execute within a single OS process in real time.

of five. This set of scenarios doubled as a regression test suite, giving us confidence that the reconfiguration protocols continued to work correctly across all scenarios as we made small changes to them.

Whole-system repeatable simulations can also be leveraged for stepped debugging. In particular, when using a modern IDE, one can set conditional breakpoints and state watches such that the stepped debugging stops when the execution reaches a particular system configuration specified as a conditional breakpoint. When a particular unit test fails, stepped debugging can be used to quickly find the root cause of the problem.

4.5 Interactive Stress Testing

Using the same experiment scenario devised for whole-system simulation, the same system code can be executed in an interactive execution mode for stress testing. In Figure 4.3 we show the respective component architecture. This is similar to the simulation architecture, however, we use our regular multi-core component scheduler and the system executes in real-time, albeit driven from the same experiment scenario. The P2pSimulator was replaced with a P2pOrchestrator which provides the Network and Timer abstractions and drives the execution from a specified experiment scenario. The P2pOrchestrator can also be configured with a custom network model.

During development it is recommended to incrementally make small changes and quickly test their effects. The interactive execution mode helps with this routine since it enables the programmer to quickly run a small or medium-scale distributed system – without the need for deployment or manual launching of multiple processes – and to interact with it, and also to conveniently monitor its state using a web browser.

In interactive stress test execution mode, experiments are run in real physical time, i.e., in the special case where simulation time is equivalent to physical system execution time. This allows us to use multiple worker threads for executing experiments, without needing to synchronize the workers on the passage of simulation time [50]. The use of real time means that events may not execute at the expected time due to queuing delays. However, most distributed systems, and all P2P systems, are tolerant to messaging delays within some application-specific bounds.

Lin et al. showed [140] that this approach is valid to the extent that the delay of events in queues does not affect application invariants. Application invariants are properties of the application that must be maintained over all execution runs. For P2P systems, application invariants can be specified as conditions on the logic of timers [140]. For example, an RPC response event cannot be delayed for an amount of time exceeding its expiration time, otherwise it would time out before it could be handled, potentially breaking some application invariant. In Kompics experiments running on a single multi-core machine, events will encounter increasing queuing delays with increasing system load. Event queuing delays occur if the system generates more events than it can process over a period of time. Using an implementation of the Cyclon overlay [227], in the next two sections we investigate how large the system can grow – for different numbers of cores and machines – while conservatively maintaining timing invariants. That is, we have to keep the highest event queuing delays considerably below the minimum configured timeout period in the Cyclon protocol.

4.5.1 Scalability of Local Stress Testing

We evaluated the scalability of our stress test execution mode for multi-core hardware by running a P2P experiment scenario on an increasing number of processing cores. Our hardware setup comprised of a Mac Pro

machine with two quad-core 2.8 GHz Intel Xeon E5462 CPUs, Windows XP 32bit, and the Sun Java server VM version 1.6.0 update 7 with a heap size of 1,426 MB using a parallel garbage collector (GC). We executed the experiment scenario using 1, 2, 4, and 8 Kompics worker threads.

We first experimented with an implementation of the Cyclon random overlay [227] and our expectations were as follows. As the size of the P2P system under test (SUT) is increased, more components are created in the system leading to an increased flow of events passed between components. With bounded processing power and an increased number of events in the system, we expect that each event will experience a larger queuing delay before being processed. If the event queuing time exceeds a certain application-specific threshold, timing-related invariants of the SUT may be broken. Therefore, to increase the accuracy of our stress test experiments, we make sure that event queuing time is bounded.

We implemented the Cyclon random overlay as Kompics components, as shown in Figure 3.10 and Figure 3.13, and we ran it in a stress test component architecture similar to the one in Figure 4.3. In Cyclon, the essential timing-related invariant is that every peer gossips with one other peer in each cycle [227]. This invariant may be broken if events are delayed for longer than the gossip period, leading to inaccurate observations on the overlay's dynamic behaviour and its emergent properties. Each peer contains four protocol components: Cyclon, BootstrapClient, PeerMonitor, and WebApplication. We bootstrapped the system with 2,000 peers gossiping every 10 seconds and we measured the event queuing time for every event in the system. We continued to join 2,000 peers at a time until the 99^{th} percentile of the event queuing time exceeded the 5 seconds threshold.

In Figure 4.4 we plot the 99^{th} percentile of event queuing time for all events in the system. As expected, we can see that for increasingly larger system sizes, the event queuing time increases. We can also observe that even for 20,000 peers, for 99% of the events in the system, the observed queuing delay is less than 5 seconds, which constitutes half of the Cyclon cycle period, thus preserving the main SUT timing invariant.

Regarding the scalability of the local stress test execution mode, we can see that event queuing times are consistently lower when a SUT with the same number of peers is executed using an increased number of processing cores. Although the 99^{th} percentile of the event queuing time for 20,000

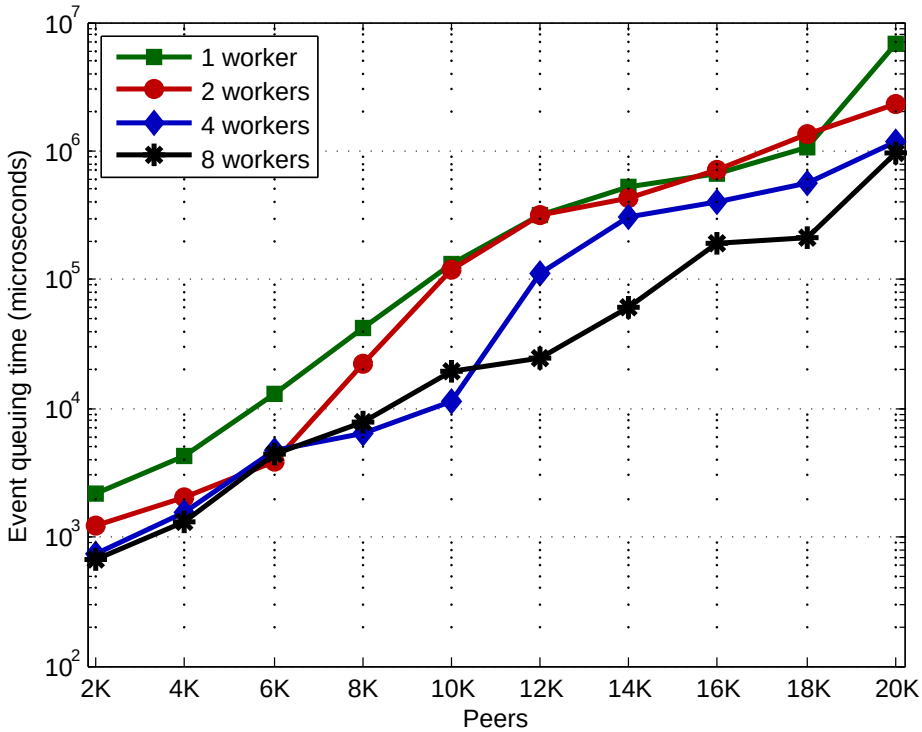

Figure 4.4. The 99th percentile of event queuing time as a function of system size in a whole-system stress test of Cyclon [227] executed on a single multi-core machine while utilizing a varying number of processing cores.

peers is five seconds when using one processing core, it drops to under one second when using eight cores. In conclusion, we can use a single multi-core machine to run an accurate local stress testing experiment of a P2P system with 20,000 peers, without breaking the SUT's timing invariants.

4.5.2 Scalability of Distributed Stress Testing

When the amount of available main memory (RAM) becomes a bottleneck to scaling the size of the systems under test (SUT), the way to further scale the stress testing experiments is to distribute them across a number of machines, e.g., using a LAN testbed like ModelNet [223] or Emulab [231], or even a wide area network (WAN) testbed like PlanetLab [33, 179].

Given our component execution model based on message-passing communication, distributing the real-time execution of interactive stress test experiments over a network is straightforward. We build a *distributed stress testing* architecture as follows [14]. We take the P2pOrchestrator component which drives the local execution of an experiment scenario – shown in Figure 4.3 – and we split its functionality into a Master and a set of Slave components, whereby each machine in the testbed hosts one Slave component. The Master drives the experiment scenario and it coordinates with the remote Slaves using a Network abstraction. The Master is in charge of allocating newly joined peers to different Slaves, such as to balance the execution load. Each Slave drives a part of the experiment and it manages the peers executing locally on its own host machine. The Master is hosted on a separate machine and so are the bootstrap and the status monitoring servers in order to maximize the system's scalability.

We have used the Cyclon implementation described in the previous section to investigate the extent to which the size of the P2P system under stress test can be scaled by distributing the experiment over multiple machines. We executed the Cyclon stress test experiment on a set of 10 IBM server blades each having two hyper-threaded 3 GHz Intel Xeon CPUs using SMP Linux 2.6.24-19-server and the Sun Java server VM version 1.6.0 update 7, with a heap size of 2,698 MB using a parallel garbage collector (GC). We used two Kompics worker threads on each machine.

We bootstrapped the system with 1,000 peers on each machine, gossiping every 10 seconds and we measured the queuing time of all events in the system for a duration of 30 seconds. We continued to join 1,000 peers at a time, on each machine, and we measured the event queuing time for 30 seconds at each step. We stopped joining peers once we reached 9,000 peers per machine for a total system size of 90,000 Cyclon peers.

We plot the measured event queuing times in Figure 4.5. The results show that we can simulate around 40,000 Cyclon peers while 99% of all events in the system are not delayed by more than 300 milliseconds. This fares better than the roughly 16,000 Cyclon peers executed a single host with two cores – illustrated in Figure 4.4 – albeit running on higher performance hardware. This demonstrates the Kompics distributed stress testing mode's potential for increasing the scalability of experiments by using additional machines within a LAN cluster or even within a WAN testbed.

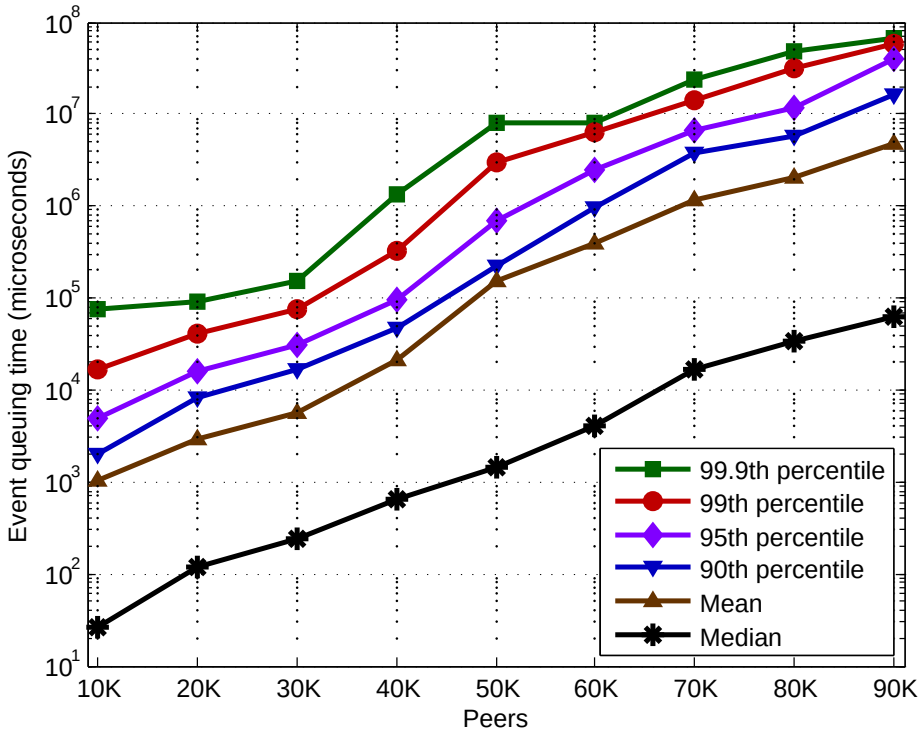

Figure 4.5. Event queuing time as a function of system size in a distributed stress test experiment executed on 10 cluster machines in a local area network (LAN).

4.5.3 Analysis

The results here show that the stress test mode of executing Kompics P2P systems enables experiments to scale in the number of peers, while maintaining bounded event queuing time. Experiments can be scaled simply by adding more CPU cores to a host or by adding additional hosts to an experimental testbed. Experiment runs introduce minor variations in the order of processing events, caused by worker scheduling and hosts running in parallel without agreement on the passage of physical time. In agreement with Lin et al. [140], we argue that these minor variations are useful when stress testing distributed systems, as they model types of event processing delays that can be expected in production systems, such as those caused by network faults or congestion. As such, our stress testing mode

provides a useful stage in the development of P2P systems, in that it enables the testing of larger-scale systems in a more challenging environment. This stage of testing for production systems could complement traditional stress testing stages, by helping to build large-scale experiment scenarios that are able to identify unexpected behaviors that only arise at large system sizes.

Compared to simulation, two drawbacks of real-time execution are that we cannot take advantage of the time-compression effects of time-stepped simulators, and experiments are not repeatable – although distributed executions in production are not reproducible either. On the other hand, the benefit of running in real time is improved scalability, since we avoid the cost of simulation controllers agreeing on the passage of virtual time [50].

4.6 Incremental Development and Testing Support

The Kompics Distributed System Launcher is a Java utility and a set of Kompics components that developers can use to quickly experiment with small-scale distributed systems implemented in Kompics, locally on the development machine. The user specifies a network topology and an execution script for each of the processes in the distributed system. We call the set of process execution scripts, the scenario. The distributed system scenario is then executed on a specified network topology.

Code 4.8 shows an example topology specification containing the characteristics of the network connecting the processes of the distributed system. The code creates a network topology, called `topology3`, with six nodes or processes. The process running at node 1 shall be receiving messages at the network address `127.0.0.1:22031`, i.e., process 1 binds TCP and UDP ports 22031 on the local host. Similarly, process 2 shall be receiving messages at network address `127.0.0.1:22032`.

The `link(1, 2, 1000, 0.5)` specifies a directed link from node 1 to node 2, which we denote by $1 \rightarrow 2$. This link has a latency of 1,000 ms and a drop rate of 0.5. This means that messages sent by node 1 to node 2 are delayed by 1,000 ms and on average 50% of the messages are dropped or lost. Note that this link is directed, so without specifying other links, node 2 could not send messages to node 1. Alternatively, we say that 2 is a neighbor of 1 but 1 is not a neighbor of 2. We can make the link

Code 4.8 A simple topology for local interactive system execution

```
1 Topology topology3 = new Topology() {
2     {
3         node(1, "127.0.0.1", 22031);
4         node(2, "127.0.0.1", 22032);
5         node(3, "127.0.0.1", 22033);
6         node(4, "127.0.0.1", 22034);
7         node(5, "127.0.0.1", 22035);
8         node(6, "127.0.0.1", 22036);
9         link(1, 2, 1000, 0.99);
10        link(3, 4, 2000, 0.05);
11        link(5, 6, 500, 0.1).bidirectional();
12        defaultLinks(100, 0);
13    }
14 };
```

bidirectional by writing `link(1, 2, 1000, 0.5).bidirectional()`. This will also create a link from node 2 to node 1 having the same latency and loss characteristics as link $1 \rightarrow 2$. An example of such a link is specified between nodes 5 and 6: `link(5, 6, 500, 0.1).bidirectional()`. When the $2 \rightarrow 1$ link has different latency and loss characteristics from the $1 \rightarrow 2$ link, we need to add an extra `link(...)` statement.

The `defaultLinks(100, 0)` statement of line 12, "fills in" the missing links to create a fully connected topology. Here, all added links have a latency specification of 100 ms and 0 message loss rate. In the absence of line 12, `topology3` would not be fully connected, and node 4 would not be able to communicate with node 5 for example.

A scenario is a set of process execution scripts. A process execution script is a sequence of commands that an Application component will execute at the process. Code 4.9 shows an example scenario.

Code 4.9 A simple local experiment scenario with two processes

```
1 Scenario scenario1 = new Scenario(Main.class) {
2     {
3         command(1, "S500:Lmsg1:S1000:X");
4         command(2, "S1000:Pmsg2:S1000:X");
5     }
6 };
```

This means that process 1 will execute script "S500:Lmsg1:S1000:X" and process 2 will execute commands "S1000:Lmsg2:S1000:X". The S500 command means that the process waits (sleeps) for 500 ms before executing the next command. Command Lmsg1 means that the process sends message msg1, over lossy links to all its neighbors. Sending a message over a lossy link means that the message may be dropped with a probability equal to the loss rate of the link specified in the topology. Command Pmsg2 means that the process sends message msg2, over perfect links to all its neighbors. Sending a message over a perfect link means that the message is delivered at the destination exactly once. Both lossy and perfect links will delay messages sent over them, according to their latency; specified in the network topology on which the scenario is executed.

Command X terminates the process. In summary, process 1 will start, wait 500 ms, send out message msg1 over lossy links to all its neighbors, wait one second, and then terminate. Process 2 will start, wait one second, send out message msg2 over perfect links to all its neighbors, wait one second, and then terminate.

In scenario1, Main.class represents the Kompics component definition of a root component that creates and initializes the protocol components in one process of the distributed system under experiment. The Main component class contains a Java main method that starts the Kompics run-time system and then creates and starts the Main component.

A scenario is therefore constituted by a particular distributed system and the commands its processes are supposed to execute. The distributed system is specified by the main component that defines the software architecture of each process in the system, e.g., Main in scenario1.

Topology and scenario definitions are written as Java anonymous classes. To execute a scenario on a particular topology, the user places the topology and scenario definitions within a Java program as illustrated in Code 4.10.

The statement on line 16 executes scenario1 on topology1. Processes 1 and 2 are launched at the same time. Each process executes the program Main which is a main Java class and a root Kompics component, which, at minimum, creates and initializes a network component, a timer component, and an application component. The application component in each process interprets and executes the process execution script specified

Code 4.10 An experiment scenario for quick local interactive testing

```
public final class Experiment1 {
    public static final void main(String[] args) {
        Topology topology1 = new Topology() {
            {
                node(1, "127.0.0.1", 22031);
                node(2, "127.0.0.1", 22032);
                link(1, 2, 1000, 0).bidirectional();
            }
        };
        Scenario scenario1 = new Scenario(Main.class) {
            {
                command(1, "S500:Lmsg1:S1000:X");
                command(2, "S1000:Pmsg2:S1000:X");
            }
        };
        scenario1.executeOn(topology1);
    }
}
```

in `scenario1`. Users can extend the application component with new commands besides the default `S`, `L`, `P`, and `X`, introduced above.

By calling `scenario1.executeOnFullyConnected(topology1)` the distributed system launcher is instructed to check that `topology1` is fully connected, and to quit with an error message if it isn't.

When `Experiment1` is executed as a Java program, the distributed system launcher creates a graphical user interface (GUI) window for each process. The process output is logged in this window, and the user can input further commands to the application component of the process. Figure 4.6 shows what happens when `Experiment1` is executed.

The different windows allow users to see the output of each process simultaneously which may facilitate debugging certain dynamic behaviours which are now more easily observed than by inspecting process logs. If the user closes the window of a process, or inputs command `X`, that process terminates. Users can also terminate the currently focused process by pressing Ctrl+Z. All processes can be killed at once by pressing Ctrl+K.

As illustrated in Figure 4.6, the output of each process is timestamped. Each window contains both the output of its own process and the output

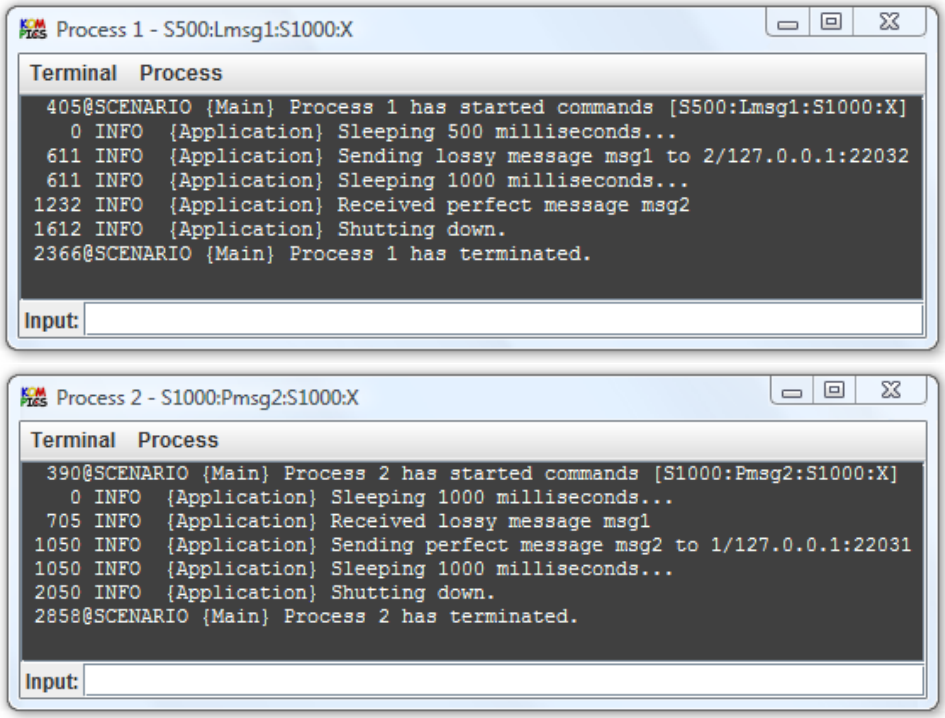

Figure 4.6. Screenshot of local execution for quick incremental testing.

of the distributed system launcher itself. The log timestamps followed by the word `INFO` in this example, come from the application component of the process, which uses log4j, and they are relative to the process start time. The log timestamps followed by the string `@SCENARIO` come from the distributed system launcher and they are relative the launcher start time. In this example, we may roughly estimate that process 1 started around 15 ms after process 2 started. This difference in the start-up times of the processes depends on the operating system load.

So far we discussed how to execute a distributed system in a *crash-stop* model, whereby processes which terminate and killed processes do not restart. If developers wish to experiment with a distributed system in a *crash-recovery* model, whereby processes may crash and later recover, they can create a crash-recovery scenario like the one defined in Code 4.11.

Code 4.11 An experiment scenario supporting the crash-recovery model

```
1 Scenario scenario2 = new Scenario(Main.class) {
2     {
3         command(1, "S500:Lmsg1:S1000:X").recover(
4                    "S500:Pmsg3:S500:X", 2000);
5         command(2, "S1000:Pmsg2:S6000:X");
6     }
7 };
```

The `recover("S500:Pmsg3:S500:X",2000)` statement in lines 3–4 means that process 1 should recover 2000 ms after it terminates, and then execute commands `S500:Pmsg3:S500:X`. When we execute `scenario2` on `topology1` we get the results presented in Figure 4.7.

Notice that process 2, which was alive for about 7 seconds, receives from process 1, both message `msg1`, sent in the first *incarnation* of process 1, and message `msg3`, sent by process 1 in its second incarnation. We can also notice that log timestamps were reset for process 1's second incarnation. Considering the crash-recovery scenario defined in Code 4.11, its worth noting that if process 1 is killed while sleeping, i.e., before it gets a chance to execute its first `X` command, process 1 is still going to be recovered.

Users can also manually recover a dead process. To recover a crashed process, one types `recover@pid@command` in the console of the Experiment process, i.e., the distributed system launcher, where `pid` represents the identifier of the process to be recovered, and `command` is the script that the process should execute upon recovery.

4.7 Implementation in Different Programming Languages

We used the Java programming language for the reference implementation of Kompics. Meanwhile, in an attempt to reduce the verbosity of the Java specification of Kompics events, ports, and components, and also to cater for more programmer audiences, Kompics has been ported to Scala and also to Python. The following sections review these implementations and show some examples of Kompics concepts programmed in these languages.

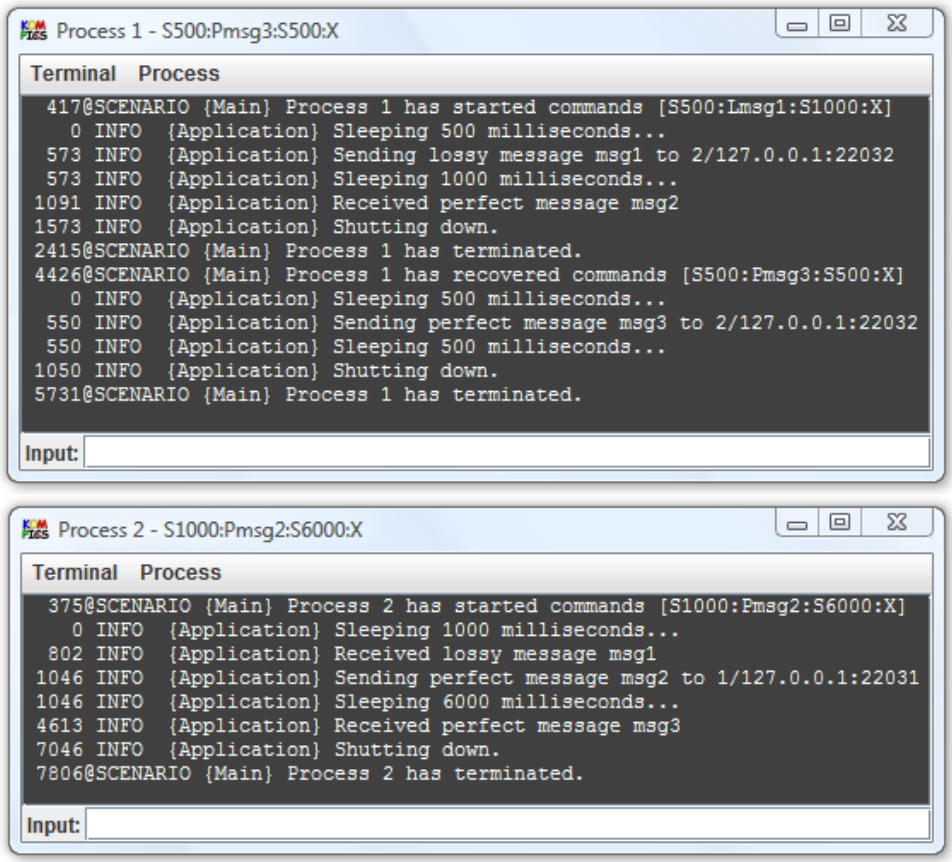

Figure 4.7. Screenshot of local execution with crash-recovery support.

4.7.1 Scala

A Scala adaptation of the Java implementation of Kompics was contributed by Lars Kroll. The Scala programming language facilitates the implementation of domain-specific languages (DSLs) and this support was leveraged into designing a very succinct expression of the Kompics concepts and constructs in Scala. The Kompics Scala front-end automatically leverages the existing Java components and the run-time system, since Scala code compiles to Java bytecode and executes on the JVM. As a result, we can build distributed systems by seamlessly composing Java and Scala components.

Code 4.12 A simple event type definition in Scala

```
case class Message(source: Address, destination: Address)
        extends Event
```

Code 4.13 A derived event type definition in Scala

```
case class DataMessage(source: Address,
                       destination: Address,
                       data: Data,
                       sequenceNumber: int)
        extends Message(source, destination)
```

We now show a few examples of Kompics concepts implemented in Scala. Code 4.12 and Code 4.13 illustrate event type definitions, analogous to the Java definitions presented in Code 2.1 and Code 2.2, respectively.

Similarly, Code 4.14 and Code 4.15 show Scala port type definitions analogous to the Java ones presented in Code 2.3 and Code 2.4, respectively.

Code 4.16 illustrates a simple Scala component, analogous to its Java counterpart shown in Code 2.6. A required port is specified using the **--** operator. Conversely, a provided port is specified using the **++** operator. Event handlers are anonymous and implicitly subscribed to ports using the `uponEvent` operator. Event deliveries to the component leverage Scala's pattern matching mechanism and multiple event types can be matched within the same `uponEvent` construct using multiple `case` statements.

Code 4.14 A Network port definition in Scala

```
object Network extends PortType {
    request(Message);
    indication(Message);
}
```

Code 4.15 A Timer port definition in Scala

```
object Timer extends PortType {
    request(ScheduleTimeout);
    request(CancelTimeout);
    indication(Timeout);
}
```

Code 4.16 A simple component definition in Scala

```
1 class MyComponent extends ComponentDefinition {
2     val service = ++ (MyService);   // provided port
3     val network = -- (Network);     // required port
4     var messages : int = 0;
5     network uponEvent {
6         case Message(source, destination) => { () =>
7             messages++;
8             println("Received from " + source);
9         }
10    }
11 }
```

Kompics types written in Java need to be referenced as `classOf[Type]` in `request`, `indication`, **++**, **--**, and `create` statements in Scala code.

Code 4.17 shows a root component definition in Scala, analogous to its Java counterpart from Code 2.7. Interestingly, connections between ports are expressed more succinctly than in Java using the **++** and **--** operators. The statement `timer` **++** `Timer` **--** `fd` means: connect the provided Timer port of the `timer` component to the required Timer port of the `fd` component. The next statement in line 8 creates two channels at the same time connecting the provided Network port of the `network` component to the required Network ports of the `fd` and `broadcast` components respectively.

Code 4.17 A root component definition in a Scala executable program

```
1 object Main extends ComponentDefinition {
2     val network = create(MyNetwork);
3     val timer = create(MyTimer);
4     val fd = create(FailureDetector);
5     val broadcast = create(ReliableBroadcast);
6
7     timer ++ Timer -- fd;
8     network ++ Network -- (fd, broadcast);
9
10    def main(args: Array[String]): Unit = {
11        Kompics.bootstrap(Main);
12    }
13 }
```

Code 4.18 Defining Kompics events in Python

```
"""A simple event type definition."""
Message = event('Message', 'source, destination', Event)

"""A derived event type definition."""
DataMessage = event('DataMessage',
                    'source, destination, data, seqNo',
                    Message, 'source, destination')
```

Code 4.19 A Network port definition in Python

```
class Network(Port):
    requests = [Message]
    indications = [Message]
```

Code 4.20 A Timer port definition in Python

```
Timer = port('Timer',
             [ScheduleTimeout, CancelTimeout],  # requests
             [Timeout])                         # indications
```

4.7.2 Python

Kompics was also ported to the Python programming language by Niklas Ekström. Again, the motivation for Kompics Python was a more succinct expression of Kompics programs as well as potential adoption by more users and students familiar with Python. Kompics Python has its own run-time system and there is no code directly shared with the Java implementation.

We now show a few examples of Kompics concepts implemented in Python. Code 4.18 illustrates event type definitions, equivalent to the Java event type definitions presented in Code 2.1 and Code 2.2. Similarly, Code 4.19 and Code 4.20 show Python port type definitions analogous to the Java ones presented in Code 2.3 and Code 2.4, respectively.

Code 4.21 A simple event handler in Python

```
def handleMessage(self, event):
    self.messages += 1  # component-local state update
    print "Received from ", event.source
```

Code 4.22 A simple component definition in Python

```
class MyComponent(Component):
    def __init__(self):
        Component.__init__(self)
        self.network = self.requires(Network)
        print "MyComponent created."
        self.messages = 0
        self.subscribe({self.network : Message})

    def handleMessage(self, event):
        self.messages += 1  # component-local state update
        print "Received from ", event.source
```

Code 4.21 illustrates a Python event handler, analogous to its Java counterpart from Code 2.5. Code 4.22 shows a simple component and Code 4.23 shows a root component definition in Python, both similar to their respective Java counterparts illustrated in Code 2.6 and Code 2.7.

Although certain Kompics Python definitions – such as those of event types and port types – are more compact than in Java, component definitions, on the other hand, appear to be cluttered with `self` references.

Code 4.23 A root component definition in a Python executable program

```
class Main(Component):
    def __init__(self):
        Component.__init__(self)

        self.network = self.create(MyNetwork)
        self.timer = self.create(MyTimer)
        self.fd = self.create(FailureDetector)

        self.channel1 = self.connect(
                self.network.provided(Network),
                self.fd.required(Network))
        self.channel2 = self.connect(
                self.timer.provided(Timer),
                self.fd.required(Timer))

scheduler = WorkStealingScheduler()
scheduler.bootstrap(Main)
```

4.8 Programming in the Large

We used Apache Maven [184] to organize the structure and manage the artifacts of the reference Kompics implementation [19]. The complete framework counts more than a hundred modules.

We organized the various framework constituents into *abstraction* and *component* packages. An abstraction package contains a Kompics port together with the request and indication events of that port. A component package contains the implementation of one component together with some component-specific events – typically subtypes of events defined in required ports. The source code for an abstraction or component package is organized as a Maven module and the binary code is packaged into a Maven artifact, a JAR archive annotated with meta-data about the package's version, dependencies, and pointers to web repositories from where (binary) package dependencies are automatically fetched by Maven.

In general, abstraction packages have no dependencies and component packages have dependencies on abstraction packages for both their required and provided ports. This is because a component implementation will use event types defined in abstraction packages, irrespective of the fact that an abstraction is required or provided. This approach of explicit binary dependencies also enables deploy-time composition.

Maven enables true reusability of protocol abstractions and component implementations. Users can start a project for a new protocol implementation and just specify what existing abstractions their implementation depends on. These are automatically fetched and made available in the new project. For convenience, our Kompics repositories [19] also contain JavaDoc and source JARs which are also fetched automatically by Maven. This enables programmers to seamlessly navigate the source code of the artifacts their projects depend on, thus enhancing developer productivity.

Chapter 5

Kompics Discussion and Comparison to Related Work

We have presented the Kompics component model and its programming framework and patterns. We showed how complex distributed systems can be built by composing simple protocol abstractions. Distributed protocols are programmed as event-driven, message-passing concurrent components. Kompics contributes a unique combination of features well suited for the development and testing of large-scale, long-lived distributed systems.

Central to the design philosophy of Kompics are the principles of nested hierarchical composition, message-passing concurrency, publish-subscribe asynchronous component communication, dynamic reconfiguration, and the ability to run the same code in either production mode or in repeatable whole-system simulation for testing, debugging, and behaviour evaluation.

Systems built with Kompics leverage multi-core machines out of the box and they can be dynamically reconfigured to support hot software upgrades. Kompics provides a simple event-driven style of programming, which lends itself naturally to expressing distributed protocols, and due to asynchronous component interaction, it enables the construction

of non-blocking, highly concurrent systems. The same system software designated for production deployment can be executed in reproducible simulation mode without changing its source code, meaning that third-party binary libraries can also be leveraged in simulation. These properties of the framework, together with a rich library of provided protocols and abstractions, led to Kompics being used for prototyping, evaluating, and developing a diverse collection of distributed systems, such as a P2P video on demand system [37], a secure and fault-tolerant distributed storage system [111], NAT-aware peer sampling protocols [73, 172], P2P live media streaming systems [170, 174, 171, 173, 176], a locality-aware scalable publish-subscribe system [187], scalable NAT-traversal protocols [164], distributed hash-table replication schemes [200], gossip protocols for distribution estimation [175], an elasticity controller simulator [162, 161], studies of multi-consistency-model key-value stores [7, 41], mechanisms for robust self-management [6, 22], and a reliable UDP transport mechanism [157]. We have been using Kompics as a teaching framework in two courses on distributed systems, for more than five years. Students used Kompics successfully, both to deliver running implementations of complex distributed systems, and to gain insights into the dynamic behavior of those systems.

Kompics blends message-passing concurrency from actor models, with nested hierarchical composition from component models, with explicit component dependencies from architecture description languages (ADLs). Consequently, Kompics is related to work in several areas including: concurrent programming models [21, 206], reconfigurable component models [44, 63, 165], reconfigurable software architectures [144, 152, 65, 8], protocol composition frameworks [168, 156, 84, 224, 180, 101] and event-based frameworks for building distributed systems [123, 230, 109].

5.1 Message-Passing Concurrency and Actor Models

The message-passing concurrency model employed by Kompics is similar to the actor model [2], of which Erlang [20, 21], the POSIX process and pipe model, Kilim [206], and Scala [165] are perhaps the best known examples.

Similar to the actor model, message passing in Kompics involves buffering events before they are handled in a first-in first-out (FIFO) order, thus

decoupling the thread that sends an event from the thread that handles an event. In contrast to the actor model, event buffers are associated with Kompics component ports, thus each component may have more than one event queue, and ports are connected using channels.

Channels that carry typed messages between processes are also found in other message-passing systems, such as Singularity [78]. Connections between processes in actor models are unidirectional and based on process identifiers, while channels between ports in Kompics are bidirectional and components are oblivious to the destination of their events. In Kompics, ports may be connected to potentially many other components enabling a publish-subscribe communication pattern.

Oblivion to the identity of other components largely circumvents the issue of circular dependency when two processes need to communicate and neither one knows the other's identity. But perhaps most importantly, this leads to loose coupling [34] which is a crucial ingredient for the dynamic reconfiguration of the component architecture. Restricting communication to occur only through ports prevents situations where it would be unsafe to destroy components because there could exist direct references to them elsewhere in the component architecture.

Kompics supports event filters on channels and subscriptions on ports, while in actor models event filtering is performed using pattern matching at processes. Pattern matching code can make Erlang processes and Scala actors less reusable than Kompics components. Erlang and Scala actors execute pattern matching on messages in their mailboxes to find a matching execution handler, while in Kompics, components subscribe event handlers to event types received over ports, with optional attribute value filtering on channels. For one-to-many connections, like in our P2P simulations, messages are filtered in channels by their destination peer address attribute, meaning they will only be delivered to the subscribers with matching attribute filters. In Erlang and Scala, all messages will be delivered to the mailboxes of all processes who would then filter the messages locally.

To support RPCs, Kompics provides the *expect* command which is similar to the blocking *receive* commands in Erlang and Scala.

Kompics is also related to the Rust [220] programming language from Mozilla, and the Go [217, 27] programming language from Google, which both support lightweight processes communicating through messages.

5.2 Reconfigurable Component Models

In previous work on dynamically reconfigurable middleware, component models developed mechanisms such as explicit dependency management, component quiescence, and reconfigurable connectors for safely adapting systems online. Fundamental features of the Kompics component model, such as nested hierarchical composition, support for strongly-typed interfaces, and explicit dependency management using ports, are found in other component models, such as OpenCom [63], Fractal [44], Oz/K [139], K-Components [72], OMNeT++ [226], and OSGi [167]. However, the style of component interaction, based on synchronous interface invocation, precludes compositional concurrency in these models making them unsuited to present-day multi-core hardware architectures.

Oz/K [139] is a kernel language for component-based open programming based on the Kell calculus [197]. Oz/K also targets component reconfiguration but in Oz/K, components communicate by atomic rendezvous on gates, which are similar to the synchronous channels of π-calculus. In contrast, Kompics components communicate asynchronously through explicit channels. Similar to Kompics, in Oz/K, gates form the only means of communication between components, ensuring isolation.

The Fractal [44] component model allows the specification of components that are reflective, hierarchical, and dynamically reconfigurable. However, Fractal is agnostic with respect to the execution model of components. Kompics is a reactive component model having these desirable properties of Fractal, but it enforces a particular execution and component interaction model, facilitates programming distributed protocols.

With respect to industrial standards such as Enterprise JavaBeans (EJB) or the CORBA Component Model (CCM), Kompics constitutes a more flexible component model, which does not embed predetermined non-functional services, like managed persistence, security, or transactional integrity. Kompics is targeted at building generic distributed systems, not just tiered client-server enterprise applications. In contrast to these models, Kompics employs message-passing component interaction, therefore enabling a simple and compositional concurrency model. These models do not support nested hierarchical composition, making them inadequate for supporting rich architectural patterns like those we showed in Chapter 3.

5.3 Software Architecture Description Languages

Component-based systems that support dynamic run-time reconfiguration functionality use either reflective techniques [150] or dynamic software architecture models, such as Fractal [44], Rapide [144], and ArchStudio4/C2 [65]. Kompics's reconfiguration model is similar to the dynamic software architecture approaches, but a major difference is that the software architecture in Kompics is not specified explicitly in an architecture description language (ADL), rather it is implicitly constructed at run time.

ArchJava [8] proposes an explicit software architecture and guarantees *communication integrity* – i.e., that components only communicate along declared connections between ports – an idea that we leverage in Kompics for safe dynamic reconfiguration.

5.4 Protocol Composition Frameworks

Protocol composition frameworks like Horus [224, 225], Ensemble [101], Appia [156], or Bast [84], were specifically designed for building distributed systems by layering modular protocols. Protocol stacks are composed from building blocks called protocol modules which interact through events. These systems, however, focus on the flow of events though the protocol stack, rather than on the encapsulation and abstraction of lower-level protocols. Enabling protocol composition solely by layering prevents the construction of complex nested hierarchical architectures. With Kompics we employ nested hierarchical composition which enables richer, more useful architectural patterns, as we illustrated in Chapter 3.

Live distributed objects [168] are the most similar to Kompics in their goal of supporting encapsulation and composition of distributed protocols. Live objects endpoints are similar to Kompics ports, providing bidirectional message passing, however, endpoints in Live objects support only one-to-one connections. Live objects do not support nested hierarchical composition nor the dynamic reconfiguration of the protocol architecture.

Our work is also relevant within the context of popular non-blocking network communication frameworks – used to build high-performance event-driven server applications – such as SEDA [230], Lift [53], Twitter's Finagle [228] for Scala, and Facebook's Tornado [71] for Python. Kompics'

asynchronous event-driven programming framework allows it to seamlessly integrate different Java NIO networking frameworks – such as Netty [163], Apache MINA [185], and Grizzly[218] – as pluggable components.

Rather than supporting hierarchical architectures or dynamic reconfiguration, in SEDA the focus is on performance, namely on self-tuning resource management and dynamic adaptation to changes in load, in order to provide graceful degradation in performance. Kompics does not inhibit support for such properties, which could be enabled by a custom component scheduler which would allocate different worker pools for different groups of components, corresponding to different stages in SEDA.

5.5 Process Calculi and Other Concurrency Models

There exist several languages and concurrency formalisms that support the study of concurrent algorithms. In particular, the Kompics concurrency model can be contrasted to the synchronous π-calculus [155], CSP [108], CCS [154], and the asynchronous I/O automata [148, 149, 145], which also model hierarchical concurrent components.

Similar to the Kompics notion of port polarity, π-calculus [155] uses names and co-names for actions. I/O automata [148, 149, 145] offer a natural model for describing distributed algorithms, and supports the construction of modular, hierarchical correctness proofs.

The Spectrum Simulation System [92] is a research tool for the design and study of distributed algorithms. Faithful to the I/O automaton model [149, 148], Spectrum provides the ability to integrate the entire process of specification, design, debugging, analysis, and correctness proofs for distributed algorithms. In Kompics we focus on distributed systems.

Statecharts [99] provide a modular way to describe complex systems. Essentially, each orthogonal component of a statechart is a finite state machine that, in response to an event, may make a state transition and generate a new event. The Statemate system [100], based on the Statechart model, provides a graphical editor for building statecharts, a statechart simulator, and automatic translation into Ada and C. Statemate exploits the hierarchical structure of statecharts by permitting users to design and study complex systems at varying levels of detail.

In the Occam programming language [183], which implements the CSP [108] concurrency model, the sequential process control flow is convenient for describing algorithms that are inherently sequential. However, it can be cumbersome for describing distributed algorithms in which a given process may interact with other processes at different stages of the protocol.

DEVS [232] is an object-oriented system in which system components, called models, have input and output ports that may be coupled in a hierarchical fashion, similar to Kompics. DEVS serves as a medium for developing hierarchical distributed simulation models and architectures.

5.6 Scalable Simulation and Replay Debugging

There exist several popular simulators for peer-to-peer systems including P2PSim [136], Peersim [159], ProtoPeer [82], RealPeer [107], WiDS [140], and Oversim [32], which extends the OMNeT++ [226] domain-independent discrete-event simulator. More relevant to Kompics, however, are the frameworks for building distributed systems that support using the same code in both simulation and production deployment, such as Distributed System Foundation (DSF) [213], Neko [222], Mace [121], and WiDS [140].

In Mace, and WiDS, programmers specify system logic using a high-level event-based language that is subsequently compiled to C++ code, which, in turn, uses APIs for framework-specific libraries. When switching from simulation to production deployment, WiDS and Mace require programmers to rebuild the system and link it to network libraries.

Splay [134] also allows system specification in a high-level language, namely Lua. Splay supports the deployment and evaluation of P2P systems in a testbed environment using declarative experiment definitions, but it does not support repeatable simulation on a single host. Oversim [32] code, on the other hand, cannot be executed in production environments.

ModelNet [223] is a scalable network emulator that allows the deployment of thousands of application nodes on a set of cluster machines. ModelNet provides a realistic network environment to the deployed application. This has the advantage of offering a real-world large-scale testbed in a controlled environment, however, ModelNet does not offer support for defining and orchestrating stress test experiments.

Liblog [86] is a tool that enables replay debugging for distributed C/C++ applications. When running the distributed application, the Liblog library is preloaded and all activity – including exchanged messages, thread scheduling, and signal handling – is logged. Each process logs its own activity locally. Post-mortem, all logs are fetched to a central machine where an interesting subset of processes are replayed in step-by-step debugging mode. Liblog integrates GDB into the replay mechanism for simultaneous source-level debugging of multiple processes. Liblog uses Lamport timestamps [127] in exchanged messages to ensure that replay is consistent with the causal order of events.

Part II

Scalable and Consistent Distributed Storage

Chapter 6

Background, Motivation, and Problem Statement

Modern web-scale applications generate and access massive amounts of semi-structured data at very high rates. To cope with such demands, the underlying storage infrastructure supporting these applications and services, must be extremely *scalable*. The need for scalability, high availability, and high performance motivated service operators to design custom storage systems [67, 125, 30, 51, 102] that *replicate* data and *distribute* it over a large number of machines in a datacenter distributed system.

Due to the semi-structured nature of the data, such systems often have a simple API for accessing data in terms of a few basic operations:

- ***put****(key, value)*
- *value :=* ***get****(key)*
- ***delete****(key)*

and hence they are referred to as *key-value stores*. The number of replicas accessed by *put* and *get* operations determines the level of data *consistency*

provided by the system [26]. To achieve strong data consistency, whereby clients have the illusion of a single storage server, *put* and *get* operations need to access overlapping quorums of replicas [88]. Typically, the more servers an operation needs to wait for, the higher its latency will be [1]. Early designs of key-value stores [67, 125] were targeted to applications that did not require strong data consistency, and driven by the need for low latency and availability, they chose to provide only *eventual consistency* for *put* and *get* operations.

Eventual consistency [215] means that for a given key, data values may diverge at different replicas, e.g., as a result of operations accessing less than a quorum of replicas or due to network partitions [66, 43, 89]. Eventually, when the application detects conflicting replicas, it needs to reconcile the conflict. This can be done automatically for data types with monotonic [11] or commutative [204] operations. In general however, conflict detection and resolution increases application complexity, both syntactically, by cluttering its logic with extra code paths, and semantically, by requiring programmers to devise reconciliation logic for all potential conflicts.

There is a significant class of applications that cannot rely on an eventually consistent data store. In particular, financial and electronic health record applications, services managing critical meta-data for large cloud infrastructures [45, 110], or more generally, systems in which the results of data-access operations have external side-effects, all need a data store with strong consistency guarantees in order to operate *correctly* and *securely*. The strongest level of consistency for *put* and *get* operations, is called *atomic consistency* or *linearizability* [106] and informally, it guarantees that for every key, a *get* returns the value of the last completed *put* or the value of a concurrent *put*, and once a *get* returns a value, no subsequent *get* can return an older, stale value. Thus, in spite of failures and concurrency, *put* and *get* operations appear to occur in the same sequential order at all clients and every *get* always returns the value of the most recent *put*.

When scalable systems grow to a really large number of servers, their management effort increases significantly. Therefore, *self-organization* and self-healing are commendable properties of modern scalable data stores [209]. Many existing key-value stores [67, 125, 30, 80] rely on *consistent hashing* [120] for automatically managing data storage and replication responsibilities when servers join and leave the system, or when they fail.

Moreover, with consistent hashing all servers are symmetric. No master server means there is no scalability bottleneck and no single point of failure.

Scaling to a very large number of servers also increases the likelihood of *network partitions* [66] and *inaccurate failure suspicions* [49] caused by network congestion or by the failure or misconfiguration of network equipment. For the class of critical applications mentioned above, it is imperative to maintain consistency during adverse network conditions, even at the expense of service availability [43, 89].

The complexities of eventual consistency and the need for atomic consistency motivated us to explore how linearizability can be achieved in scalable key-value stores based on consistent hashing [67, 125, 30, 80]. The problem is that simply applying quorum-based *put* and *get* operations [23] within replication groups dictated by consistent hashing [208], fails to satisfy linearizability in the face of dynamic group membership, network partitions, message loss, partial synchrony, and false failure suspicions. We show the pitfalls of a naïve approach and describe the challenge of achieving linearizability in Section 6.5.

With CATS we make the following contributions:

- We introduce *consistent quorums* as an approach to guarantee linearizability in a decentralized, self-organizing, dynamic system spontaneously reconfigured by consistent hashing, and prone to inaccurate failure suspicions and network partitions.

- We showcase consistent quorums in the design and implementation of CATS, a scalable distributed key-value store where every data item is an atomic register with linearizable *put* and *get* operations and a dynamically reconfigurable replication group.

- We evaluate the cost of consistent quorums and the cost of achieving atomic data consistency in CATS. We give evidence that consistent quorums admit both, on the one hand, system *designs* which are scalable, elastic, self-organizing, fault-tolerant, consistent, and partition-tolerant, as well as, on the other hand, system *implementations* with practical performance and modest overhead – 5% decrease in throughput for read-intensive and 25% for write-intensive workloads.

In designing CATS we leveraged the research work on consistent hashing, which has been used for building scalable, self-organizing, yet weakly-consistent distributed key-value stores [67, 125, 30, 80], as well as the work on quorum-based replication, both in static and dynamic systems, which provide strong data consistency but are not scalable [23, 56, 4, 143].

6.1 Consistent Hashing and Distributed Hash Tables

Consistent hashing [120] is a technique for partitioning data among the nodes in a distributed storage system, such that adding and removing nodes requires minimal repartitioning of data. Consistent hashing employs an identifier space perceived as a ring. Both data items and nodes are mapped to identifiers in this space. Many distributed hash tables (DHTs), such as Chord [208] and Pastry [192], were built using consistent hashing.

Our architecture leverages Chord, yet the idea of consistent quorums can be applied to other DHTs as well, to build consistent, partition tolerant, and scalable key-value storage systems. Chord provides a scalable, self-organizing, and fault-tolerant system for maintaining a consistent hashing ring topology, which determines the partitioning of data among nodes. Additionally, Chord provides mechanisms for efficiently finding the node responsible for storing a particular key-value pair.

Each node in the system maintains a *succ* pointer to its successor on the consistent hashing ring. The successor of a node n is the first node met going in the clockwise direction on the identifier ring, starting at n. Similarly, each node keeps a *pred* pointer to its predecessor. The predecessor of n is the first node met going anti-clockwise on the ring, starting at n. A node n is *responsible* for storing all key-value pairs for which the key identifier, or a hash thereof, belongs to the range $(p.pred, p]$. For fault tolerance of the ring topology, each node n maintains a *successor-list*, consisting of n's c immediate successors. For fault tolerance on the data level, all key-value pairs stored on n are replicated on the first $r-1$ nodes in n's successor-list, where r is the replication degree. A periodic stabilization algorithm was proposed in Chord [208] to maintain the ring pointers under node dynamism, i.e., nodes joining and leaving the system, or failing.

Because the Chord periodic stabilization protocol was not designed to cope with network partitions and mergers, it is possible that a network partition divides the ring topology into two independent rings. In the design of CATS we chose an extended topology maintenance protocol [202], which repairs the consistent hashing ring pointers after a transient network partition, effectively merging multiple partitioned ring overlays into a single consistent hashing ring. This is essential for achieving complete tolerance to network partitions [66].

6.2 Consistency, Availability, and Partition Tolerance

Brewer's conjecture [43], formalized and proven by Gilbert and Lynch [89], and generally referred to as the CAP theorem, states that a distributed service operating in an asynchronous network, cannot simultaneously satisfy guarantees of consistency, availability, and network partition tolerance.

Consistency guarantees are rules governing the operations of the service. For example, a particular consistency model might require that every read operation returns the value written by the most recent update operation. We review two major consistency models in the following section.

Availability means that every operation request received by a non-failing node must eventually generate a response. This is a liveness property [10] requiring the termination of the protocol that implements the service operation. Notwithstanding the fact that it places no bound on how long the algorithm may run before terminating, availability requires that even when severe network failures occur, every request must terminate.

A network partition is a situation where the nodes of a distributed system are split into disconnected components which cannot communicate with each other [66]. This is a degenerate case of message loss whereby the communication links between the partitioned components systematically drop messages for a while, until the network partition is repaired.

System designers have to chose two out of three of these properties. Since in practical large-scale networks we have no guarantee of the absence of network partitions, the choice boils down to consistency vs. availability. Such a choice depends on the target application. For some applications, availability is of utmost importance, while weaker consistency guarantees

such as eventual consistency [67] suffice. In contrast, our focus is on applications that require strong consistency guarantees. The system can become unavailable under certain failure scenarios, yet it provides consistency and it tolerates network partitions and process crash failures. It is important to stress that as long as a majority of nodes are accessible, service unavailability only occurs in the presence of network partitions, which are relatively rare in practice. This means the system is actually consistent and available while the network is connected. Consistency is always maintained, however, once a network partition occurs, the service may become unavailable in minority partition components. Yet, the service may still be available in partition components with a majority of alive nodes.

6.3 Linearizability and Sequential Consistency

For a replicated storage service, linearizability provides the illusion of a single storage server. Despite the possibility of multiple clients issuing concurrent operations which could reach different replicas in different orders, each operation appears to take effect instantaneously at some point between its invocation and its response [106]. Failed update operations, whereby the client crashes before the operation completes, either successfully change the state of the system, or have no apparent effect, regardless of the number of replicas that the client managed to update before crashing. As such, linearizability is sometimes called atomic consistency [129].

In the context of a key-value store, linearizability guarantees that for every key, a *get* always returns the value updated by the most recent *put*, never a stale value, thus giving the appearance of a globally consistent shared memory. Linearizability is the strongest level of consistency for *put* and *get* operations on a single key-value data item.

Linearizability is said to be a *local* property [106]. A property is called local, if the whole system satisfies the propety whenever each individual object satisfies the property. In other words, linearizability is compositional. In a key-value store, this means that if operations on an individual key-value pair are linearizable, then the interleaving of all operations on the whole set of key-value pairs in the store is itself linearizable. Linearizability is also a *non-blocking* property, since the invocation of an operation is never

required to wait for another pending invocation to complete. This enhances concurrency and allows for low operation response times.

A related but strictly weaker consistency model, introduced by Lamport [128] in the context of concurrent programming for shared-memory multiprocessors, is *sequential consistency*. Sequential consistency requires that all concurrent operations on a shared data object appear to have executed atomically, in some sequential order that is consistent with the order seen at each individual process [25]. For linearizability, this order must also preserve the global ordering of nonoverlapping operations. Two operations are said to be nonoverlapping if one completes, in real time, before the other one is invoked. This extra real time requirement is what makes linearizability strictly stronger than sequential consistency.

Informally, the real time ordering guarantee offered by linearizability, means that once an update operation completes, any subsequent read operation will immediately observe its effects. In contrast, with sequential consistency, updates are not guaranteed to become immediately visible. For example, a writer may complete an update operation and then send a message to a waiting reader, instructing it to read the shared register. After attempting to read the register, the reader may still not see the update, a behaviour which could lead to confusion in some applications. In the absense of such out-of-band communication, however, sequential consistency is not distinguishable from linearizability from the point of view of the processes in the system.

In contrast to linearizability, sequential consistency is not compositional.

6.4 Quorum-Based Replication Systems

In order to provide fault-tolerant storage, data is replicated across multiple computers. The notion of using quorum-based voting for operations on replicated data was introduced by Gifford [88].

For a static set of nodes replicating a data item, Attiya, Bar-Noy, and Dolev showed how a shared memory register abstraction can be implemented in a fully asynchronous message-passing system while satisfying linearizability [23]. Their protocol, known as the ABD algorithm, implemented a single-writer multiple-reader (SWMR) atomic register. The ABD

algorithm was extended by Lynch and Shvartsman [146] to implement a multiple-writer multiple-reader (MWMR) atomic register, whereby each read and write operation proceeds in two phases. In the remainder of this thesis, we'll continue to use the ABD alias to refer to such a two-phase protocol, where each read and write operation is applied on a majority quorum of nodes, such that the quorums of any two operations always intersect in at least one node [88].

ABD was extended by protocols like RAMBO [147], RAMBO II [90], RDS [56], and DynaStore [4] to dynamic networks, where replica nodes can be added and removed, while still preserving linearizability. Similarly, SMART [143] enabled reconfiguration in replicated state machines [198].

While these systems can handle dynamism and provide strong data consistency, they are not scalable as they cannot partition the data across a large number of machines. We leverage the reconfiguration techniques contributed by these works and we attempt to apply them at large scale, to build a system that is completely decentralized and self-managing.

6.5 Problem Statement

A general replication scheme used with consistent hashing is *successor-list* replication [208], whereby every key-value data item is replicated at a number of servers that succeed the responsible node on the consistent hashing ring. An example is shown in Figure 6.1. Here, the replication degree is three and a quorum is a majority, i.e., any set of two nodes from the replication group. A naïve attempt of achieving linearizable consistency is to use a shared memory register approach, e.g. ABD [23], within every replication group. This will not work as false failure suspicions, along with consistent hashing, may lead to non-overlapping quorums. The diagram in Figure 6.2 depicts such a case, where node 15 falsely suspects node 10. According to node 10, the replication group for keys in the range (5, 10] is {10, 15, 20}, while from the perspective of node 15, the replication group for the same keys is {15, 20, 25}. Now, two different operations, for instance on key 8, may access non-intersecting quorums leading to a violation of linearizability. For example, a *put* operation may complete after updating the value associated with key 8 at replicas 10 and 20. A subsequent *get*

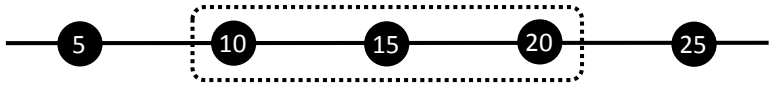

Figure 6.1. A correct replication group for keys in range $(5, 10]$ using consistent hashing with successor-list replication. Replication degree is three and a majority quorum is any set of two in the replication group, i.e., {10, 15}, {10, 20}, or {15, 20}.

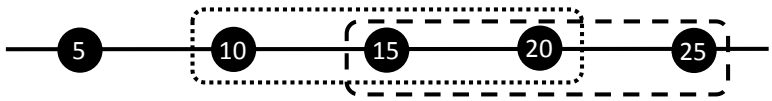

Figure 6.2. Node 15 inaccurately suspects node 10 to have crashed. As a consequence, node 15 assumes the replication group for key range $(5, 10]$ is {15, 20, 25}. Other nodes may still assume the replicas set is {10, 15, 20}, potentially leading to non-overlapping quorums, e.g., {10, 20} and {15, 25}, or {10, 15} and {20, 25}.

operation may reach replicas 15 and 25 and return a stale value despite contacting a majority quorum of replicas.

In a key-value store with replication groups spontaneously reconfigured by consistent hashing, applying *put* and *get* operations on majority quorums is not sufficient for achieving linearizability. Furthermore, in such a self-managing system, where changes in one replication group are related to changes in other groups, and where *put* and *get* operations may occur during reconfiguration, guaranteeing atomic consistency is non-trivial. In fact, any quorum-based algorithm will suffer from the problem of non-intersecting quorums when used in a dynamic replication group dictated by consistent hashing. We propose consistent quorums as a solution. In contrast to reusing an existing dynamic replication protocol within each replication group, as a black box, consistent quorums allow us to decouple group reconfigurations from data operations in a clean way, avoiding the complexities and unnecessary overheads of those protocols.

Chapter 7

Consistent Quorums

In a typical quorum-based protocol an operation coordinator sends request messages to a set of participants and waits for responses. Upon receiving a request message, each participant acts on the request and responds to the coordinator with an acknowledgement. The coordinator completes the operation as soon as it receives a quorum [88] of acknowledgements. Typically, essential *safety* properties of the protocol are satisfied by ensuring that the quorums for different operations, e.g., *put* and *get*, intersect in at least one participant.

Quorum intersection is easily achieved in a static system with a fixed set of nodes. In a dynamic system however, different nodes may have inconsistent views of the group membership. It is possible thus, that the number of nodes which consider themselves responsible for a key range, i.e., the number of nodes in a replication group, is larger than the replication degree. As a result, successive *put* and *get* operations may complete by contacting non-overlapping quorums, as we've shown in the previous chapter, which could lead to a violation of linearizability.

The idea is then to maintain a membership *view* of the replication group at each node which considers itself to be a replica for a particular key range according to the principle of consistent hashing. Each node in a replication

group has a view $v_i = \langle s, G_i \rangle$, where s represents the set of keys or the key range replicated by the nodes in the group, G_i is the set of nodes in the replication group, and i is the version number of the view. A node has an *installed* view for each key range that it replicates. We say that a node n is in view v_i, not when $n \in G_i$, but when n has view v_i installed.

Definition 1. *For a given replication group G, we say that a quorum Q is a **consistent quorum** of G, if every node n in Q has the same view of G installed, at the time when Q is assembled, i.e., when n sends its acknowledgement in Q.*

When a node replies to a request for a key k, it stamps its reply with its currently installed view for the corresponding key range, s, where $k \in s$. The main idea is that a quorum-based operation will succeed only if it collects a quorum of nodes with the same view, i.e., a *consistent quorum*.

As node membership changes over time, a mechanism is needed to reconfigure the membership views consistently at all replication group members. For that we devised a group reconfiguration protocol based on Paxos consensus [130], extended with an extra view installation phase and augmented with consistent quorums. We present our group reconfiguration protocol in the following section.

7.1 Paxos-based Group Reconfiguration using Consistent Quorums

Replication groups must be *dynamically reconfigured* [3, 132] to account for new node arrivals and to restore the replication degree after group member failures. The system starts in a consistent configuration, whereby each node has consistent views installed for every key range that it replicates. Thereafter, within each replication group, the system reconfigures by using the members of the current view as acceptors in a consensus instance which decides the next view. Each consensus instance is identified by the view in which it operates. Therefore, views are decided in sequence and installed at all members in the order in which they were decided. This decision sequence determines view version numbers. Algorithms 1–3 illustrate our Paxos-based group reconfiguration protocol using consistent quorums. Earlier we defined a consistent quorum as an extension of a regular quorum.

Without loss of generality, in the remainder of this chapter we focus on majority-based (consistent) quorums.

A reconfiguration is proposed and overseen by a coordinator node which could be a new joining node, or an existing node that suspects one of the group members to have failed. Reconfiguration $(v_i \Rightarrow v_{i+1})$ takes the group from the current view v_i to the next view v_{i+1}. Group size stays constant and each reconfiguration changes the membership of a replication group by a single node. One new node joins the group to replace a node which leaves the group. The version number of a view is incremented by every reconfiguration. The reconfiguration protocol amounts to the coordinator getting the group members of the current view, v_i, to agree on the next view, v_{i+1}, and then *installing* the decided next view at every node in the current and the next views, i.e., $G_i \cup G_{i+1}$. We say that a node is in view v_i, once it has installed view v_i and before it installs the next view, v_{i+1}. Nodes install views sequentially, in the order of the view versions, which reflects the order in which the views were decided.

The key issue catered for by the reconfiguration protocol is to always maintain the quorum-intersection property for consistent quorums, even during reconfigurations. To make sure that for any replication group, G, no two consistent quorums may exist simultaneously, e.g., for the current and the next views of a reconfiguration, the decided next view, v_{i+1}, is first installed on a majority of the group members of the current view, G_i, and thereafter it is installed on the new group member, $G_{i+1} \setminus G_i$.

Reconfiguration Proposals

Proposed new views are devised based on changes in the consistent hashing ring topology. Under high churn [191], different nodes may concurrently propose conflicting next views, e.g., when a node joins the system shortly after another node fails, and both events lead to the reconfiguration of the same replication group. Using consensus ensures that the next view is agreed upon by the members of the current view, and the group reconfiguration proceeds safely. When a reconfiguration proposer p notices that the decided next view $v_{i+1} = \langle s, G_d \rangle$ is different from the one it had proposed, say $v_{i+1} = \langle s, G_p \rangle$, p assesses whether a reconfiguration is still needed. This may be the case, for example, when G_d still contains a node

Algorithm 1 Reconfiguration coordinator

Init: phase1Acks[v_i] $\leftarrow \emptyset$, phase2Acks[v_i] $\leftarrow \emptyset$, phase3Acks[v_i] $\leftarrow \emptyset$
prop[v_i] $\leftarrow 0$, pRec[v_i] $\leftarrow \perp$ ▷ $\forall$ *consensus instance* v_i

1: **on** $\langle$P1B: v_i, ACK, $pn, rec, v\rangle$ **do**
2: phase1Acks[v_i] $\leftarrow$ phase1Acks[v_i] $\cup \{(pn, rec, v)\}$
3: $v_Q \leftarrow$ ***consistentQuorum***(*extractViewMultiset*(phase1Acks[v_i]))
4: **if** $v_Q \neq \perp$ **then**
5: r $\leftarrow$ *highestProposedReconfiguration*(phase1Acks[v_i], v_Q)
6: **if** r $\neq \perp$ **then**
7: pRec[v_i] $\leftarrow$ r
8: **send** $\langle$P2A: v_i, prop[v_i], pRec[v_i]$\rangle$ **to** *all members of* G_Q

9: **on** $\langle$P1B: v_i, NACK$\rangle \vee \langle$P2B: v_i, NACK$\rangle$ **do**
10: prop[v_i]++ ▷ *retry with higher proposal number, unique by process id*
11: **send** $\langle$P1A: v_i, prop[v_i]$\rangle$ **to** *all members of* G_i

12: **on** $\langle$P2B: v_i, ACK, $v\rangle$ **do**
13: phase2Acks[v_i] $\leftarrow$ phase2Acks[v_i] $\cup \{v\}$
14: $v_Q \leftarrow$ ***consistentQuorum***(phase2Acks[v_i])
15: **if** $v_Q \neq \perp$ **then**
16: **send** $\langle$P3A: v_i, rec[v_i]$\rangle$ **to** *all members of* G_Q

17: **on** $\langle$P3B: $v_i, v\rangle$ **do**
18: phase3Acks[v_i] $\leftarrow$ phase3Acks[v_i] $\cup \{v\}$
19: **if** ***consistentQuorum***(phase3Acks[v_i]) $\neq \perp$ **then**
20: **send** $\langle$P3A: v_i, pRec[v_i]$\rangle$ **to** *new group member* ($G_{i+1} \setminus G_i$)

which p suspects to have failed. In such a scenario, p generates a new reconfiguration to reflect the new view, and then proposes it in the new protocol instance determined by v_{i+1}.

In the algorithm specifications we omit the details pertaining to ignoring orphan messages or breaking ties between proposal numbers based on the proposer id. The *consistentQuorum* function tests whether a consistent

Algorithm 2 Current group member

Init: $\text{wts}[v_i] \leftarrow 0$, $\text{rts}[v_i] \leftarrow 0$, $\text{aRec}[v_i] \leftarrow \perp$ ▷ $\forall$ *consensus instance* v_i

1: **on** $\langle$P1A: $v_i, p\rangle$ **do** ▷ *acceptor role*
2: **if** $p \geq \text{rts}[v_i] \wedge p \geq \text{wts}[v_i]$ **then**
3: $\text{rts}[v_i] \leftarrow p$ ▷ *promise to reject proposal numbers lower than p*
4: **send** $\langle$P1B: v_i, ACK, $\text{wts}[v_i]$, $\text{aRec}[v_i]$, *view*$(v_i.s)\rangle$ **to** *coordinator*
5: **else send** $\langle$P1B: v_i, NACK$\rangle$ **to** *coordinator*

6: **on** $\langle$P2A: $v_i, p, (v_i \Rightarrow v_{i+1})\rangle$ **do** ▷ *acceptor role*
7: **if** $p > \text{rts}[v_i] \wedge p > \text{wts}[v_i]$ **then**
8: $\text{wts}[v_i] \leftarrow p$ ▷ *promise to reject proposal numbers lower than p*
9: $\text{aRec}[v_i] \leftarrow (v_i \Rightarrow v_{i+1})$ ▷ *accepted reconfiguration*
10: **send** $\langle$P2B: v_i, ACK, *view*$(v_i.s)\rangle$ **to** *coordinator*
11: **else send** $\langle$P2B: v_i, NACK$\rangle$ **to** *coordinator*

12: **on** $\langle$P3A: $v_i, (v_i \Rightarrow v_{i+1})\rangle$ **do** ▷ *learner role*
13: *installView*(v_i, v_{i+1})
14: **send** $\langle$P3B: v_i, *view*$(v_i.s)\rangle$ **to** *coordinator*
15: **send** $\langle$Data: $(v_i \Rightarrow v_{i+1})$, *data*$(v_i.s)\rangle$ **to** *new member* $(G_{i+1} \setminus G_i)$

Algorithm 3 New group member

1: **on** $\langle$P3A: $v_i, (v_i \Rightarrow v_{i+1})\rangle$ **do**
2: *installView*(v_i, v_{i+1}) ▷ *makes* v_{i+1} ***busy*** *if the data was not received yet*

3: **on** $\langle$Data: $(v_i \Rightarrow v_{i+1})$, *data*$\rangle$ **do** ▷ *from old members of group* G_i
4: $\text{dataSet}[v_{i+1}] \leftarrow \text{dataSet}\ [v_{i+1}] \cup \{(data, v_i)\}$
5: **send** $\langle$DataAck: $v_{i+1}\rangle$ **to** *old member of group* G_i
6: **if** ***consistentQuorum***(*extractViewMultiset*($\text{dataSet}[v_{i+1}]$)) $\neq \perp$ **then**
7: *storeHighestItems*($\text{dataSet}[v_{i+1}]$) ▷ *makes* v_{i+1} ***ready***

quorum exists among a set of views and if so, it returns the view of that consistent quorum. Otherwise it returns $\perp$. The *extractViewMultiset* function maps a multiset of (*proposal number*, *reconfiguration*, *view*) triples to the corresponding multiset of *views*. The *highestProposedReconfiguration* takes a multiset of such triples and returns the *reconfiguration* with the highest *proposal number*, among the triples whose *view* matches view v_Q, its second parameter. The *view* function returns the currently installed

view corresponding to a given key range or just a single key, and the *data* function retrieves the timestamped data items corresponding to a given key range. The *storeHighestItems* function takes a multiset of sets of timestamped key-value data items, and for each distinct key, it stores locally the corresponding data item with the highest timestamp. Finally, the *installView* function takes two consecutive views v_i and v_{i+1}. If the local node belongs to group G_i, it must have v_i installed before proceeding with the view installation. If the local node is the new node in group G_{i+1}, it can proceed immediately. We discuss these situations in more detail below, when we describe the *install queue* and the *data chain* mechanisms.

Phase 1 and phase 2 of the protocol are just the two phases of the Paxos consensus algorithm [130], augmented with consistent quorums. We could view Paxos as a black-box consensus abstraction whereby the group members of the currently installed view, G_i, are the acceptors deciding the next view to be installed, v_{i+1}. Nonetheless, we show the details of phases 1 and phase 2 as an illustration of using consistent quorums.

View Installation and Data Transfer

Phase 3 of the protocol is the view installation phase. Once the next view v_{i+1} is decided, the coordinator asks the members of the current view v_i to install view v_{i+1}. Once v_{i+1} is installed at a majority of nodes in G_i, only a minority of nodes are still in view v_i, and so it is safe to install v_{i+1} at the new member, without allowing two simultaneous majorities, i.e., one for v_i and one for v_{i+1}. When a member of group G_i installs v_{i+1}, it also sends the corresponding data to the new member of v_{i+1}. Conceptually, once the new member receives the data from a majority of nodes in the old view, it stores the data items with the highest timestamp from a majority. In practice however, we optimize the data transfer such that only keys and timestamps are pushed from all nodes in G_i to the new node, which then pulls the latest data items in parallel from different replicas.

Ensuring that the new group member gets the latest data items among a majority of nodes in the old view is necessary for satisfying linearizability of the *put* and *get* operations that occur during reconfiguration. To see why, consider a case where a *put* operation occurs concurrently with reconfiguration $(v_i \Rightarrow v_{i+1})$. Assume that this *put* operation updates the

value of key k with timestamp t, to a newer value with timestamp $t+1$, and further assume that a majority of replicas in G_i have been updated while a minority of replicas are yet to receive the update. If the new group member, n, didn't get the latest value of k from a majority of replicas in G_i, and instead n transferred the data from a single replica, it would be possible that n got the old value from a replica in the not-yet-updated minority. In this situation, a majority of nodes in G_{i+1} have the old value of k. As we discuss in Section 7.2, the concurrent *put* operation will complete using the old view v_i. A subsequent *get* accessing a consistent quorum with view v_{i+1} may later return the old value of k, thus violating linearizability.

Install Queue

Two interesting situations may arise in asynchronous networks. Recall that once a reconfiguration $(v_i \Rightarrow v_{i+1})$ has been decided, P3A messages instruct the nodes in G_{i+1} to install the new view v_{i+1}. First, it is possible that multiple consecutive reconfigurations progress with a majority of nodes while the nodes in a minority temporarily do not receive any P3A messages. Later, the minority nodes may receive P3A messages in a different order from the order in which their respective reconfigurations were decided. Assume for example that node n is such a node in this "left behind" minority. When node n is instructed to apply a reconfiguration $(v_i \Rightarrow v_{i+1})$ whereby n is a member of group G_i, but n has not yet installed view v_i, node n stores the reconfiguration in an *install queue* marking it for installation in the future, immediately after installing view v_i. Accordingly, node n will issue view installation acknowledgments, i.e., P3B messages, and it will initiate the corresponding data transfers to the new node in G_{i+1}, only after node n installs view v_{i+1}. This install queue mechanism ensures that even if nodes receive view installation requests out of order, views are still going to be installed in the order in which they were decided.

Data Chain

Another interesting situation that may be caused by asynchronous execution is one where after having installed a view v_i, a new group member n, subsequently and in rapid succession installs newer views v_{i+1}, v_{i+2}, etc.,

before having received all the data for view v_i. In such cases, node n stores the newer views in a *data chain*, reminding itself that upon completely receiving the data for view v_i, it should transfer it to the new group member in group G_{i+1}, G_{i+2}, etc. Even if data is transferred slowly and it arrives much later than the view installation, node n still honors its responsibility to push the data forward to the new nodes in subsequently newer views. This data chain mechanism ensures that upon a view change $(v_i \Rightarrow v_{i+1})$, all the nodes in G_i push the data to the new node in G_{i+1}. Considering various failure scenarios, this increases the new node's chances of collecting the data from a majority of nodes in G_i, and therefore it preserves the linearizability and liveness of *put* and *get* operations.

Termination

We say that a reconfiguration $(v_i \Rightarrow v_{i+1})$ has *terminated*, when a majority of nodes in group G_{i+1} have installed the new view v_{i+1}. Once v_{i+1} has been installed at a majority of nodes in G_{i+1}, a consistent quorum for the view v_{i+1} becomes possible, enabling the group to be reconfigured yet again, to evolve and adapt to new node joins and failures.

Considering a reconfiguration $(v_i \Rightarrow v_{i+1})$ to be complete once view v_{i+1} has been installed by a majority of nodes in G_{i+1}, is sufficient for progress, however, it introduces a window of vulnerability. Assuming a group size of r, when view v_i is installed at all nodes in G_i, the system tolerates the failure of $\lfloor \frac{r}{2} \rfloor$ nodes in G_i. In contrast, if view v_{i+1} is only installed at $\lceil \frac{r}{2} \rceil$ nodes in G_{i+1} and $\lfloor \frac{r}{2} \rfloor$ nodes of G_{i+1} don't have it installed, the system tolerates no failures in G_{i+1}, since any such failure would lead to the unavailability of a consistent quorum for v_{i+1}. This window of vulnerability is reduced as soon as more nodes in group G_{i+1} install view v_{i+1}, increasing the group's tolerance to individual node failure.

End-game Mode

A stronger variant of reconfiguration termination would be to consider a reconfiguration complete, only when all the nodes in the new view have the new view installed and ready, meaning that the new node has also received all the data for the view (see Algorithm 3, line 7).

To minimize the risk of unavailability in the case where view v_{i+1} is installed only at $\lceil \frac{r}{2} \rceil$ nodes of group G_{i+1}, and yet one of these nodes crashes shortly thereafter, the other live members of group G_i which are aware of the reconfiguration decision, e.g., by having received a P3A message, keep on trying to get view v_{i+1} installed on the remaining nodes of G_{i+1}. This behavior is essentially duplicating the phase-3 behaviour of the reconfiguration coordinator, and has the benefit of overseeing the termination of a reconfiguration even when the coordinator crashes. In other words, once any node n in $G_i \cup G_{i+1}$ receives a view installation request, i.e., a P3A message, it enters an *end-game mode* whereby it tries to get the reconfiguration applied to all other nodes in $G_i \cup G_{i+1}$. This is akin to flooding the reconfiguration to all nodes involved, so that the reconfiguration has a chance of terminating even in the worst failure scenarios. When in end-game mode, a node periodically keeps retrying to get the reconfiguration applied at all other nodes, until it gets an acknowledgement from each, in the form of a P3B message. We omit the illustration of the end-game mode from Algorithms 1–3 for clarity. The end-game mode quiesces only once the reconfiguration has been applied at all nodes in $G_i \cup G_{i+1}$.

Taking into account the possibility of message loss, network partitioning, or the crash of the reconfiguration coordinator, the nodes in group G_i will also periodically keep trying to get a DataAck message from the new group member, acknowledging that it has received all the data for the key range, and managed to make the view v_{i+1} ready.

7.2 Linearizable *Put* and *Get* Operations using Consistent Quorums

We adapted the ABD [23] algorithm, which implements an atomic register in a static asynchronous system, to work with consistent quorums in a dynamic replication group. Algorithms 4–6 illustrate our adaptation which provides linearizable *put* and *get* operations even during group reconfigurations. Any node in the system that receives a *put* or *get* operation request from a client, will act as an operation coordinator. The operation coordinator first locates the replicas for the requested key, by looking up the list of successors for the key, on the consistent hashing ring. Then, it

Algorithm 4 Operation coordinator (part 1)

Init: readAcks[k] $\leftarrow \emptyset$, writeAcks[k] $\leftarrow \emptyset$, val[k] $\leftarrow \perp$, $v_Q[k] \leftarrow \perp$, reading[k] $\leftarrow$ *false* ▷ $\forall k$ *key*

1: **on** ⟨GetRequest: k⟩ **do** ▷ *from client*
2: reading[k] $\leftarrow$ *true*
3: **send** ⟨ReadA: k⟩ **to** *all replicas of k determined by successor-list lookup*

4: **on** ⟨PutRequest: k, val⟩ **do** ▷ *from client*
5: val[k] $\leftarrow$ *val*
6: **send** ⟨ReadA: k⟩ **to** *all replicas of k determined by successor-list lookup*

7: **on** ⟨ReadB: $k, ts, val, view$⟩ **do**
8: readAcks[k] $\leftarrow$ readAcks[k] $\cup \{(ts, val, view)\}$
9: $v_Q[k] \leftarrow$ ***consistentQuorum***(*extractViewMultiset*(readAcks[k]))
10: **if** $v_Q[k] \neq \perp$ **then** ▷ *write phase must use view from read phase*
11: (t, v) $\leftarrow$ *highestTimestampValue*(readAcks[k], $v_Q[k]$)
12: **if** $\neg$ reading[k] **then**
13: **send** ⟨WriteA: k, t + 1, val[k], $v_Q[k]$⟩ **to** *all members of* $G_Q[k]$
14: **else if** $\neg$ *sameValueTimestamp*(readAcks[k], $v_Q[k]$) **then**
15: val[k] $\leftarrow$ v ▷ *read-impose*
16: **send** ⟨WriteA: k, t, val[k], $v_Q[k]$⟩ **to** *all members of* $G_Q[k]$
17: **else** ▷ *latest value already committed at a consistent quorum*
18: **send** ⟨GetResponse: k, v⟩ **to** *client*
19: *resetLocalState*(k) ▷ *readAcks, writeAcks, val,* v_Q*, reading*

engages in a two-phase quorum-based interaction with the replicas. The replicas are assumed to be the first r successors of the requested key, where r is the replication degree. This is usually the case in the absence of node dynamism, and the coordinator is then able to assemble a consistent quorum with these nodes. Otherwise, failing to assemble a consistent quorum, the coordinator retries the request with more nodes in the key's successor list. Note that not finding the replicas for the key, which could happen, e.g., during a network partition, does not compromise safety.

For a *get* operation, the coordinator reads the value with the latest timestamp from a consistent quorum with view $v_Q[k]$. The *highestTimestampValue* function takes a multiset of (*timestamp*, *value*, *view*) triples and

Algorithm 5 Replication group member

Init: *retrieve*(value[k], version[k]) ▷ $\forall k \in s \mid isInstalledView(v) \wedge v = \langle s, G \rangle$

1: **on** $\langle$ReadA: $k\rangle \wedge isReady(k)$ **do** ▷ *from coordinator*
2: **send** $\langle$ReadB: k, version[k], value[k], $view(k)\rangle$ **to** *coordinator*

3: **on** $\langle$WriteA: $k, ts, val, v_Q\rangle$ **do** ▷ *from coordinator*
4: **if** $ts >$ version[k] $\wedge$ $isInstalledView(v_Q)$ **then**
5: value[k] $\leftarrow$ val ▷ *update local replica*
6: version[k] $\leftarrow$ ts
7: **send** $\langle$WriteB: k, $view(k)\rangle$ **to** *coordinator*

Algorithm 6 Operation coordinator (part 2)

1: **on** $\langle$WriteB: $k, view\rangle$ **do**
2: writeAcks[k] $\leftarrow$ writeAcks[k] $\cup$ $\{view\}$
3: **if** $v_Q[k] =$ ***consistentQuorum***(writeAcks[k]) **then**
4: **if** reading[k] **then**
5: **send** $\langle$GetResponse: k, val[k]$\rangle$ **to** *client*
6: **else**
7: **send** $\langle$PutResponse: $k\rangle$ **to** *client*
8: *resetLocalState*(k) ▷ *readAcks, writeAcks, val, v_Q, reading*

returns a pair with the highest *timestamp* and associated *value*, among the triples whose *view* matches $v_Q[k]$, the view of the consistent quorum. The *sameValueTimestamp* function looks at the timestamps associated with values received from nodes in $v_Q[k]$, and checks whether all these timestamps are equal. If the coordinator sees values with different timestamps among a consistent quorum, a concurrent *put* operation must be in progress, or the coordinator of a previous *put* might have failed before managing to update all replicas. Since the coordinator cannot be sure that the latest value is already committed at a consistent quorum, it commits it himself (WriteA) before completing the *get* operation. This mechanism, known as *read-impose*, preserves linearizability by preventing a subsequent *get* from returning an old value by contacting a consistent quorum of nodes that didn't get the new value yet. In the absence of concurrent or incomplete *put* operations on the same key, the *get* operation completes in a single round since all

value timestamps received in a consistent quorum must be equal.

When a server has just joined the replication group, i.e., it installed the view but is still waiting for the data, we say that the view is *busy* (see Algorithm 3 lines 2 and 7). Before the view becomes *ready* the server will not reply to ReadA messages (see Algorithm 5 line 1). The *isInstalledView* function checks whether a given view is currently installed, and the *isReady* function checks whether the local view associated with a given key is currently ready. The *resetLocalState* function resets the state pertaining to an operation once a consistent quorum was assembled. In particular, the acknowledgement sets are reset to prevent additional "detections" of the same consistent quorum upon receiving subsequent acknowledgements.

For clarity, we omit from the algorithms details pertaining to breaking ties between value timestamps based on coordinator id, or ignoring orphan messages. We also omit illustrating operation timeout and retrial, e.g., when a coordinator does not manage to assemble a consistent quorum within a given timeout or when the operation is retried because a view change has occurred between the two phases, which we discuss next.

Concurrent View Changes during Operations

For a *put* operation, the coordinator first reads the highest value timestamp, t, from a consistent quorum with view v_Q. It then attempts to commit its value, X, using timestamp $t+1$, at a consistent quorum of the same view v_Q. In order to conservatively preserve linearizability as we adapt this two-phase protocol from a static replication group to a dynamic setting using consistent quorums, we require that the two phases of the protocol operate within the same view (see Algorithm 6 line 3). If the view of the replication group changes between the two phases, and the second phase is not anymore able to assemble a consistent quorum with the same view as the first phase, the operation is repeated so that the first phase can assemble a consistent quorum with the new view. Note that retrying the *put* operation ensures termination, however, to preserve linearizability, it is important that the *put* is retried with the *same* timestamp, $t+1$. Incrementing the timestamp again during retrial could give the appearance of multiple *put* operations for value X, as follows: a *get* observes and returns X, followed by a *put* which updates k to value Y using timestamp $t+2$, followed by

another *get* which observes Y, followed by the retrial of our *put* of X with a timestamp incremented to $t + 3$, and finally followed by another *get* which observes X again. This violates linearizability as the *put* of X does not appear to have occurred instantaneously. *Get* operations needing to perform a *read-impose* are retried in a similar manner during view changes.

Crash-recovery

Upon the initialization of a replication group member, the *retrieve* function loads from persistent storage any key-value data items that the node may have stored in a previous incarnation, before crashing.

When saving all algorithm meta-data in stable storage, crash-recovery is very similar to the node being partitioned away for a while. In both cases, when a node recovers or the partition is reconnected, the node has the same configuration and data as it had before the crash or network partition. Since they are partition tolerant, these algorithms already support crash-recovery.

7.3 Network Partitions and Inaccurate Failure Suspicions

A communication network may fail such that the network is fragmented into multiple isolated components [66]. During a network partition, the nodes of a distributed system are disconnected from each other, such that messages sent by nodes in one component to nodes in another component are systematically dropped. Abstractly, a partitioned network is a network which is allowed to lose arbitrarily many messages sent from one node to another. Once the communication failure is remedied, the partitioned components are reconnected and nodes may communicate again. A closely related situation is that of *inaccurate failure suspicions* where due to similar network failures or congestion, some nodes may suspect other nodes to have crashed after not receiving responses from them for long enough [49]. We say that a distributed protocol is *partition tolerant* if it continues to satisfy its correctness properties despite these adverse network conditions.

Paxos [130] and ABD [23] are intrinsically partition tolerant. Since they depend on majority quorums, operations issued in any partition component

that contains a majority of nodes will succeed, while operations issued in partition components containing only a minority of nodes will block. To maintain their partition tolerance when applying Paxos and ABD within a consistent hashing ring, we use consistent quorums to preserve their *safety* properties. Let us turn to *liveness* properties and examine two facts. First, any topology maintenance protocol used to repair the consistent hashing ring – in order to preserve its consistent hashing invariant as nodes join the system or fail – is inherently fault tolerant. Second, it is impossible for any node in a distributed system to discern between a situation where another node has crashed or it is just partitioned away. As a consequence of these two facts, a network partition will cause a consistent hashing ring topology to split into two disjoint independent rings forever isolated.

For example, a network partition may split a consistent hashing ring in three different rings. Assuming a replication degree of three, it is easy to imagine how some replication groups of the original ring are partitioned in three minority components, with one node in each, therefore getting stuck forever since the replication group cannot be reconfigured in any of the partition components due to the lack of a majority of acceptors. Similarly, no read or write operations could be completed either, leading to permanent service unavailability.

To preserve the liveness properties of Paxos and ABD when adapting them to operate within replication groups determined automatically by consistent hashing, we employ a ring unification algorithm [202] which repairs the consistent hashing ring topology after a transient network partition. Once a network partition has ceased, replication group views are reconciled with the node replication responsibilities dictated by consistent hashing, causing the replication group views to converge to the overlay network topology. This makes our overall solution partition tolerant, satisfying both safety and liveness properties.

7.4 Safety

A *safety* property [126] states that something *will not* happen. Informally, a safety property specifies that "nothing bad" will happen during the execution of an algorithm, i.e., important correctness invariants are preserved.

We examine the safety properties of our consistent-quorums-based algorithms through the following statements.

Lemma 1. *After a successful (terminated) reconfiguration $(v_i \Rightarrow v_{i+1})$, at most a minority of nodes in group G_i may still have view v_i installed.*

Proof. If reconfiguration $(v_i \Rightarrow v_{i+1})$ terminated, it must have completed phase 3, thus at least a majority of nodes in group G_i must have installed view v_{i+1} (or yet a newer view). Therefore, at most a minority of nodes in group G_i may still have view v_i installed. □

Lemma 2. *For any replication group G of a key range s, there cannot exist two disjoint majorities (w.r.t. group size $|G|$) with consistent views, at any given time.*

Proof. Case 1 (*same view*): no view is ever installed on more than $|G|$ nodes. Therefore, there can never exist two or more disjoint majorities with the same consistent view.

Case 2 (*consecutive views $v_i \Rightarrow v_{i+1}$*): by phase 3 of the algorithm, a majority for view v_{i+1} cannot exist before view v_{i+1} is installed at a majority of nodes in G_i. Once a majority of nodes in G_i have installed v_{i+1}, they now constitute a majority in v_{i+1} and by Lemma 1 at most a minority of nodes in G_i still has view v_i installed, thus two disjoint majorities for consecutive views v_i and v_{i+1} cannot exist simultaneously.

Case 3 (*non-consecutive views $v_i \rightsquigarrow v_{i+k} \mid k > 1$*): views are always installed in sequence. For the replication group to reach view v_{i+k} from view v_i, a majority of nodes in G_i must have first applied a reconfiguration $(v_i \Rightarrow v_{i+1})$. At that particular time, by Case 2, a consistent majority for view v_i ceased to exist. □

Lemma 3. *For any replication group G of a key range s, no sequence of network partitions and mergers may lead to disjoint consistent quorums.*

Proof. By the algorithm, a majority of nodes in group G_i must be available and connected for a reconfiguration $(v_i \Rightarrow v_{i+1})$ to succeed. Thus, a reconfiguration of group G_i can only occur in partition components containing a majority of the nodes in G_i, while nodes in any minority partitions are stuck in view v_i, unable to decide a reconfiguration.

CASE 1: network partition splits group G_i in multiple minority partitions so no reconfiguration can occur in any partition; when the partitions merge, by CASE 1 of Lemma 2 we cannot have disjoint consistent quorums.

CASE 2: a sequence of network partitions and reconfigurations (in a majority partition component M) results in multiple minority partitions that later merge (independently from M). Because every reconfiguration generates a new view, the views available in different minority partitions are all distinct and thus, their union cannot form a consistent quorum (disjoint from a consistent quorum in M). □

From Lemmas 2 and 3, we have Theorem 1 which gives a sufficient guarantee for linearizability.

Theorem 1. *No two disjoint consistent quorums may exist simultaneously, for any key replication group. Hence, any two consistent quorums always intersect.*

From Theorem 1 it follows that consistent quorums fulfill the core safety assumption made by quorum-based protocols, namely the quorum intersection principle. This suggests that consistent quorums may be used to adapt any static quorum-based protocol to operate correctly in dynamic replication groups. Group reconfiguration can be initiated automatically by consistent hashing, potentially endowing the static protocol with properties of scalability and self-management.

7.5 Liveness

A *liveness* property [10, 126] states that something *must* happen. Informally, a liveness property specifies that "something good" will happen, eventually, during the execution of an algorithm, e.g., the system converges to a legitimate state or the algorithm terminates.

We examine the liveness properties of our consistent-quorums-based algorithms through the following statements.

Lemma 4. *Provided a consistent quorum of the current view v_i is accessible, a group reconfiguration $(v_i \Rightarrow v_{i+1})$ will eventually terminate.*

Proof. Given that the reconfiguration coordinator does not crash, and a majority of nodes in group G_i is accessible, with periodic retrials to counter for message loss, the coordinator will eventually succeed in installing view v_{i+1} on a majority of nodes in G_{i+1}. If the coordinator crashes during phase 1 or phase 2, another node will become coordinator guided by consistent hashing, and it will take over the reconfiguration and complete it by periodically retrying the reconfiguration protocol with the acceptor group G_i until a majority becomes accessible. If the coordinator crashes during phase 3, all other nodes in G_i are now in end-game mode and will effectively act as coordinators and oversee the completion of the reconfiguration. Once the reconfiguration completes, new reconfigurations may be proposed in order to reconcile the group membership with the ring membership. □

Corollary 1. *Provided that all network partitions cease, every ongoing group reconfiguration will eventually terminate.*

Proof. After all network partitions merge, even groups that had been split into multiple minority partitions are now merged, thus satisfying the premise of Lemma 4. □

Lemma 5. *Provided a consistent quorum is accessible, put and get operations will eventually terminate.*

Proof. Given that an operation's coordinator does not crash before the operation completes, it will periodically retry to assemble a consistent quorum for the operation's key, until one becomes available and connected. When a client detects the crash of the operation coordinator, the client retries its operation with a different coordinator. □

From Lemmas 4 and 5, and Corollary 1, we have Theorem 2 regarding the termination of protocols based on consistent quorums.

Theorem 2. *For any key replication group, provided a consistent quorum is available and connected, any put and get operations issued in the same partition, and any group reconfigurations will eventually terminate. If the network is fully connected, all operations and all group reconfigurations will eventually terminate.*

From Theorem 2 it follows that under reasonable network conditions, protocols using consistent quorums satisfy termination, arguably one of the most important liveness properties of any distributed algorithm. While machines and network failures are not uncommon in a datacenter network, unrecoverable failure scenarios tend to be not as extreme as to invalidate the availability assumptions we made in this chapter. This suggests that such protocols based on consistent quorums are practical and fit for deployment in a cloud computing environment.

Chapter 8

CATS System Architecture and Testing using Kompics

To validate and evaluate the technique of consistent quorums, we have designed and built the CATS system, a scalable and self-organizing key-value store which leverages consistent quorums to provides linearizable consistency and partition tolerance. CATS was implemented in Kompics Java [19] which, on the one hand, allows the system to readily leverage multi-core hardware by executing concurrent components in parallel on different cores and, on the other hand, enables protocol correctness testing through whole-system repeatable simulation.

In this chapter, we describe the software architecture of the CATS system as a composition of protocols and service abstractions, and we discuss various system design choices. We first show the component architecture designated for distributed production deployment, and then we show the architecture for local interactive stress testing and whole-system simulation. This chapter aims to present the CATS system as a concrete case study and a demonstration of using the Kompics methodology for the design, development, testing, debugging, and deployment of distributed systems.

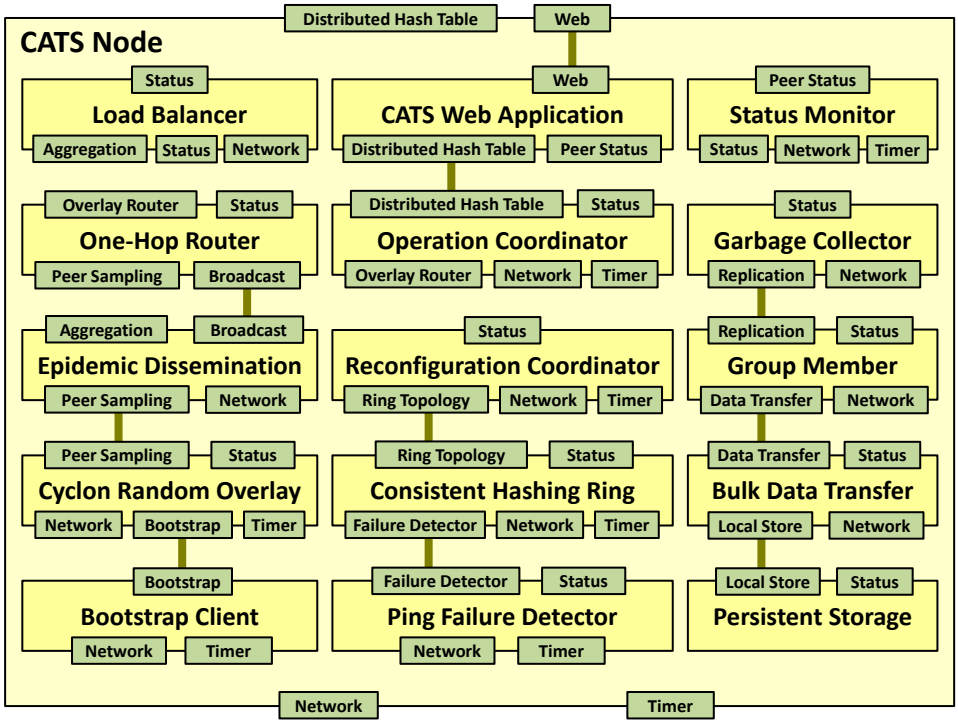

Figure 8.1. System architecture: protocol components of a single CATS node.

8.1 Protocol Components and System Design

We now give a high-level overview of the CATS system architecture and we discuss some of the decisions we made during the design of the system. The system design assumes a trusted deployment infrastructure, such as a datacenter network. Figure 8.1 illustrates the main protocol components of a single CATS server node. In addition to this, a client library, linked with application clients, handles the location of relevant servers in the system as well as relaying, and occasionally retrying, *put* and *get* operations on behalf of the application. The CATS Node is a composite component reused within multiple environments, such as in the production deployment architecture shown later in Figure 8.2, or the whole-system simulation architecture and the stress testing architecture shown in Figure 8.6. We now review the role of the different protocol components operating within each CATS Node.

Topology Maintenance

A fundamental building block for CATS is the Consistent Hashing Ring module which implements a fault tolerant and partition tolerant ring topology maintenance algorithm [202]. It subsumes a periodic stabilization protocol [208] for maintaining the ring pointers under node dynamism, incorporating structural changes to the ring as dictated by consistent hashing [120]. Since periodic stabilization does not cater for network partitions and mergers, it is possible that during a transient network partition, the periodic stabilization protocol reorganizes the ring into two disjoint rings. We use a ring unification protocol [201, 202] to repair pointers and converge to a single ring after a network partition. As a result, CATS's Consistent Hashing Ring overlay is partition tolerant. Both periodic stabilization and ring unification are best-effort protocols: they do not guarantee lookup consistency [87, 203] and may lead to non-overlapping quorums as we described in Section 6.5. We mitigate these inconsistencies by using consistent quorums and consensus-based reconfiguration of replication groups.

Failure Detection

The Consistent Hashing Ring module relies on a Ping Failure Detector protocol component to monitor its ring neighbors, namely the node's predecessor and a list of successors. Since CATS uses a successor-list replication scheme, a node's successors on the ring are likely to be the replicas for the data items the node is responsible of storing. The failure detector is *unreliable* [49] and it can inaccurately suspect monitored nodes to have crashed.

Peer Sampling and Efficient Routing

Another foundational component of CATS is the Cyclon Random Overlay. This module encapsulates the Cyclon gossip-based membership protocol [227]. Cyclon implements a Peer Sampling service which provides every node with a continuous stream of random nodes in the system [112, 115]. We use this uniform stream of peers to build a full membership view of the system in the One-Hop Router component. This enables an Operation Coordinator to very efficiently – in one hop [96] – look up the responsible replicas for a given key-value pair. The full view at each node is not required to

immediately reflect changes in node membership and so, it can be stale for short periods, for large system sizes. To mitigate the staleness of views, and for lookup fault tolerance, a node can forward a lookup request to multiple nodes in the view – while still employing greedy routing – and use the first lookup response. A membership change detected in a local neighborhood is propagated to the rest of the system by the Epidemic Dissemination module. This module also relies on the random peer sampling service to quickly and robustly broadcast churn events to all nodes [69].

Replication Group Reconfiguration

The Group Member module handles consistent replication group membership views and view reconfigurations, acting as an *acceptor* and *learner* in Algorithms 2 and 3. View reconfigurations are *proposed* – as shown in Algorithm 1 – by the Reconfiguration Coordinator component, which monitors the state of the Consistent Hashing Ring and tries to reconcile the replication group membership with the ring membership. When a group G replicating keys $k \in (x, y]$ has to be reconfigured to G', any member of G can propose the new configuration G'. To avoid multiple nodes proposing the same reconfiguration operation, e.g., when they all detect the crash of one group member, we employ a *selfish* mechanism, whereby only the node responsible for a key-range replication group – according to consistent hashing – proposes a reconfiguration in this group. In this mechanism, the node responsible for the keys in range $(x, y]$ is the only node in charge with proposing the new configuration. If the reconfiguration does not succeed, e.g., if the network is partitioned, the responsible node retries the reconfiguration operation periodically. Due to inaccurate failure suspicions, if multiple nodes consider themselves responsible for the same key range, our consensus-based group reconfiguration will make sure that only one reconfiguration operation will succeed. Apart from averting multiple nodes from proposing the same reconfiguration, this mechanism has an added benefit. In consistent hashing, there is always at least one node responsible for each key range, and this node will keep attempting to repair the replication group for that key range. Periodic retrials will make sure that replication group reconfigurations will eventually terminate for all key ranges, despite message loss and transient network partitions.

Bulk Data Transfer

The Bulk Data Transfer component implements optimizations for fetching data to the new node joining a replication group after a view reconfiguration. The new node transfers data in parallel from existing replicas, by evenly dividing the requested data among all replicas. For example, assume that node D joins group $G = \{A, B, C\}$ to replicate key range $(1, 900]$. Node D requests key range $(1, 300]$ from node A, range $(300, 600]$ from B, and range $(600, 900]$ from C. This results in better bandwidth utilization, fast data transfer due to parallel downloads, and it avoids disproportionately loading a single replica. If a replica fails during data transfer, the requesting node reassigns the requests sent to the failed replica, to the remaining alive replicas. Before transferring values, each replica first transfers keys and their timestamps to the new node. For each key, the new node retrieves the latest value from the replica with the highest timestamp. This avoids redundant transfers as well as unnecessary transfers of stale values from existing replicas to the new replica, thus lowering bandwidth usage.

Persistent Storage

The Group Member module also handles operation requests coming from an Operation Coordinator – see Algorithms 4 and 6 – hence acting as a replica storage server in Algorithm 5. In serving operation requests, it relies on a local key-value store provided by the Persistent Storage module. CATS provides four different implementations of the Persistent Storage module. The first is based on SleepyCat, the Java Edition of BerkeleyDB [35], the second leverages Google's LevelDB [135], the third uses bLSM [199] from Yahoo! Research, and the fourth uses an in-memory sorted map. In Chapter 9 we evaluate the in-memory implementation. The persistent stores are used to implement single-node and system-wide recovery protocols. Crash-recovery, while using persistent storage, is very similar to the node being partitioned away for a while. In both cases, when a node recovers or the partition heals, the node has the same configuration and data as it had before the crash or partition. Therefore, our algorithms already support crash-recovery since they are partition tolerant. System-wide coordinated shutdown and recovery protocols are important in cloud environments.

Put and *Get* Operations Coordinator

To implement *put* and *get* operations we use the ABD algorithm [23, 146], a quorum-based atomic register protocol, which we augmented with consistent quorums. The node carrying out the ABD protocol with the replicas is called the *coordinator* for a request. The CATS design allows multiple options as to which node acts as the coordinator. A client itself can act as a coordinator. While this scheme has low message complexity and latency, it requires the client to have knowledge of all the servers and their placement on the identifier space. Such a solution may not scale when the number of clients and servers becomes large, as it is limited by the need for clients to open connections to all servers.

As an alternative, a client can maintain a cache of servers, and it can send an operation request for key k to a randomly selected server S from the cache. Node S may or may not be a replica for k. If S is a replica for k, it can act as the coordinator for the request performing ABD with the replicas and sending the result back to the client. If S is not a replica for k, it can either act as the coordinator, or forward the request to node R, one of the replicas for key k, which can then act as the coordinator. Here, if the request is forwarded to R, the latency of the operation will be higher by one message delay as an extra hop is taken. Also, R will have to open a connection to the client to send the operation response. On the other hand, if S acts the coordinator, the message complexity will be higher by two messages since S is not one of the replicas and all ABD messages will have to be sent to remote nodes. Yet, the operation latency is lower and no new connections are required. We have implemented all of the aforementioned variations for placing the Operation Coordinator component, and we allow the user to select the desired mechanism via a configuration parameter.

Load Balancing

As a result of machine failures and changes in operation request workload, the distribution of storage responsibilities among the nodes in the system may become skewed. Systems built on consistent hashing can balance storage and request load by employing the concept of *virtual nodes* as in Dynamo [67], Riak [30], or Voldemort [80]. Each physical machine joins the

system as multiple virtual nodes using different identifiers. The number of virtual nodes hosted by a physical machine, and the placement of each virtual node on the ring, largely ameliorate any load imbalance.

In CATS, load balancing is assisted by the Load Balancer component which relies on the Epidemic Dissemination module to aggregate statistics about the load at different nodes in the system. These statistics are then used to make load balancing decisions, such as moving virtual nodes to different positions on the consistent hashing ring, creating new virtual nodes on lightly loaded machines, or removing virtual nodes from overloaded machines. Load balancing enables the support of *range queries* in the Operation Coordinator, by allowing the keys to be stored in the system in their natural sort order, without hashing, and removing the load imbalances arising from skewed key distributions. Load balancing and range queries in CATS are the subject of work in progress.

Garbage Collection of old Views

The Garbage Collector module implements a periodic mechanism of garbage collecting (GC) old replication group views, in order to avoid unnecessary copies of data lingering around in the system as a result of transient network partitions. For example, if a replica R from group G gets partitioned away, G may still have a consistent quorum in a majority partition. Therefore, G can be reconfigured, and thus evolve into subsequent new group views. After the network partition ceases, the old view stored at node R is stale, and thus considered garbage. Garbage collection runs periodically and it makes sure to remove data only for those views which were already reconfigured and to which node R no longer belongs.

Bootstrapping, Web-based Interaction, and Status Monitoring

We use a Bootstrap Server to construct an initial configuration of replication groups for a newly started instance of CATS. The Status Monitor component periodically aggregates the state of each module, and sends it to the CATS Web Application which renders it in HTML and exposes it through a web interface. These utilities are similar to those provided by the Kompics P2P framework (see Section 3.6.4); some of them are completely reused.

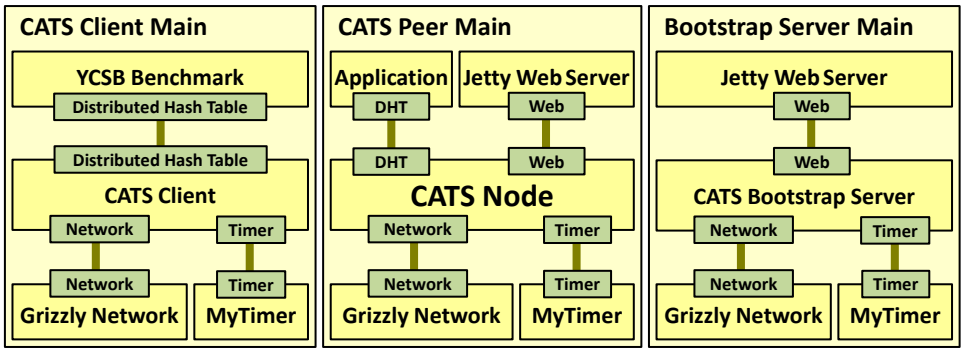

Figure 8.2. CATS system architecture for distributed production deployment. On the left we have a YCSB benchmark client and on the right the bootstrap server.

8.2 Distributed Production Deployment

In Figure 8.2 we illustrate the CATS component architecture designated for distributed production deployment. In the center we have the executable process CATS Peer Main, which contains the CATS Node, a network component embedding the Grizzly NIO framework [218], a timer component based on the Java timer service, a web interface component embedding the Jetty web server [219], and an Application component which may embed a command-line interface (CLI) or a graphical user interface (GUI).

The CATS Node provides a DHT service abstraction. By encapsulating all the protocol components – discussed in the previous section – behind the DHT port, the rest of the system is oblivious to the complexity internal to the CATS Node component. The benefit of encapsulation and nested hierarchical composition is even more pronounced in the whole-system simulation and stress testing architectures of Figure 8.6 where multiple CATS Nodes are manipulated as if they were simple components.

The JettyWebServer enables users to monitor the status of a node's components and issue interactive commands to the node using a web browser. The CATS Node exposes its status through a Web port. The HTML page representing the node's status will typically contain hyperlinks to its neighbor nodes and to the bootstrap and monitoring servers. This enables users and developers to browse the set of nodes over the web, and inspect the state of each remote node. An example is shown later in Figure 8.4.

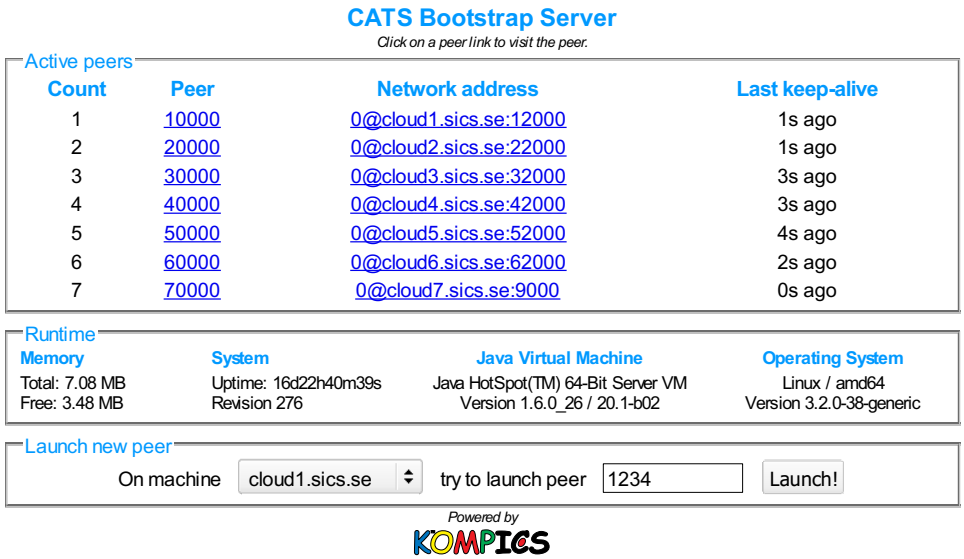

Figure 8.3. Interactive web interface exposed by the CATS bootstrap server.

On the left side of Figure 8.2 we have the architecture of a CATS client generating *put* and *get* workloads using the YCSB benchmark [60]. We have used this type of load generator clients in the performance evaluation of CATS presented in Chapter 9. The architecture of the CATS bootstrap server appears on the right side of Figure 8.2.

Figure 8.3 illustrates the web interface at the CATS bootstrap server. Here, users can see the list of active peers, with the last time a keep-alive message was receive from each. User can navigate to any live peer, and even launch a new peer on one of a set of preconfigured cluster machines.

Figure 8.4 shows the web interface exposed by a CATS peer. Here, users can interact with the local node, and inspect the status of its consistent hashing ring topology, its key range replication responsibilities, as well as failure detection statistics, and various information about the run-time system, such as the size of the heap and the amount of free memory.

Users may kill a peer – using its web interface – in order to test or to demonstrate a system reconfiguration. Users may also initiate *put* and *get* operations using the Distributed Hash Table input form. The interactive operation results and statistics are illustrated in Figure 8.5.

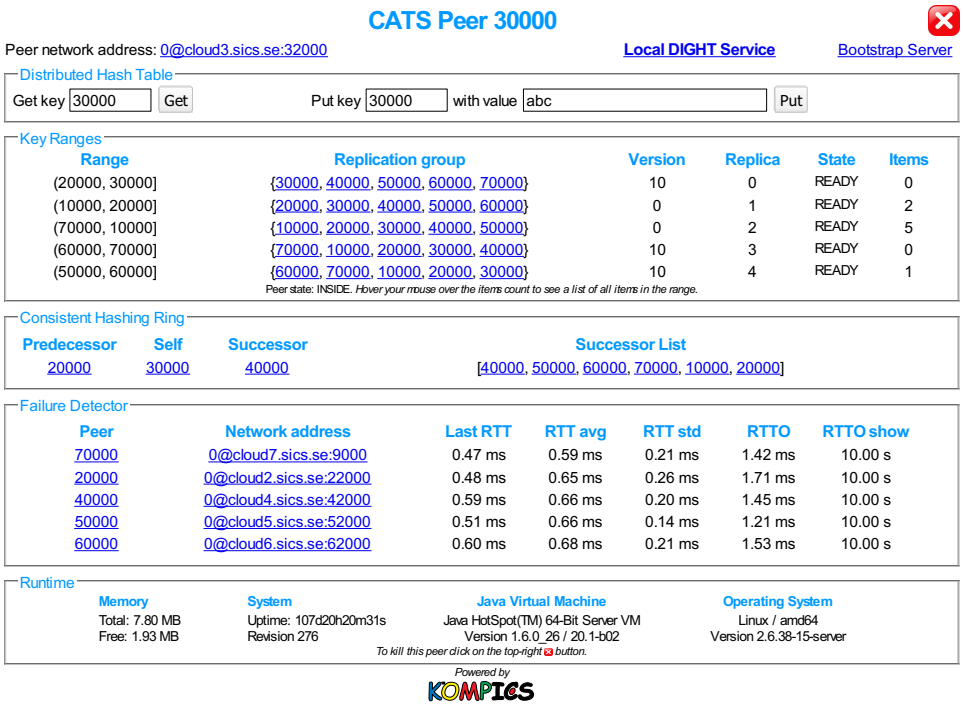

Figure 8.4. Interactive web interface exposed by one of the CATS peers.

Figure 8.5 describes the results of interactive *put* and *get* operations. First, the operation response or an error message is printed. Then, the successor list returned by the lookup for the requested key is shown together with the latency of the lookup. Fine-grained timing statistics are shown for both the read and the write phases of the operation. This illustrates how quorums are formed in each phase, showing the latency of the round-trip times to each quorum member, in the order in which acknowledgements were received. In this example the replication degree was five, so a majority quorum consists of three nodes. Finally, the total operation latency is shown together with the consistent quorum accessed by the operation.

For the *get* operation, the returned item value or an error message is printed first. The remaining statistics are very similar to the ones output by a *put* operation. A notable difference is that for a *get* operation, if all

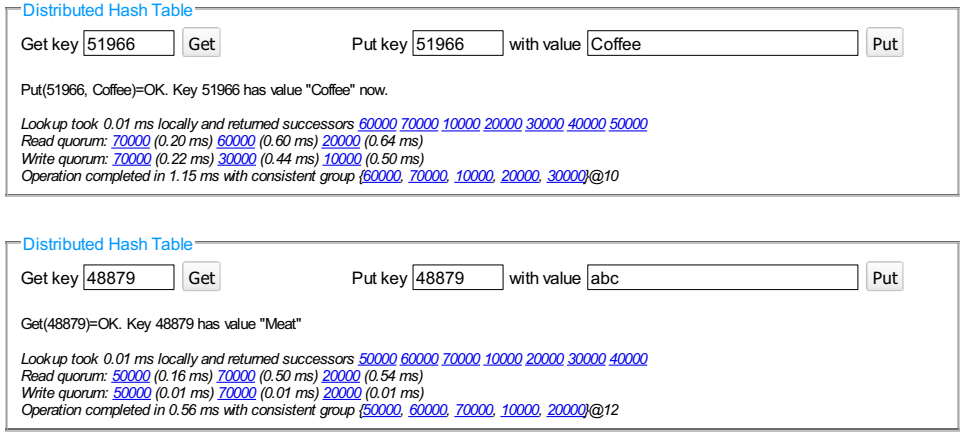

Figure 8.5. *Put* and *Get* operations executed through the CATS peer web interface.

the item timestamps seen in the read-phase quorum are the same, then the write phase need not be performed at all. This is the case in this example, as we can observe from the write phase latency of under 10 μs.

We have deployed and evaluated CATS on our local cluster, on the PlanetLab testbed [33], and on the Rackspace cloud [186]. Using the web interface to interact with a LAN deployment of CATS – configured with a replication degree of five – resulted in sub-millisecond end-to-end latencies for *get* and *put* operations. This includes the LAN latency of two message round-trips (i.e., 4×the one-way latency), message serialization and deserialization (4×), compression and decompression (4×), and Kompics run-time overheads for message dispatching and execution. In a 1 KB-value read-intensive workload, generated on Rackspace by 32 clients, CATS scaled linearly to 96 server machines providing just over 100,000 reads/sec.

8.3 Whole-System Repeatable Simulation and Local Interactive Stress Testing

We now show how we can reuse the CATS Node and all of its subcomponents, without modifying their code, to execute the system in simulation mode for testing, stepped debugging, or for repeatable simulation studies.

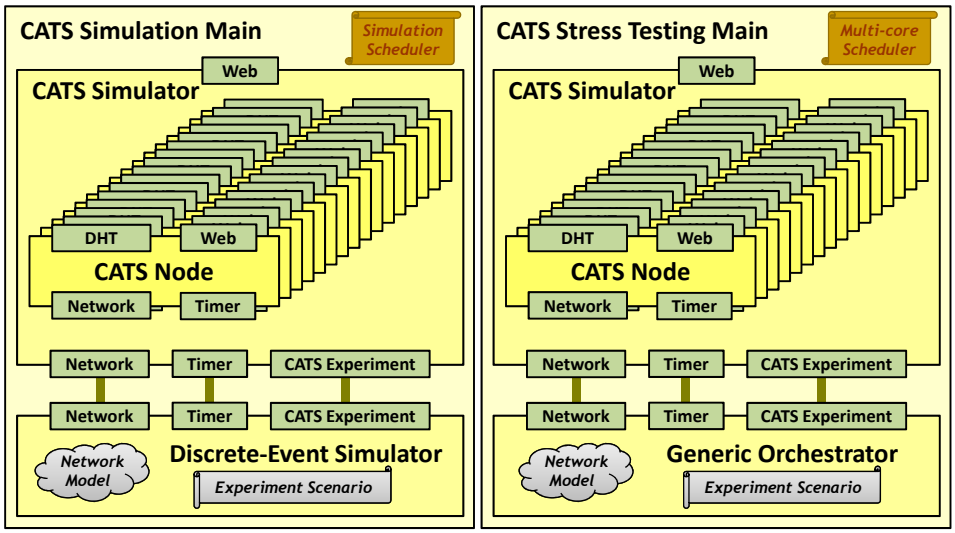

Figure 8.6. CATS architecture for whole-system simulation (left) / interactive stress test execution (right). All nodes and servers execute within a single OS process. On the left, the system is executed deterministically in simulated virtual time. On the right, the system is executed in real time leveraging multiple cores.

The left side of Figure 8.6 shows the component architecture for simulation mode. Here, a generic Discrete-Event Simulator interprets an experiment scenario and issues command events to the CATS Simulator component. A command – triggered through the CATS Experiment port – may tell the CATS Simulator to create and start a new CATS Node, to stop and destroy an existing node, or to instruct an existing node to execute a DHT operation. The ability to create and destroy node subcomponents in the CATS Simulator is clearly facilitated by Kompics' support for dynamic reconfiguration and nested hierarchical composition. The generic Discrete-Event Simulator component also provides the Network and Timer abstractions, and it can be configured with a custom network model, in order to emulate realistic conditions of network latency and bandwidth as discussed in Section 4.3.1.

This whole architecture is executed in simulation mode, i.e., using a simulation component scheduler, which executes all components that have received events, and when it runs out of work, it passes control to the Discrete-Event Simulator to advance the simulation time – see Section 4.1.2.

First Previous Current Next Last Up

▶ 534 5100 10123 ■ ..

Pred	Node	SuccList	Replica 0	Replica 1	Replica 2
45	10	[15,20,30,35,40]	(45,10]{10,15,20}@0	(40,45]{45,10,15}@0	(35,40]{40,45,10}@0
10	15	[20,30,35,40,45]	(10,15]{15,20,30}@0	(45,10]{10,15,20}@0	(40,45]{45,10,15}@0
15	20	[30,35,40,45,10]	(15,20]{20,30,35}@0	(10,15]{15,20,30}@0	(45,10]{10,15,20}@0
NIL	**25**	[30]			
20	30	[35,40,45,10,15]	(20,30]{30,35,40}@0	(15,20]{20,30,35}@0	(10,15]{15,20,30}@0
30	35	[40,45,10,15,20]	(30,35]{35,40,45}@0	(20,30]{30,35,40}@0	(15,20]{20,30,35}@0
35	40	[45,10,15,20,30]	(35,40]{40,45,10}@0	(30,35]{35,40,45}@0	(20,30]{30,35,40}@0
40	45	[10,15,20,30,35]	(40,45]{45,10,15}@0	(35,40]{40,45,10}@0	(30,35]{35,40,45}@0

Configuration is INVALID.
Time: @Peer: Message
5000: @25: Sent BOOT_REQ to 0@127.0.0.1:8081
5059: @25: 9223372036854775807->BOOT_RESP(false)
5059: @25: Finding my succ with insider 45.
5100: @25: LOOK_RESP(25)=[30, 35, 40, 45, 10], req 1
5100: @25: Joining ring with successor 30.

Figure 8.7. CATS global state snapshot immediately after a node joins.

Using the same experiment scenario and the same network model used in simulation mode, we can execute the entire system in local interactive stress testing mode. The right side of Figure 8.6 shows the component architecture for local interactive stress testing. This is similar to the simulation architecture, however, our multi-core work-stealing component scheduler is used – see Section 4.1.1 – and the system executes in real time.

8.4 Simulation-Based Correctness Tests

We leveraged Kompics' support for whole-system repeatable simulation, described in Section 4.3, for testing the correctness of the CATS replication group reconfiguration algorithms. We devised a wide range of experiment scenarios comprising concurrent reconfigurations and failures, and we used an exponential message latency distribution with a mean of 89 ms. We verified stochastically that our algorithms satisfied their safety invariants and liveness properties, in all scenarios, for one million RNG seeds.

First Previous Current Next Last Up

▶ 11249 11386 12252 ■ ..

Pred	Node	SuccList	Replica 0	Replica 1	Replica 2
45	10	[15,20,30,35,40]	(45,10]{10,15,20}@0	(40,45]{45,10,15}@0	(35,40]{40,45,10}@0
10	15	[20,30,35,40,45]	(10,15]{15,20,30}@0	(45,10]{10,15,20}@0	(40,45]{45,10,15}@0
15	20	[25,30,35,40,45]	(15,20]{20,25,30}@1	(10,15]{15,20,30}@0	(45,10]{10,15,20}@0
20	25	[30,35,40,45,10]	(20,25]{25,30,35}@1	(15,20]{20,25,30}@1	
25	30	[35,40,45,10,15]	(25,30]{30,35,40}@1	(20,25]{25,30,35}@1	(10,15]{15,20,30}@0 (15,20]{20,25,30}@1
30	35	[40,45,10,15,20]	(30,35]{35,40,45}@0	(25,30]{30,35,40}@1	(20,25]{25,30,35}@1
35	40	[45,10,15,20,30]	(35,40]{40,45,10}@0	(30,35]{35,40,45}@0	(25,30]{30,35,40}@1
40	45	[10,15,20,30,35]	(40,45]{45,10,15}@0	(35,40]{40,45,10}@0	(30,35]{35,40,45}@0

Configuration is INVALID.
Time: @Peer: Message
11386: @25: 40->DATA((20,25]) for Split to (20,25]{25,30,35}@1 + (25,30]{30,35,40}@1
11386: @25: READY_DATA((20,25]) in Split to (20,25]{25,30,35}@1 + (25,30]{30,35,40}@1

Figure 8.8. CATS global state snapshot during reconfiguration.

During simulation, we monitor the global state of the system and whenever a part of the state is updated, a new global state snapshot is dumped into an HTML file. The collection of snapshots dumped during an execution allows us to time-travel forward and backward through the execution, and to observe how the state of a particular node is updated in response to receiving a particular message. Examples of such global state snapshots are shown in Figures 8.7, 8.8, and 8.9. There are three sections in any snapshot file. At the top there is a navigation menu that allows us to go back to the previous snapshot, forward to the next snapshot, as well as jump directly to the first or the last snapshot or to the list of all snapshots. Each snapshot is identified by the virtual time at which it was taken.

The next section contains the actual global state snapshot. Here we have a table row for each node in the system containing state relevant to the reconfiguration protocols, namely the node's predecessor and successor list, together with all its installed views. The various pieces of state are highlighted with different colors depending on whether they match or not the "ground truth" computed by a validator for each experiment scenario. If all the relevant state for each node matches the expected "ground truth",

First Previous Current Next Last Up

▶ 20560 25246 N/A ■ ..

Pred	Node	SuccList	Replica 0	Replica 1	Replica 2
45	10	[15,20,25,30,35]	(45,10]{10,15,20}@0	(40,45]{45,10,15}@0	(35,40]{40,45,10}@0
10	15	[20,25,30,35,40]	(10,15]{15,20,25}@1	(45,10]{10,15,20}@0	(40,45]{45,10,15}@0
15	20	[25,30,35,40,45]	(15,20]{20,25,30}@1	(10,15]{15,20,25}@1	(45,10]{10,15,20}@0
20	25	[30,35,40,45,10]	(20,25]{25,30,35}@1	(15,20]{20,25,30}@1	(10,15]{15,20,25}@1
25	30	[35,40,45,10,15]	(25,30]{30,35,40}@1	(20,25]{25,30,35}@1	(15,20]{20,25,30}@1
30	35	[40,45,10,15,20]	(30,35]{35,40,45}@0	(25,30]{30,35,40}@1	(20,25]{25,30,35}@1
35	40	[45,10,15,20,25]	(35,40]{40,45,10}@0	(30,35]{35,40,45}@0	(25,30]{30,35,40}@1
40	45	[10,15,20,25,30]	(40,45]{45,10,15}@0	(35,40]{40,45,10}@0	(30,35]{35,40,45}@0

Configuration is VALID.

Figure 8.9. CATS global state snapshot after reconfiguration completes.

the validator prints that the configuration is VALID; if not, it prints INVALID.

The final section of the snapshot file contains the logs output by each peer in the system, after the time of the previous snapshot, and up to, and including, the time of the current snapshot. The logs output at the exact time of the current snapshot are likely corresponding to actions that lead to the state update which caused the current snapshot, therefore they are highlighted in blue to make them easier to spot when debugging.

In Figure 8.7 we show a snapshot of the global state of CATS right after a new node joined the system and a reconfiguration is triggered as a result. In this example, node 25 just joined, and because it is not yet part of any replication group, all the views that should contain node 25 but do not yet, are deemed invalid. Similarly, none of the state of node 25 is valid.

In Figure 8.8 we show a snapshot of the global state of CATS during reconfiguration. At this point, node 25 has successfully joined the replication group and installed the view (20,25]{25,30,35}@1; it has also just received all data for range (20,25]. In contrast, node 25 has also installed view (15,20]{20,25,30}@1 but it is still waiting for the data at this point, which is indicated by the yellow highlighting of the range. Finally, at this point in time, node 25 is yet to install view (10,15]{15,20,25}@1.

In Figure 8.9 we show a snapshot of the global state of CATS after the reconfiguration protocol was completed, and the global state is now VALID.

Chapter 9

Scalability, Elasticity, and Performance Evaluation

In this chapter we present a performance evaluation of the CATS system implemented in Kompics Java. The system is subject to several workloads with a different mix of *put* and *get* operations. We measure the system throughput and the average latency of *put* and *get* operations while varying the operation request rate proportionally to the amount of data loaded into the system. We used the YCSB benchmark [60] as a load generator.

Next, we present a scalability study. We deployed CATS on Rackspace cloud servers [186], and we repeatedly doubled the number of used servers from three to ninety six. We measured the system's throughput while increasing the workload proportionally to the system size. We validated the elasticity of CATS by adding and removing servers while the system was running and subject to a live workload. We also measured the performance overhead of providing atomic consistency using consistent quorums.

Finally, we compare CATS with Cassandra [125], a scalable and self-organizing key-value store which has a very similar architecture to CATS, but which guarantees only eventual consistency for its data operations.

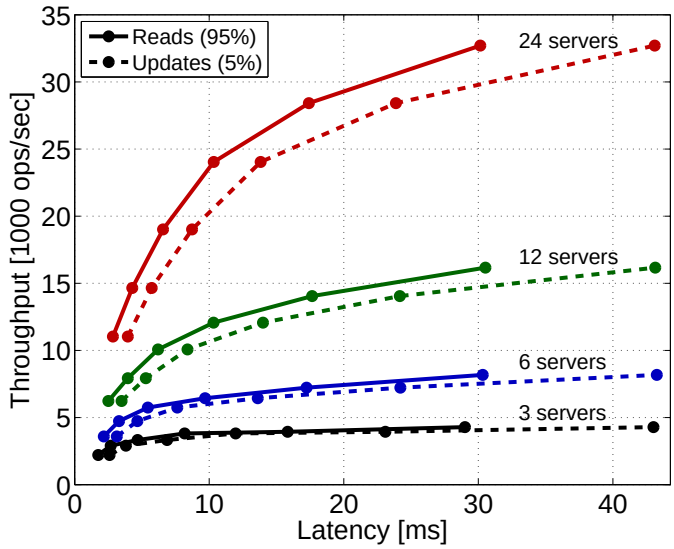

Figure 9.1. CATS performance under a read-intensive workload.

9.1 Benchmark and Experimental Setup

All experiments were conducted on the Rackspace cloud infrastructure, which is based on Xen virtualization [29]. We used up to 128 servers with 16 GB RAM, the largest instance available at the time, in order to guarantee that each CATS server had all the resources of a physical machine available to itself. This was necessary in order to minimize experiment variability.

In all experiments we used the YCSB [60] benchmark as a load generator. We defined two workloads with a uniform distribution of keys: a read-intensive workload with 95% reads and 5% updates, and an update-intensive workload comprising of 50% reads and 50% updates. The dataset size was set such that the data could fit in main memory. We chose to perform updates instead of inserts, to keep the data set constant and minimize variability due to Java garbage collection. This choice is without loss of generality since CATS uses the same *put* protocol for updates and inserts. Unless otherwise specified, we used data values of size 1 KB and the replication degree was three. To obviate the need for load-balancing, we placed the servers at equal distance on the consistent hashing ring.

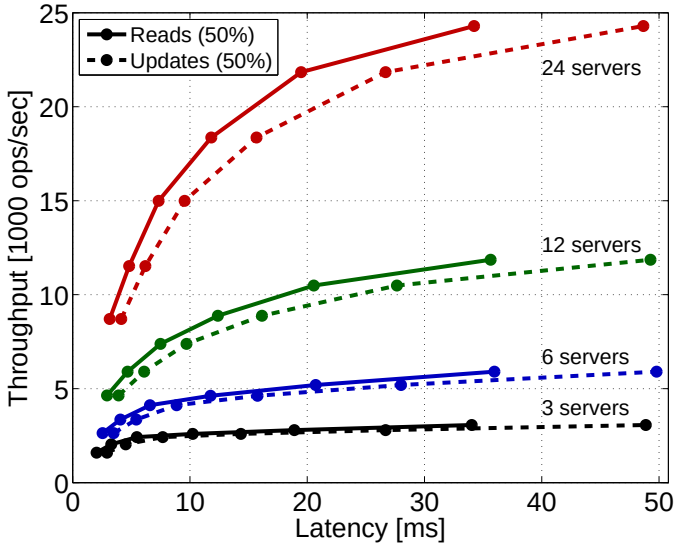

Figure 9.2. CATS performance under an update-intensive workload.

9.2 Throughput and Latency

In the first set of experiments, we measured the performance of CATS in terms of average operation latency and total throughput. We increased the load, i.e., the dataset size and the operation request rate, proportionally to the number of servers, by increasing the number of key-value data items initially inserted into CATS, and the number of YCSB clients, respectively. For example, we loaded 300,000 items and used one YCSB client to generate requests for three servers; we loaded 600,000 items and used two YCSB clients for six servers, and so on. For each system size, we varied the intensity of the request load by varying the number of threads in each YCSB client. For a small number of client threads, the request rate is low and thus the servers are under-utilized, while a large number of client threads can overload the servers. We started with four client threads, and doubled the thread count for each data point until we reached 128 threads.

Figures 9.1 and 9.2 show the results, averaged over three runs, with different curves for different numbers of servers. For each server count, as the request load increases, the throughput also increases up to a certain

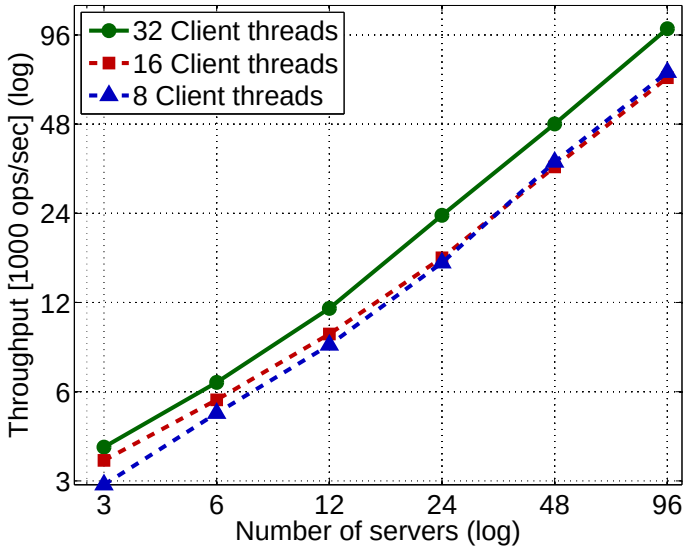

Figure 9.3. Scalability under a read-intensive (95% reads) workload.

value where it begins to plateau. After that, only the latency increases without any further increase in throughput. In this regime, the system is saturated and it cannot offer any more throughput, exhibiting operation queueing effects. When the system is underloaded – few client threads – we have low latency yet server resources are not fully utilized. As the request rate is increased by increasing the number of client threads, the latency and throughput increase up to a saturation threshold. For example, with three CATS servers, 32 YCSB client threads, and a read-intensive workload, the system saturates at approximately 4,000 operations/second with an average latency of eight milliseconds. Further increasing the request rate does not increase the throughput, while the latency keeps on increasing. The same behavior is exhibited under both workloads.

In summary, CATS delivers sub-millisecond operation latencies under light load, single-digit millisecond operation latencies at 50% load, and it sustains a throughput of 1,000 operations/second, per server, under read-intensive workloads. For update-intensive workloads the throughput is $1/3$ lower, which is expected since the message complexity for reads is half of that for updates, leading to a 2:3 cost ratio between the two workloads.

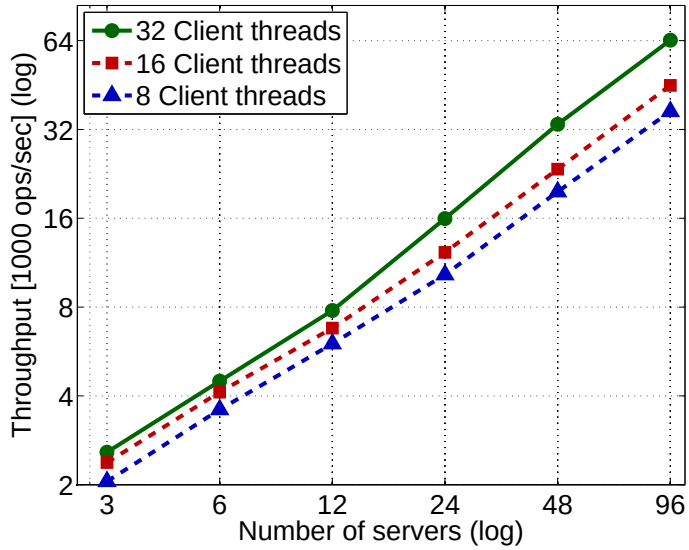

Figure 9.4. Scalability under an update-intensive (50% writes) workload.

9.3 Scalability

To evaluate the scalability of CATS, we increased the dataset size and the request rate, proportionally to the number of servers as before, i.e., by increasing the number of data items loaded initially, and the number of YCSB clients, respectively. Figures 9.3 and 9.4 show the throughput of the system as we vary the number of servers for each workload. CATS scales linearly with a slope of one! With a small number of servers, it is more likely that requests arrive directly at one of the replicas for the requested key, therefore the message complexity is lower. This reduced bandwidth usage explains the slightly higher throughput for three and six servers.

The reason for linear scaling is that CATS is completely decentralized and all nodes are symmetric. Linear scalability facilitates resource provisioning; the number of servers needed to store a certain amount of data and to handle a certain rate of requests, can be calculated easily when deploying CATS in a cloud environment, provided the load is balanced evenly across the servers. Such a decision can be made either by an administrator, or by a feedback control loop that monitors the rate of client requests.

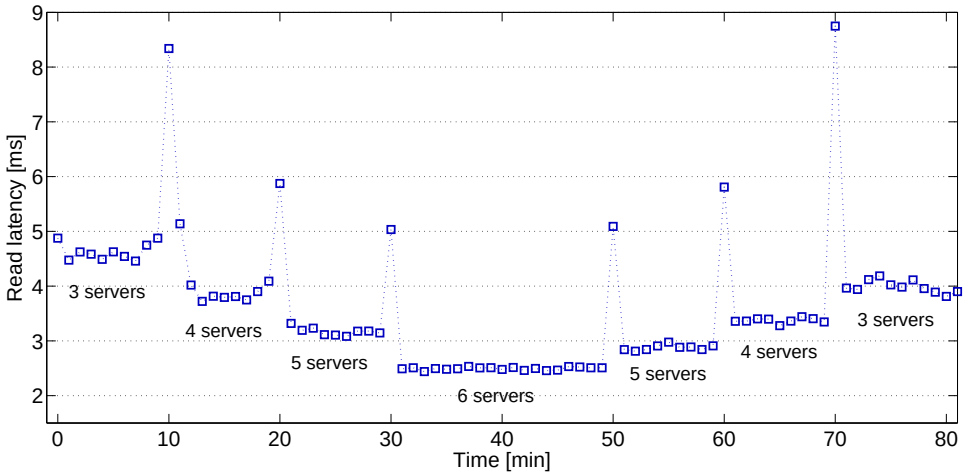

Figure 9.5. Elasticity under a read-only workload.

9.4 Elasticity

A highly desirable property for cloud computing systems is *elasticity*, the ability to add or remove servers while the system is running, in order to accommodate fluctuations in service demand. When a system is overloaded, and the operation latency is so high that it violates its service-level agreements (SLAs), performance can be improved by adding new servers. Similarly, when the load is very low, resource utilization can be improved by removing servers from the system without violating any of its SLAs.

A system with good elasticity should perform better as servers are added, perhaps operating at moderately reduced throughput and slightly higher latency for a brief period of time, while the system reconfigures to add or remove servers. The length of this period depends on the amount of data that needs to be transferred among the servers to complete the reconfiguration. A well-behaved system should still offer fairly low latency during reconfiguration to minimize its impact on serving client requests.

In this experiment, we evaluated the elasticity of CATS. We started the system with three servers, loaded 2.4 million 1 KB values, and injected a high operation request rate via the YCSB client. While the workload was running at a constant request rate, we added a new server every 10 minutes

until the server count doubled to six servers. After 20 minutes, we started to remove one server every 10 minutes until we were down to three servers again. We measured the average operation latency in one minute intervals throughout the experiment. The results, presented in Figure 9.5, show that CATS is able to reconfigure promptly, e.g., within a span of roughly one to two minutes. The duration of the reconfiguration depends mostly on the amount of data transferred and the bandwidth capacity of the network.

The average operation latency during reconfiguration does not exceed more than roughly double the average latency in steady state, i.e., before the reconfiguration is triggered. For example, with six servers in steady state, the system offers an average operation latency of approximately 2.5 ms, while during reconfiguration that latency grows to circa 5 ms.

Because CATS is linearly scalable, the latency approximately halves when the number of servers doubles from three to six: while during the first 10 minutes of the experiment, three servers offer an average latency of 5 ms, between minutes 30 and 50, six servers deliver a latency of indeed only 2.5 ms. As expected, an increase in latency occurs once nodes are removed after 50 minutes. As the CATS servers were running under load for more than one hour, the JVM had been constantly optimizing the hot code paths in the system. This explains the asymmetric latencies whereby instead of a completely mirrored graph, in the second half of the experiment we observe slightly better performance for the same system configuration.

9.5 Overhead of Consistent Quorums

Next, we evaluate the performance overhead of atomic consistency compared to eventual consistency. For a fair comparison, we implemented eventual consistency in CATS, enabled through a configuration parameter. Here, read and update operations are always performed in one phase, and read-impose is never performed. When a node n performs a read operation, it sends read requests to all replicas. Each replica replies with a timestamp and a value. After n receives replies from a majority of replicas, it returns the value with the highest timestamp as the result of the read operation. Similarly, when node n performs an update operation, it sends write requests to all replicas, using the current wall clock time as a timestamp.

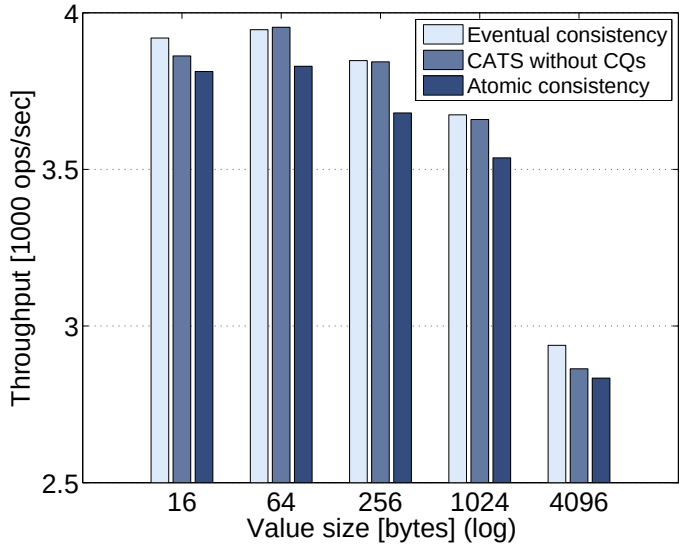

Figure 9.6. Overhead of atomic consistency and consistent quorums versus an implementation of eventual consistency, under a read-intensive workload.

Upon receiving a write request, a replica stores the value and timestamp only if the received timestamp is higher than the replica's local timestamp for that particular data item. The replica then sends an acknowledgment to the writer m. Node m considers the write operation complete upon receiving acknowledgments from a majority of the replicas.

We also measured the overhead of consistent quorums. For these measurements, we modified CATS such that nodes did not send replication group views in read and write messages. Removing the replication group view from messages reduces their size, and thus requires less bandwidth.

For these experiments, we varied the size of the stored data values, and we measured the throughput of a system with three servers. The measurements, averaged over five runs, are shown in Figures 9.6 and 9.7. The results show that as the value size increases, the throughput falls, meaning that the network becomes a bottleneck for larger value sizes. The same trend is observable in both workloads. As the value size increases, the cost of using consistent quorums becomes negligible. For instance, under both workloads, the throughput loss when using consistent quorums is less

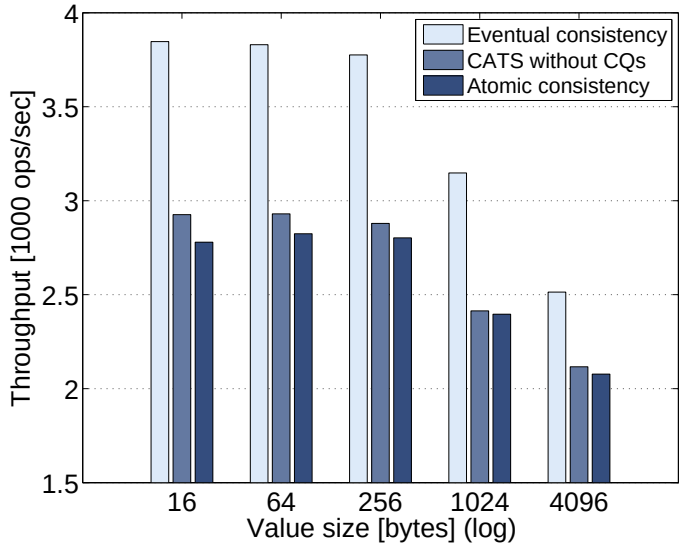

Figure 9.7. Overhead of atomic consistency and consistent quorums vs. an eventual consistency implementation, under an update-intensive workload.

than 5% for 256 B values, 4% for 1 KB values, and 1% for 4 KB values.

Figures 9.6 and 9.7 also show the cost of achieving atomic consistency by comparing the throughput of regular CATS with the throughput of our implementation of eventual consistency. The results show that the overhead of atomic consistency is negligible for a read-intensive workload and as high as 25% for an update-intensive workload. The reason for this difference between the two workloads is that for a read-intensive workload, read operations rarely need to perform the read-impose phase, since the number of concurrent writes to the same key is very low due to the large number of keys in the workload. For an update-intensive workload, due to many concurrent writes to the same key, read operations often require to impose the read value. Therefore, in comparison to an update-intensive workload, the overhead of achieving linearizability is very low – less than 5% loss in throughput for all value sizes – for a read-intensive workload. We believe that this is an important result. Applications that are read-intensive can opt for atomic consistency without a significant loss in performance, while avoiding the complexities of using eventual consistency.

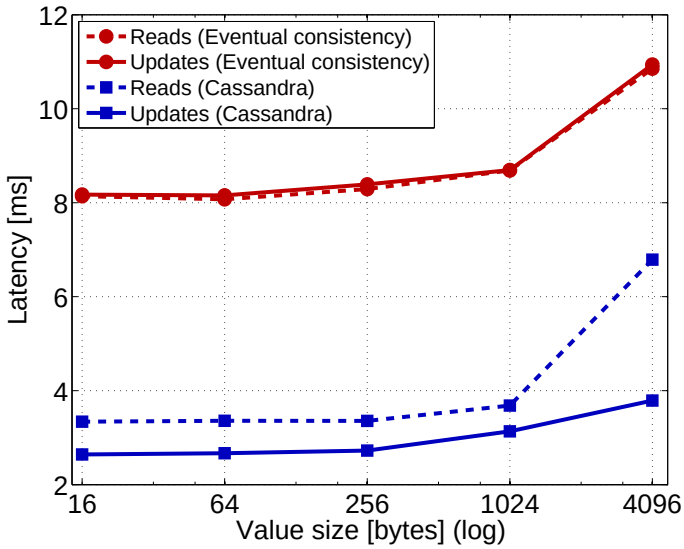

Figure 9.8. Latency comparison between Cassandra and an implementation of eventual consistency in CATS, under a read-intensive workload.

9.6 Comparison with Cassandra

Cassandra [125] and other distributed key-value stores [67, 30, 80] which use consistent hashing with successor-list replication have a very similar architecture to that of CATS. Since Cassandra was freely available, we compared the performance of CATS with that of Cassandra.

We should note that we are comparing our research system prototype with a system that leverages half a decade of implementation optimizations and fine tuning by a community of open-source contributors. Our goal is to give the reader an idea about the relative performance difference between the two systems. Extrapolating our previous evaluation of the overhead of atomic consistency using consistent quorums, this may give an insight into the cost of atomic consistency if implemented in Cassandra. We leave the actual implementation of consistent quorums in Cassandra to future work.

Both CATS and Cassandra are implemented in Java. We used Cassandra version 1.1.0, the latest version available at the time, and we used the `QUORUM` consistency level for a fair comparison with CATS. We chose the

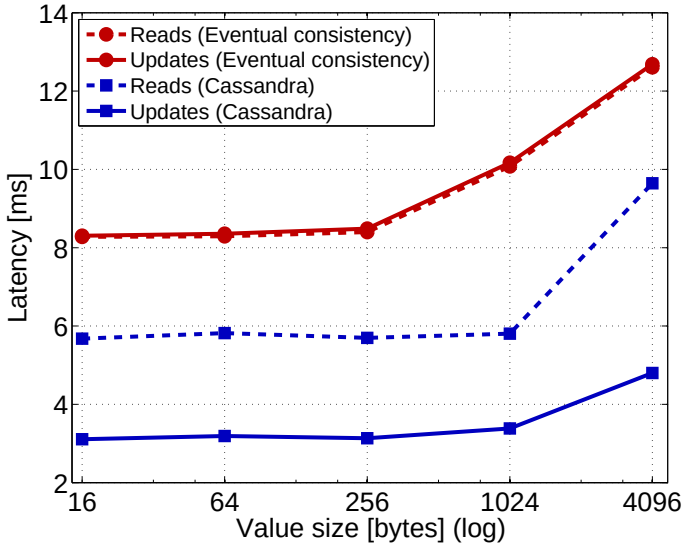

Figure 9.9. Latency comparison between Cassandra and an implementation of eventual consistency in CATS, under an update-intensive workload.

initial data size such that the working set would fit in main memory. Since CATS was storing data in main memory while Cassandra used disk, we set `commitlog_sync: periodic` in Cassandra to minimize the effects to disk activity on operation latencies and make for a fair comparison. Figures 9.8 and 9.9 show mean operation latencies, whereby each data point represents measurements averaged over five runs. Using the same workloads, we compared Cassandra and CATS with eventual consistency. The trend of higher latencies for large value sizes remains the same for both systems and workloads as the network starts to become a bottleneck. For CATS, read and update latencies are the same since both operations have the same message complexity and same-size messages. On the other hand, Cassandra updates are faster than reads, which was expected since in Cassandra updates are committed to an append-only log and require no disk reads or seeks, while read operations may need to consult multiple uncompacted SSTables[1] in search for the requested data. The results show that the operation latencies in CATS are approximately three times higher

[1] `http://wiki.apache.org/cassandra/ArchitectureOverview`

than in Cassandra, except for reads under an update-intensive workload, where SSTable compactions occur too seldom relatively to the high update rate, causing the need to consult multiple SSTable for each read operation and thus affecting Cassandra's performance.

Given our comparison between Cassandra and CATS with eventual consistency, as well as the relatively small decrease in throughput when providing atomic consistency – using consistent quorums and two-phase write operations – instead of only eventually consistent single-phase operations (see Section 9.5), we believe that an implementation of consistent quorums in Cassandra can provide linearizable consistency without a considerable drop in performance, e.g., less than 5% overhead for a read-intensive workload, and about 25% overhead for update-intensive workloads.

Chapter 10

CATS Discussion and Comparison to Related Work

In this chapter we discuss alternative consistency models that can be implemented in a simple manner on top of the foundation of scalable reconfigurable group membership provided by CATS. We also discuss possible efficient implementations of these models and we compare CATS with related work in the areas of scalable key-value stores and consistent meta-data storage systems.

CATS brings together the scalability and self-organization of DHTs with the linearizable consistency and partition tolerance of atomic registers.

10.1 Alternatives to Majority Quorums

For some applications majority quorums may be too strict. To accommodate specific read-intensive or update-intensive workloads, they might want flexible quorum sizes for *put* and *get* operations, like *read-any-update-all* or *read-all-update-any*, despite the fault-tolerance caveats entailed. Interestingly, our ABD-based two-phase algorithm, depends on majority quorums for

linearizability, however, by using more flexible yet overlapping quorums, the algorithm still satisfies *sequential consistency* [25], which is slightly weaker, but still a very useful level of consistency, as we discussed in Section 6.3. This means that system designers are free to decide the size of quorums for read and write operations to suit their workloads, as long as the read and write quorums overlap. For instance, in a stable environment, like a data center, motivated by the need to handle read-intensive workloads, a system designer may choose the size of write quorums to be larger than the size of read quorums, in order to enable lower read latencies at the expense of more costly and less fault-tolerant writes – meaning that write operations need to send more messages and wait for acknowledgements from more replicas, and thus they can tolerate fewer crashed replicas. Consider a read-intensive workload and a replication degree of three. The write quorum size can be chosen as three and the read quorum as one. Such a configuration makes writes more expensive and less fault-tolerant, yet the read latency reduces tremendously since only one node – any node in the replication group – is involved in the read operation.

On a related note, the idea of primary-backup replication could be applied onto the consistent replication groups of CATS, to enable efficient primary reads. For instance, the node with the lowest identifier in each group could be considered to be the primary for that group; thus enabling primary-based replication in CATS. With a primary-backup scheme there are two possible designs: lease-based and non-lease-based.

The lease-based design [133] assumes a timed-asynchronous model and relies on this assumption to guarantee that at all times, at most one node considers itself to be the primary. In this design, read operations can always be directly answered by the primary, without contacting other nodes since the unique primary must have seen the latest write. Write operations can be sequenced by the primary but cannot be acknowledged to clients before the primary commits them at a majority of replicas in order to avoid lost updates in case of primary failure.

In the non-lease-based design, all operations are directed at the primary and for both read and write operations, the primary must contact a majority of replicas before acknowledging the operation to the client. Because all operations involve a majority of nodes, there is no safety violation when more than one node considers itself to be a primary. This can be achieved if

a primary is elected by a majority of the cohorts who have knowledge of the last elected primary. One of the candidates will fail to be elected primary by a majority of the cohorts. When electing a new primary, a majority of cohorts will inform the new primary of the largest operation sequence number they have seen so far. Therefore, the newly elected primary can continue to sequence operations starting from the last operation committed at a majority of replicas.

10.2 Sequential Consistency at Scale

We found that for a slightly lower cost than that of providing linearizability [106], we could provide a slightly weaker but still very useful consistency guarantee, namely sequential consistency [128]. More concretely, for our linearizable operations, both reads and writes need two phases, while in the absence of concurrent updates, read operations only require one phase. A phase is a round-trip to k servers, waiting for a majority of acknowledgements, so a phase costs 2 message delays and $2k$ messages. Within one replication group, sequential consistency can be guaranteed with single-phase writes by maintaining Lamport logical clocks [127] at each process, and using the current Lamport clock as the write timestamp. This saves the initial phase of the write operation during which the latest register timestamp is consulted. In general, in contrast to linearizability, sequential consistency is not composable. Interestingly, using write operations based on Lamport timestamps, and maintaining the Lamport clocks across all nodes, also preserves sequential consistency across replication groups, therefore this scheme enables sequential consistency at large scale [142]. This appears to be a very interesting result which we plan to investigate in depth in future work.

10.3 Scalable Key-Value Stores

Distributed key-value stores, such as Cassandra [125] and Dynamo [67], employ principles from DHTs to build scalable and self-managing data stores. In contrast to CATS, these systems chose availability over atomic consistency, hence only providing eventual consistency. While eventual

consistency is sufficient for some applications, the complexities of merging divergent replicas can be non-trivial. We avoid the complexities entailed by eventual consistency while providing scalable storage for critical applications which need atomic consistency, guaranteeing it at the cost of a modest decrease in throughput. We showed in Chapter 9 that the overhead of atomic consistency is indeed very low for read-intensive workloads.

10.4 Reconfigurable Replication Systems

To handle dynamic networks, atomic registers were extended by protocols such as RAMBO [147], RAMBO II [90], RDS [56] and DynaStore [4] to be reconfigurable. Similarly, SMART [143] enabled reconfiguration in replicated state machines (RSMs). With consistent quorums we provide high-throughput *put* and *get* operations without paying the full cost of state machine replication which needs coordination for every operation. Moreover, our design does not depend on electing a single leader and the complexities that come with that [47]. While these systems can handle dynamism and provide atomic consistency, they are not scalable as they were not designed to partition the data across a large number of machines. The novelty of CATS is in extending the reconfiguration techniques contributed by these works, such that they can be used at large scale, in order to build a system that is completely decentralized and self-managing.

10.5 Consistent Meta-Data Stores

Datacenter systems providing distributed coordination and consistent meta-data storage services, such as Chubby [45] and ZooKeeper [110, 119], provide linearizability and crash-recovery, but are neither scalable, nor freely reconfigurable. The idea of consistent quorums applied to consistent hashing rings can be used to scale such meta-data stores to larger capacities.

Master-based key-value stores, such as Bigtable [51], HBase [102], and MongoDB [158], rely on a central server for coordination and data partitioning. Similarly, Spinnaker [188] uses Zookeeper [110]. Since these systems are centralized, their scalability is limited. In contrast, CATS is decentralized and all nodes are symmetric, allowing for unlimited scalability.

10.6 Scalable and Consistent Key-Value Stores

Similar to CATS, Scatter [91] is a scalable and consistent key-value store. Scatter employs an extra subsystem and policies to decide when to reconfigure replication groups. While this makes Scatter flexible, it also requires a distributed transaction [229, 229] across three adjacent replication groups for the split and merge reconfiguration operations to succeed. In contrast, CATS has a simpler and more efficient reconfiguration protocol – both in the number of messages and message delays – which does not require distributed transactions. In CATS, each reconfiguration operation only operates on the replication group that is being reconfigured. Therefore, the period of unavailability to serve operations is much shorter, almost non-existent in CATS, compared to Scatter. The unavailability of Scatter's implementation precludes a detailed comparison, e.g., in terms of elasticity and data unavailability during reconfiguration. We focus on consistent-hashing at the node level, which makes our approach directly implementable in existing key-value stores like Cassandra [125].

Perhaps the main distinguishing advantage of CATS over Scatter is CATS' ability to handle network partitions and mergers, an aspect largely ignored in Scatter. Once network partitions cease, CATS merges partitioned subsystems into a single overlay, while Scatter will continue to operate as separate overlays. Where Scatter provides scalability and consistency, CATS provides scalability, consistency, and partition tolerance.

10.7 Related Work on Consistency

An orthogonal approach to atomic consistency is to explore the trade-offs between consistency and performance [26]. For instance, PNUTS [59] introduces time-line consistency, whereas COPS [142] provides causal consistency at scale. These systems provide consistency guarantees weaker than linearizability, yet stronger guarantees than eventual consistency. While such systems perform well, the semantics of the consistency models they offer restricts the class of applications that can use them.

In the CALM approach, programming language support is used to automatically separate distributed programs into monotonic and non-monotonic parts [11]. Logically monotonic distributed code – for which the order or

the contents of the input can never cause a need for some earlier output to be "revoked" once it has been generated – is already eventually consistent [215] without requiring any coordination protocols, which are only needed to protect regions of non-monotonic code.

Costly coordination protocols can also be avoided when dealing with replicated data storage, if data types and operations are designed to commute. Operation commutativity is explored by Commutative Replicated Data Types (CRDTs), which are data types whose operations commute when they are concurrent. If all concurrent update operations to some data object commute, and all of its replicas execute all updates in causal order [127], replicas of a CRDT eventually converge without requiring any complex concurrency control.

10.8 Fault-Tolerant Replicated Data Management

Abbadi et al. [75] proposed a fault-tolerant protocol for replicated data management. Their solution is similar to CATS with respect to quorum-based operations and consensus-based replication group reconfigurations. In contrast to their solution, CATS relies on consistent hashing, which enables it to be self-managing and self-organizing under churn. Consistent hashing partitions the keys in a balanced manner, and the notion of responsibility in terms of which nodes are responsible for storing which key ranges is well-defined. Thus, the reconfigurations required when nodes join and fail is dictated by consistent hashing. Furthermore, owing to the routing mechanisms employed by CATS, any node can find any key in a few hops even for very large network sizes.

Chapter 11

Conclusions

We are witnessing a boom in distributed services and applications. Many companies independently develop complex distributed systems from the ground up. The current situation is comparable to the times when companies were independently developing different networking architectures before the ISO/OSI model [234] came along. We believe that industry would benefit tremendously from the availability of a systematic approach to building, testing, and debugging distributed systems.

The goal of this thesis has been to devise a programming model that would streamline the development of dynamic, evolving, and adaptive distributed systems. In the light of our experience with Kompics, as well as our qualitative and quantitative evaluations, we firmly believe that using a reactive, concurrent, and hierarchically nested component model, with an explicit software architecture and explicit component dependencies, will contribute to this goal in at least three different ways. First, the challenge imposed by the complexity of a modern distributed system is tackled by providing mechanisms for building scalable and reusable abstractions. Second, by employing message-passing concurrency, our execution model allows for multi-core scalable component scheduling and compositional concurrency. Third, the testing, evaluation, and deployment of distributed

systems is streamlined due to the ability to reuse component abstractions across different executions environments. We are able to execute a complete distributed system in deterministic simulation mode, for purposes of large-scale behavior evaluation, as well as for protocol correctness testing and for debugging. We can subject the same system implementation to stress tests by executing it either locally on one machine, or in a controlled cluster environment. Finally, the same system implementation can be deployed and executed in a production environment.

The experience that we gained from using Kompics to design, program, compose, test, debug, and evaluate distributed systems, on the one hand, and the ease with which we and others were able to develop non-trivial systems, by leveraging lower-level abstractions and encapsulating them into first-class higher level abstractions, on the other hand, leads us to conclude that hierarchically nested, message-passing, reactive components constitute a promising programming model to streamline the development cycle for complex and reconfigurable distributed systems.

We have successfully used the Kompics component model as a teaching framework, for more than five years, in two Master's level courses on distributed systems given at KTH; a course on advanced distributed algorithms and abstractions and a course on large-scale and dynamic peer-to-peer systems. Kompics enabled students both to compose various distributed abstractions and to experiment with large-scale overlays and content-distribution networks in simulation and real execution. The students were able both to deliver running implementations of complex distributed systems, and to gain insights into the dynamics of those systems. We believe that making distributed systems easier to program and experiment with, will significantly improve the education process in this field and will lead to better equipped practitioners.

The practicality of Kompics has been confirmed by its use for rapid prototyping, development, and evaluation of a broad collection of distributed systems, both within and outside of our research group. Systems built with Kompics include a peer-to-peer *video-on-demand* system [37], a secure and fault-tolerant distributed *storage* system [111], NAT-aware *peer-sampling* protocols [73, 172], peer-to-peer *live media streaming* systems [170, 174, 171, 173, 176], locality-aware scalable *publish-subscribe* systems [187], scalable *NAT-traversal* protocols [164], distributed hash-table

replication schemes [200], gossip protocols for *distribution estimation* [175], an *elasticity controller* simulator [162, 161], studies of multi-consistency-model *key-value stores* [7, 41], mechanisms for *robust self-management* [6, 22], and a *reliable UDP* transport mechanism [157]. The ample diversity of these applications is a testament to the usefulness of Kompics.

As a comprehensive case study of using Kompics to develop and test distributed systems, we presented CATS, a scalable and consistent key-value store which trades off service availability for guarantees of atomic data consistency and tolerance to network partitions.

We have shown that it is non-trivial to achieve linearizable consistency in dynamic, scalable, and self-organizing key-value stores which distribute and replicate data according to the principle of consistent hashing. We introduced consistent quorums as a solution to this problem for partially synchronous network environments prone to message loss, network partitioning, and inaccurate failure suspicions. We argued that consistent quorums can be used to adapt any static quorum-based distributed algorithm to function correctly in dynamic replication groups automatically reconfigured by consistent hashing, potentially at large scale, and we presented adaptations of Paxos and ABD as examples.

In essence, we provide a reconfigurable replicated state machine for the membership view of each replication group, which is then seamlessly leveraged by consistent quorums to simply adapt existing quorum-based algorithms to operate at large scales in dynamic groups. This novel approach of decoupling reconfiguration from the *put* and *get* operations allows for more operation parallelism and higher throughput than existing approaches where linearizability is guaranteed by state machine replication which is inherently sequential.

We described the design, implementation, testing, and evaluation of CATS, which leverages consistent quorums to provide linearizable consistency and partition tolerance. CATS is self-managing, elastic, and it exhibits unlimited linear scalability, all of which are key properties for modern cloud computing storage middleware. Our evaluation shows that it is feasible to provide linearizable consistency for those applications that do indeed need it. The throughput overhead of atomic consistency over an eventual consistency implementation, was less than 25% for write-intensive workloads and less than 5% for read-intensive workloads. Our system im-

plementation can deliver practical levels of performance, comparable with those of similar but heavily-optimized industrial systems like Cassandra. This suggests that if implemented in Cassandra, consistent quorums can deliver atomic consistency with acceptable performance overhead.

CATS delivers sub-millisecond operation latencies under light load, single-digit millisecond operation latencies at 50% load, and it sustains a throughput of one thousand operations per second, per server, while scaling linearly to hundreds of servers. This level of performance is competitive with that of systems with a similar architecture but which provide only weaker consistency guarantees [67, 125, 30, 80].

11.1 Kompics Limitations and Lessons Learnt

Without adequate programming language support, one current limitation of the Kompics message-passing mechanism is related to the message copying vs. sharing trade-off. Currently, we have to choose between the overhead of copying messages from the source to the destination component, like in Erlang, or the efficiency of passing messages by reference at the cost of potential concurrent access to the message from the source and the destination components. A Kompics implementation in a programming language providing single-reference types would alleviate this problem, as was done in Kilim [206] and Singularity [78].

A second limitation stemming from the lack of adequate language support regards the expression of event handlers that invoke services or RPCs, whereby a request is sent and the execution of the event handler can continue only after receiving a response. Currently, such a handler must be split in two different handlers: one handler containing the code before the service invocation, and another handler containing the continuation code executed after receiving the service response. The reference Kompics implementation is written in Java which does not provide continuations [190]. Support for continuations in the host language would enable Kompics programmers to write event handlers containing multiple remote service invocations. Once a service request is sent, the component state, including the local state of the active handler, is saved in a continuation. When the component receives the service response, the state saved in the continuation

is restored, and the active handler may resume execution where it left off before the service invocation. The Scala programming language supports continuations but that is currently not leveraged in the Kompics Scala port.

Another language-related drawback of our Kompics reference implementation was caused by the verbosity of the Java programming language. Programmers needed to write a lot of scaffolding code, even when defining relatively simple concepts like events and ports. The result was that in some cases, potential Kompics users were put off by this aspect and decided not to use it. This means that user adoption was not as good as it could have been. To alleviate this problem, Kompics was ported to Python and Scala which yield much more succinct programs. Brevity in these languages is enabled by dynamic typing in Python and type inference, case classes, DSL support, pattern matching, and lambda expressions in Scala.

We found that some aspects of explicitly managing the software architecture can prove tricky. For example, it happens that programmers forget to subscribe certain event handlers to incoming ports or forget to initialize newly created components. To this end, an architectural static checker was developed and packaged into an Eclipse IDE plug-in, which triggers warnings for common pitfalls and also allows users to visually inspect the Kompics component architecture.

While evaluating the performance of CATS we noticed a plateau in throughput even though neither the CPU utilization nor the network utilization was at a maximum. We found that although our machines had many available cores, not all processing cores were used, because only a few CATS components were exercised under the benchmark workload. This was an artifact of our component execution model, in particular, of the rule that event handlers are mutually exclusive within one component instance. This prevents a component from being executed in parallel by multiple workers, even when the component has a lot of work and there are free workers available. The motivation for this design choice was to simplify programming components by freeing the user from implementing critical sections by hand. While this is generally very useful, in some cases it proves to be a limitation. In this particular case, the default execution model provides too coarse synchronization, and allowing the user to explicitly program component state synchronization may enable increased performance. The lesson here is that while our execution model provides

a good default, we need to extend Kompics with a special type of *concurrency unsafe* component. Unsafe components would allow the component programmer to manage state synchronization explicitly and would enable better throughput through increased multi-core utilization. This would be particularly appropriate for stateless components which need little or no state synchronization.

One other lesson we learnt while using Kompics for building various distributed systems, is that dynamic reconfiguration is not always needed. In our reference Kompics implementation, support for dynamic reconfiguration is baked into the core of the run-time system. Some aspects of dynamic reconfiguration support are on the critical path of component execution, and systems which do not need it are currently paying for it anyway. It would be useful to add a configuration flag to the Kompics run-time system, to enable users to turn off dynamic reconfiguration when they don't need it, and gain improved performance in exchange.

Notwithstanding these limitations, our overall experience confirms the hypothesis that Kompics is an effective approach to modularize distributed systems, compose, develop, test, and debug them.

11.2 CATS Limitations and Lessons Learnt

Linearizable consistency is the strongest form of consistency for a distributed shared-memory read/write register. In the context of a scalable key-value store, we deal with many such registers, and applications are free to use multiple registers to store their data. A useful property of linearizability is that it is composable, that is, the interleaving of operations on a set of linearizable registers is itself linearizable [106]. In other words, composing a set of linearizable distributed shared-memory cells into a global memory, yields a linearizable memory. This is very practical, however, some applications need consistency guarantees for operations that access multiple registers and need to take effect atomically. These are commonly referred to as transactions [229, 36]. Even though transaction processing would be easy to implement on top of the consistent, scalable, and reconfigurable node membership foundation provided by CATS, together with consistent quorums, currently transactions are not available.

An abstraction related to transaction processing is that of state machine replication [198]. In this abstraction, the set of all shared-memory registers are interpreted as the state of a state machine. Like transactions, operations on a replicated state machine may access multiple registers. In general, while transactions on distinct sets of registers may execute in parallel without interfering with each other, a replicated state machine executes operations sequentially. Again, it is relatively easy to implement a reconfigurable replicated state machine abstraction [132, 143] on top of CATS's underpinnings of reconfigurable group membership and consistent quorums. The downside of this approach is that each replication group would form its own replicated state machine, independent of other state machines. This means that in contrast to linearizable registers, replicated state machines are not composable. Nevertheless, within a replicated state machine, each individual register could support stronger types of operations like read-modify-write [79]. Currently, read-modify-write operations like increment or compare-and-swap are not available in CATS.

While experimenting with the CATS system, we learnt that successor-list replication has some negative consequences of practical concern, regarding efficiency and complexity. Since a given node is part of multiple, adjacent replication groups, it becomes very complicated to implement different replication policies, like replica placement, for adjacent ranges of registers. This also causes load balancing to become more complex and inefficient. Furthermore, supporting replication groups with different replication degrees is unnecessarily complex. With CATS our initial focus was on guaranteeing linearizable consistency and partition tolerance, hence we did not pay much attention to the details of the replication scheme. While indeed this is an orthogonal aspect to our contribution, the lesson we learnt is that when building a practical system, the choice of replication scheme is very important.

While examining the message complexity of the CATS protocols for read and write operations on a single register, we learnt that for a slightly lower cost than that of providing linearizability [106], we could provide a slightly weaker but still very useful consistency guarantee, namely sequential consistency [128]. More concretely, for our linearizable operations, both reads and writes need two phases, while in the absence of concurrent updates, reads only require one phase. A phase is a round-trip to k servers,

waiting for a majority of acknowledgements, so a phase costs 2 message delays and $2k$ messages. Within one replication group, sequential consistency can be guaranteed with single-phase writes by maintaining Lamport logical clocks [127] at each process, and using the current Lamport clock as the write timestamp. This saves the initial phase of the write operation during which the latest register timestamp is consulted. In general, in contrast to linearizability, sequential consistency is not composable. Noteworthily, we found that using write operations based on Lamport timestamps, and maintaining the Lamport clocks across all nodes, also preserves sequential consistency across replication groups, therefore this scheme enables sequential consistency at large scale, a very interesting lesson.

11.3 Future Work

Addressing some of the limitations identified in the previous two sections, as well as leveraging some of the lessons we learnt, constitute the subject of future work. We discuss additional directions of future work below.

For Kompics, we need to design and evaluate mechanisms for component *deployment* and *dependency management*. The nodes of a complex distributed system need not evolve homogeneously, thus exhibiting diverse component configurations. In such a scenario, mechanisms are required to resolve and deploy component dependencies upon the installation of new components. Such mechanisms may need to be Turing complete. We plan to investigate the appropriate programming constructs to support them.

A second direction of future work on Kompics is the design and evaluation of mechanisms to *augment the power of the component model with plug-in components*. Take for example a component that implements an atomic commit protocol [93]. Once this component is installed on a group of nodes of the system, it could be leveraged for the *transactional reconfiguration* of the software architecture at those nodes. As a result, the reconfiguration would take effect only if it were successful at all nodes involved.

We plan to evaluate case studies for *distribution transparency* [42, 212, 62]. Kompics components are not distributed. A component is local to one node and it communicates with remote counterparts by sending messages using lower-level abstractions, e.g., network links, broadcast, etc.; therefore

distribution is explicit. We would like to investigate case studies where the transparent distribution of some of the model's entities would considerably simplify the system's implementation.

Another direction for future work on Kompics regards the design and evaluation of efficient *multi-core component schedulers* and mechanisms for component prioritization and resource allocation. We have already explored some component schedulers based on work-stealing [40] and work-balancing [105], but we would like to further investigate schedulers that adapt their behavior according to the system load.

We would like to extend our current simulation environment with a *stochastic model checker* to improve the testing and debugging of Kompics-based systems. A particular implementation of a distributed abstraction is replaced with an implementation that generates random admissible executions for the replaced abstraction. This enables the stochastic verification of other overlying abstractions, which become subject to various legal behaviors from the services they use. Verification of input/output event sequences through a port would constitute another interesting direction for future work on supporting distributed protocol verification in Kompics.

Finally, a number of optimizations could be implemented in the Kompics run-time system. The publish-subscribe mechanism for component communication could be optimized for event brokering close to the publication site, therefore avoiding the delivery of events through chains of channels and ports when there are no subscriptions at the destination components. For a given protocol composition and a particular workload, requests arriving at a server tend to trigger recurring sequences of event handlers. An event handled by a component triggers another event which is handled by another component, and so on, in a pattern, or event path, which is repeated for every request arriving at the server. To cater for this typical scenario, a form of just-in-time compilation could be employed within the Kompics run-time system, whereby hot chains of event handlers could be fused together and the publish-subscribe message-passing mechanism would be largely sidestepped. These dynamic optimizations offer a great potential for performance improvement.

In CATS, consistent quorums provide a consistent view of dynamic replication groups. Besides atomic read/write registers, such consistent views can be leveraged to implement replicated objects with stronger semantics.

For example, conditional update operations [79] such as conditional multi-writes [5], compare-and-swap [103], or increment, could be supported in a scalable setting. In the same vein, consistent quorums could be leveraged to implement reconfigurable replicated state machines [132, 143] or distributed multi-item transactions [61, 64]. As a system, CATS can be extended to support improved load-balancing, column-oriented APIs, as well as data indexing and search capabilities.

Other interesting directions for future work on CATS, are providing sequential consistency at large scale, and implementing a transactional key-value store supporting multiple levels of transaction isolation [221] and geographic replication [205].

11.4 Final Remarks

The contributions of this thesis are a programming model and protocols for reconfigurable distributed systems. We explored the theme of dynamic reconfiguration along two different axes. On the one hand, we introduced the Kompics programming model which facilitates the construction of distributed systems that support the dynamic reconfiguration of their software architecture. On the other hand, we proposed the technique of consistent quorums to decouple the dynamic reconfiguration of replication groups from the implementation of a distributed atomic read/write register, a fault-tolerant shared memory abstraction.

The two contributions came together in the design, implementation, testing, and evaluation of CATS; a consistent, network-partition-tolerant, scalable, elastic, and self-organizing key-value data storage system. With CATS we demonstrated that consistent quorums admit system designs with such salient properties, as well as efficient system implementations providing linearizable consistency with low latency and modest overhead.

Concomitantly, we used CATS to highlight the architectural patterns and abstractions facilitated by the Kompics component model, as well as to illustrate the Kompics methodology of using the same code base for both production deployment and whole-system repeatable simulation, which enables testing and debugging, reduces the potential for errors, increases confidence, and streamlines the development of distributed systems.

Appendix A

Kompics Abstract Syntax

This appendinx presents an interesting subset of the grammar corresponding to the Kompics abstract syntax.

⟨EventSpec⟩ ::= '**event**' ⟨EventType⟩ ['**extends**' ⟨EventType⟩] '**{**'
⟨Attribute⟩*
'**}**'

⟨Attribute⟩ ::= ⟨Type⟩ ⟨AttributeRef⟩
| ⟨EventType⟩ ⟨AttributeRef⟩

⟨PortSpec⟩ ::= '**port**' ⟨PortType⟩ '**{**'
⟨Request⟩*
⟨Indication⟩*
'**}**'

⟨Request⟩ ::= '**request**' ⟨EventType⟩
| '**negative**' ⟨EventType⟩

⟨Indication⟩ ::= '**indication**' ⟨EventType⟩
| '**positive**' ⟨EventType⟩

⟨ComponentSpec⟩ ::= '**component**' ⟨ComponentType⟩ '**{**'
⟨Port⟩*
⟨Component⟩*
⟨Constructor⟩
⟨StateVariable⟩*
⟨EventHandler⟩*
'**}**'

⟨Port⟩ ::= '**provides**' ⟨PortType⟩ ⟨PortRef⟩
| '**requires**' ⟨PortType⟩ ⟨PortRef⟩

⟨Component⟩ ::= '**component**' ⟨ComponentRef⟩ '**=**'
⟨ComponentType⟩ '**(**' ⟨ArgVal⟩* '**)**'

⟨Constructor⟩ ::= '**constructor**' '**(**' ⟨Parameter⟩* '**)**' '**{**'
⟨Statement⟩*
'**}**'

⟨Parameter⟩ ::= ⟨Type⟩ ⟨ParameterRef⟩

⟨StateVariable⟩ ::= '**var**' ⟨Type⟩ ⟨VarRef⟩

⟨EventHandler⟩ ::= '**handler**' ⟨HandlerRef⟩ '**(**' ⟨EventType⟩ ⟨Ref⟩ '**)**' '**{**'
⟨Statement⟩*
'**}**'

⟨Statement⟩ ::= ⟨LanguageStatement⟩
| ⟨Subscribe⟩
| ⟨Unsubscribe⟩
| ⟨Connect⟩
| ⟨Trigger⟩
| ⟨Expect⟩
| ⟨Create⟩
| ⟨Destroy⟩
| ⟨Hold⟩
| ⟨Resume⟩
| ⟨Unplug⟩
| ⟨Plug⟩

⟨Subscribe⟩ ::= '**subscribe**' ⟨HandlerRef⟩ '**to**' ⟨PortRef⟩

⟨Unsubscribe⟩ ::= '**unsubscribe**' ⟨HandlerRef⟩ '**from**' ⟨PortRef⟩

⟨Connect⟩ ::= ⟨ChannelRef⟩ '**=**' '**connect**' ⟨PortRef⟩ '**to**' ⟨PortRef⟩
['**filterby**' ⟨EventFilter⟩]

⟨EventFilter⟩ ::= '**positive**' ⟨Condition⟩ ['**and**' ⟨EventFilter⟩]
| '**negative**' ⟨Condition⟩ ['**and**' ⟨EventFilter⟩]

⟨Condition⟩ ::= ⟨EventType⟩ '**.**' ⟨AttributeRef⟩ '**=**' ⟨Value⟩

⟨Trigger⟩ ::= '**trigger**' ⟨EventType⟩ '**(**'
[⟨AttributeRef⟩ '**=**' ⟨ArgVal⟩]*
'**)**' '**on**' ⟨PortRef⟩

⟨Expect⟩ ::= '**expect**' ⟨ExpectFilter⟩ ['**or**' ⟨ExpectFilter⟩]*

⟨ExpectFilter⟩ ::= ⟨EventType⟩
['**(**' ⟨BooleanConditionOnStateVarsAndEvent⟩ '**)**']
['**on**' ⟨PortRef⟩]

⟨Create⟩ ::= ⟨ComponentRef⟩ '**=**' '**create**' ⟨ComponentType⟩ '**(**'
⟨ArgVal⟩*
'**)**'

⟨Destroy⟩ ::= '**destroy**' ⟨ComponentRef⟩
| '**destroy**' ⟨ChannelRef⟩

⟨Hold⟩ ::= '**hold**' ⟨ChannelRef⟩

⟨Resume⟩ ::= '**resume**' ⟨ChannelRef⟩

⟨Unplug⟩ ::= '**unplug**' ⟨ChannelRef⟩ '**from**' ⟨PortRef⟩

⟨Plug⟩ ::= '**plug**' ⟨ChannelRef⟩ '**to**' ⟨PortRef⟩

Appendix B

Kompics Operational Semantics

This appendix presents the operational semantics [182] for Kompics. We use an approach to describing the Kompics operational semantics similar the one taken by Van Roy and Haridi in describing the semantics of the Oz programming language [193, ch.13]. The configuration of a Kompics system is represented in an abstract store, containing predicate assertions about a component's state and the architectural relations among components.

The system bootstraps by creating a Main component from a given component specification and activating it. When a component is created, it executes a constructor procedure, which is a sequence of statements. A statement can be any sequential statement in the underlying programming language, or a statement of the Kompics model. For simplicity of presentation, and without loss of generality, we consider that the statements of the underlying language consist only of assignment and sequential composition even though consitional statements and loops are possible. We present the kernel Kompics statements in Figure B.1 and we describe their semantics in the remainder of the appendix.

S	::=	`skip`	*empty statement*
	|	`var` $v : T$	*variable introduction*
	|	$v :=$ *val*	*assignment*
	|	$S_1; S_2$	*sequential composition*
	|	`create` C c	*component creation*
	|	`provide` P p	*provided port creation*
	|	`require` P p	*required port creation*
	|	`subscribe` h `to` p	*handler subscription*
	|	`unsubscribe` h `from` p	*handler unsubscription*
	|	$x :=$ `connect` p `to` q `filterby` F_x	*channel creation*
	|	`trigger` e `on` p	*event triggering*
	|	`expect` F_e	*expectation for next event*
	|	`start` c	*component activation*
	|	`stop` c	*component passivation*
	|	`destroy` c	*component destruction*
	|	`hold` x	*channel passivation*
	|	`resume` x	*channel activation*
	|	`unplug` x `from` p	*channel disconnection*
	|	`plug` x `to` p	*channel connection*

Figure B.1. Kompics kernel language. `skip` is a no-op statement used for convenience in expressing reduction rules. S_1 and S_2 denote statements in the implementation programming language, v denotes a state variable, T denotes a data type, C denotes a component type, P denotes a port type, c denotes a subcomponent of the current component, p and q denote ports, h denotes an event handler, x and y denote channels, F_x denotes a channel filter, e denotes an event, and F_e denotes a pattern of expected events.

The system advances by successive reduction steps. A reduction rule of the form

$$\frac{\mathcal{C} \parallel \mathcal{C}'}{\sigma \parallel \sigma'} \text{ if } C$$

states that the computation makes a transition from a multiset of components $\mathcal{C}$ connected to a store σ, to a multiset of components $\mathcal{C}'$ connected to a store σ', if the boolean condition C holds. A store represents a conjunction of primitive assertions. A primitive assertion is a predicate of the form `pred`(...), which qualifies or relates model entities.

The following rule expresses *concurrency:*

$$\frac{\mathcal{C} \uplus \mathcal{D} \;\|\; \mathcal{C}' \uplus \mathcal{D}}{\sigma \;\|\; \sigma'} \text{ if } \frac{\mathcal{C} \;\|\; \mathcal{C}'}{\sigma \;\|\; \sigma'}$$

A subset of components can execute without affecting or depending on the other components in the system.

A components runs by executing its event handlers, sequentially, in response to received events. An event handler is a sequence of statements ending with the special statement `done`, which we use in the semantics as a marker for the termination of handler execution. We extend the reduction rule notation to allow the reduction of statements in addition to multisets of components. A statement S is reduced in the context of the component executing the statement, κ, which we denote by $\kappa\langle S \rangle$.

Sequential composition:

$$\frac{\kappa\langle S_1; S_2 \rangle \;\|\; \kappa\langle S_1'; S_2 \rangle}{\sigma \;\|\; \sigma'} \text{ if } \frac{\kappa\langle S_1 \rangle \;\|\; \kappa\langle S_1' \rangle}{\sigma \;\|\; \sigma'}$$

The reduction of a sequence of statements $S_1; S_2$, replaces the topmost statement S_1 with its reduction S_1'.

Empty statement:

$$\frac{\kappa\langle \texttt{skip}; T \rangle \;\|\; \kappa\langle T \rangle}{\sigma \;\|\; \sigma}$$

The empty statement `skip` is removed from any sequence of statements.

Assignment:

$$\frac{\kappa\langle v := val \rangle \;\|\; \kappa\langle \texttt{skip} \rangle}{\sigma \;\|\; \sigma \wedge \kappa(v) = val} \text{ if } \sigma \models v \in \mathcal{V}_\kappa \wedge type(v) \subseteq type(val)$$

If v is one of κ's state variables and its type is assignable from the type of the value val, then the assignment statement reduces to the empty statement and the store now records that the value of v in κ is now val.

Provide:

$$\frac{\kappa\langle \texttt{provide } P\ p \rangle \;\|\; \kappa\langle \texttt{skip} \rangle}{\sigma \;\|\; \sigma \wedge \sigma'} \text{ if } \sigma \models p^- \notin \mathcal{P}_\kappa \wedge P = spec(\pi)$$

where $\sigma' \equiv p^- \in \mathcal{P}_\kappa \wedge p^+ \in \mathcal{P}_{\kappa'} \wedge \kappa \in \mathcal{C}_{\kappa'} \wedge p \in \pi$. Port p of type π, defined in port specification P, is added to κ. This makes the pole p^- visible in κ and the pole p^+ visible in κ's parent.

Require:

$$\frac{\kappa\langle \texttt{require } P\ p\rangle}{\sigma} \ \Big\| \ \frac{\kappa\langle \texttt{skip}\rangle}{\sigma \wedge \sigma'} \quad \text{if } \sigma \models p^+ \notin \mathcal{P}_\kappa \wedge P = spec(\pi)$$

where $\sigma' \equiv p^+ \in \mathcal{P}_\kappa \wedge p^- \in \mathcal{P}_{\kappa'} \wedge \kappa \in \mathcal{C}_\kappa \wedge p \in \pi$. This rule is similar to the `provide` rule above. When port p is required, the pole p^+ is visible in κ and the pole p^- visible in κ's parent.

Subscribe:

$$\frac{\kappa\langle \texttt{subscribe } h \texttt{ to } p\rangle}{\sigma} \ \Big\| \ \frac{\kappa\langle \texttt{skip}\rangle}{\sigma \wedge sub(h(\varepsilon), p^{dir}) \in \mathcal{S}_\kappa} \quad \text{if}$$
$$\sigma \models h(\varepsilon) \in \mathcal{H}_\kappa \wedge p^{dir} \in \mathcal{P}_\kappa \wedge \varepsilon' \triangleright \pi^{dir} \wedge p \in \pi \wedge \varepsilon \subseteq \varepsilon' \wedge dir \in \{+, -\}$$

If h is a handler of κ that handles events of type ε and p is a port visible in κ, then h can be subscribed to p if ε', some supertype of ε, flows through p towards the pole visible in κ. The subscription of h to p is recorded in the store as a member of $\mathcal{S}_\kappa$, the set of all κ's subscriptions.

Unsubscribe:

$$\frac{\kappa\langle \texttt{unsubscribe } h \texttt{ from } p\rangle}{\sigma \wedge sub(h(\varepsilon), p^{dir}) \in \mathcal{S}_\kappa} \ \Big\| \ \frac{\kappa\langle \texttt{skip}\rangle}{\sigma} \quad \text{if } \sigma \models p^{dir} \in \mathcal{P}_\kappa$$

where $dir \in \{+, -\}$. If there exists a subscription of handler h to port p, in κ, it is removed.

Create:

$$\frac{\kappa\langle \texttt{create } C\ c\rangle}{\sigma} \ \Big\| \ \frac{\{\kappa\langle \texttt{skip}\rangle\} \uplus \{c\langle T\rangle\}}{\sigma \wedge \sigma'} \quad \text{if } \sigma \models c \notin \mathcal{C}_\kappa$$

where C is a component specification, T is the statement representing C's constructor, and $\sigma' \equiv c \in \mathcal{C}_\kappa \wedge \mathcal{V}_c = vars(C) \wedge \mathcal{H}_c = handlers(C) \wedge \mathcal{C}_c = \emptyset \wedge \mathcal{P}_c = \emptyset \wedge \mathcal{S}_c = \emptyset \wedge \mathcal{X}_c = \emptyset \wedge \neg active(c)$. Component c is created as a child of κ, from the specification C. Component c starts executing its constructor but will not execute any event handlers until it is started.

Connect:

$$\frac{\kappa\langle x = \texttt{connect}\ p\ \texttt{to}\ q\ \texttt{filterby}\ F_\pi\rangle \;\Big\|\; \kappa\langle\texttt{skip}\rangle}{\sigma \;\Big\|\; \sigma \wedge \sigma'} \text{ if}$$
$$\sigma \models x \notin \mathcal{X}_\kappa \wedge p \in \pi \wedge q \in \pi' \wedge \pi = \pi' \wedge p^{dir}, q^{\overline{dir}} \in \mathcal{P}_\kappa$$

where $\sigma' \equiv x(p,q) \in \mathcal{X}_\kappa \wedge active(x) \wedge F_\pi \in x$ and

$$dir \in \{+,-\} \wedge \overline{dir} = \begin{cases} - & \text{if } dir = + \\ + & \text{if } dir = - \end{cases}$$

A channel, x, is created to connect port p and q, both visible within κ. Ports p and q have to be of the same type and different polarities. The channel filter F_π is associated with channel x. Specifying a channel filter is optional.

Trigger:

$$\frac{\kappa\langle\texttt{trigger}\ e\ \texttt{on}\ p\rangle \;\Big\|\; \kappa\langle\texttt{fwd}\ e\ \texttt{at}\ p^{\overline{dir}}\rangle}{\sigma \;\Big\|\; \sigma} \text{ if}$$
$$\sigma \models p^{dir} \in \mathcal{P}_\kappa \wedge p \in \pi \wedge e \in \varepsilon \wedge \varepsilon \subseteq \varepsilon' \wedge \varepsilon' \triangleright \pi^{\overline{dir}} \wedge active(\kappa)$$

where $dir \in \{+,-\}$. If event e can flow through port p in the direction in which it is triggered (p^{dir}), then e is forwarded by the port p from the pole where it was triggered to the opposite pole ($p^{\overline{dir}}$). Here, according to the rule below, e is (1) carried further by the active channels connected to ($p^{\overline{dir}}$), (2) enqueued in the passive channels connected to ($p^{\overline{dir}}$), and (3) delivered to all local subscriptions to ($p^{\overline{dir}}$).

$$\frac{\kappa\langle\texttt{fwd}\ e\ \texttt{at}\ p^{dir}\rangle \;\Big\|\; \kappa\langle\texttt{fwd}\ e\ \texttt{at}\ q_i^{\overline{dir}}\texttt{;enq}\ e\ \texttt{at}\ y_j^{dir}\texttt{;deliver}\ e\ \texttt{to}\ s_l\rangle}{\sigma \;\Big\|\; \sigma}$$

where $\sigma \models p^{dir} \in \mathcal{P}_{\kappa'} \wedge q_i^{\overline{dir}} \in \mathcal{P}_{\kappa'} \wedge p \in \pi \wedge q_i \in \pi \wedge x_i(p,q_i) \in \mathcal{X}_{\kappa'} \wedge active(x_i) \wedge match(e,x_i) \wedge y_j(p,q_j) \in \mathcal{X}_{\kappa'} \wedge \neg active(y_j) \wedge match(e,y_j) \wedge s_l \in \mathcal{S}_{\kappa'} \wedge s_l = sub(h_l(\varepsilon_l), p^{dir}) \wedge \varepsilon \subseteq \varepsilon_l \wedge e \in \varepsilon$.

$$\frac{\kappa\langle\texttt{enq}\ e\ \texttt{at}\ x^{dir}\rangle \;\Big\|\; \kappa\langle\texttt{skip}\rangle}{\sigma \;\Big\|\; \sigma \wedge \sigma'}$$

where $\sigma' \models queue^{dir}(x) = queue^{dir}(x)\#e$.

$$\frac{\kappa\langle\texttt{deliver}\ e\ \texttt{to}\ s\rangle \;\Big\|\; \kappa\langle\texttt{skip}\rangle}{\sigma \;\Big\|\; \sigma \wedge \sigma'}$$

where $s \in \mathcal{S}_{\kappa'} \wedge s = sub(h(\varepsilon), p^{dir}) \wedge \sigma' \equiv queue(p^{dir}) = queue(p^{dir})\#(h, e)$. Delivering an event e to a local subscription, means enqueuing the pair (h, e), where h represents the subscribed handler, into an incoming event queue associated with pole p^{dir}.

Expect:

$$\frac{\kappa\langle\texttt{expect}\ F\rangle \;\Big\|\; \kappa\langle\texttt{skip}\rangle}{\sigma \;\Big\|\; \sigma \wedge \sigma'} \text{ if } \sigma \not\models expects(\kappa, F')$$

where $\sigma' \equiv expects(\kappa, F)$.

$$\frac{\kappa\langle\texttt{expect}\ F\rangle \;\Big\|\; \kappa\langle\texttt{skip}\rangle}{\sigma \wedge \sigma'' \;\Big\|\; \sigma \wedge \sigma'} \text{ if } \sigma'' \models expects(\kappa, F')$$

where $\sigma' \equiv expects(\kappa, F)$. If the `expect` primitive is invoked multiple times within one event handler the last invocation wins.

Start:

$$\frac{\kappa\langle\texttt{start}\ c\rangle \;\Big\|\; c\langle\texttt{start}\ c_i'\rangle}{\sigma \wedge \sigma' \;\Big\|\; \sigma \wedge active(c)} \text{ if}$$
$$\sigma \models c \in \mathcal{C}_\kappa \wedge (\exists sub(h_k, p_j) \in \mathcal{S}_\kappa \vee \exists x_k(q_k, p_j) \in \mathcal{X}_\kappa) \wedge \sigma' \models \neg active(c)$$

where $c_i' \in \mathcal{C}_c \wedge (p_j \in \mathcal{P}_\kappa \vee p_j \in \mathcal{P}_c)$. A component c can be started only if all its ports are connected. Starting a component recursively tries to start all its subcomponents.

Stop:

$$\frac{\kappa\langle\texttt{stop}\ c\rangle \;\Big\|\; c\langle\texttt{stop}\ c_i'\rangle}{\sigma \wedge \sigma' \;\Big\|\; \sigma \wedge \neg active(c)} \text{ if } \sigma \models c \in \mathcal{C}_\kappa \wedge \sigma' \models active(c)$$

where $c_i' \in \mathcal{C}_c$. Stopping a component recursively stops all its subcomponents.

Destroy:

$$\frac{\kappa\langle\texttt{destroy}\ c\rangle \;\Big\|\; c\langle\texttt{destroy}\ c_i'\rangle}{\sigma \wedge \sigma' \;\Big\|\; \sigma} \text{ if } \sigma' \models c \in \mathcal{C}_\kappa \wedge \sigma \models \neg active(c)$$

where $c'_i \in \mathcal{C}_c$. A component can be destroyed only if stopped.

Hold:

$$\frac{\kappa\langle \texttt{hold}\ x\rangle \parallel \kappa\langle \texttt{skip}\rangle}{\sigma \wedge \sigma' \parallel \sigma \wedge \neg active(x)} \text{ if } x \in \mathcal{X}_\kappa \wedge \sigma' \models active(x)$$

Passivating a channel causes all events flowing through the channel to be locally enqueued in the channel, without being delivered to the destination ports, in either direction.

Resume:

$$\frac{\kappa\langle \texttt{resume}\ x\rangle \parallel \kappa\langle \texttt{fwd}\ e_i^{dir}\ \texttt{at}\ p^{\overline{dir}}; \texttt{fwd}\ e_j^{\overline{dir}}\ \texttt{at}\ q^{dir}\rangle}{\sigma \wedge \sigma' \parallel \sigma \wedge active(x)} \text{ if } x(p^{dir}, q^{\overline{dir}}) \in \mathcal{X}_\kappa \wedge \sigma' \models \neg active(x)$$

where $e_i \in queue^{dir}(x) \wedge e_j \in queue^{\overline{dir}}(x)$. Reactivating a channel delivers all the enqueued events to the destination ports in both directions.

Done:

$$\frac{\kappa\langle \texttt{done}\rangle \parallel \kappa\langle S\rangle}{\sigma \parallel \sigma \wedge \sigma'} \text{ if } \sigma \models active(\kappa) \wedge \exists p^{dir} \in \mathcal{P}_\kappa, queue(p^{dir}) \neq \emptyset$$

where $\sigma' \models queue(p^{dir}) = queue(p^{dir}) \setminus (h, e)$ and S represents the compound statement of handler h and (h, e) is the head of the event queue of port p^{dir}. Upon terminating the execution of an event handler or a constructor, a component blocks if it is stopped or if it has no received events. Otherwise, it executes the first event enqueued in an event queue of one of its ports.

Bibliography

[1] Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, February 2012.

[2] Gul Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[3] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, Jean-Philippe Martin, and Alexander Shraer. Reconfiguring replicated atomic storage: A tutorial. *Bulletin of the EATCS*, 102:84–108, 2010.

[4] Marcos K. Aguilera, Idit Keidar, Dahlia Malkhi, and Alexander Shraer. Dynamic atomic storage without consensus. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 17–25, New York, NY, USA, 2009. ACM.

[5] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proceedings of twenty-first ACM Symposium on Operating Systems Principles*, SOSP '07, pages 159–174, New York, NY, USA, 2007. ACM.

[6] Ahmad Al-Shishtawy, Muhammad Asif Fayyaz, Konstantin Popov, and Vladimir Vlassov. Achieving robust self-management for large-scale distributed applications. In *Proceedings of the 2010 Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '10, pages 31–40, Washington, DC, USA, 2010. IEEE Computer Society.

[7] Ahmad Al-Shishtawy, Tareq Jamal Khan, and Vladimir Vlassov. Robust fault-tolerant majority-based key-value store supporting multiple consistency levels. In *IEEE 17th International Conference on Parallel and Distributed Systems*, ICPADS '11, pages 589–596, Tainan, Taiwan, December 2011.

[8] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 334–367, London, UK, 2002. Springer-Verlag.

[9] André Allavena, Alan Demers, and John E. Hopcroft. Correctness of a gossip based membership protocol. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC '05, pages 292–301, New York, NY, USA, 2005. ACM.

[10] Bowen Alpern and Fred B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181 – 185, 1985.

[11] Peter Alvaro, Neil Conway, Joe Hellerstein, and William R. Marczak. Consistency analysis in Bloom: a CALM and collected approach. In *Online Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research*, CIDR '11, pages 249–260, 2011.

[12] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[13] Cosmin Arad and Seif Haridi. Practical protocol composition, encapsulation and sharing in Kompics. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, SASOW '08, pages 266–271, Washington, DC, USA, 2008. IEEE Computer Society.

[14] Cosmin Arad, Jim Dowling, and Seif Haridi. Developing, simulating, and deploying peer-to-peer systems using the Kompics component model. In *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE*, COMSWARE '09, pages 16:1–16:9, New York, NY, USA, 2009. ACM.

[15] Cosmin Arad, Jim Dowling, and Seif Haridi. Building and evaluating P2P systems using the Kompics component framework. In *Proceedings of the 9th IEEE International Conference on Peer-to-Peer Computing*, P2P '09, pages 93–94, Seattle, WA, USA, September 2009. IEEE.

[16] Cosmin Arad, Jim Dowling, and Seif Haridi. Message-passing concurrency for scalable, stateful, reconfigurable middleware. In *Proceedings of the 13th ACM/USENIX International Middleware Conference*, Middleware '12, pages 208–228, New York, NY, USA, 2012. Springer-Verlag New York, Inc.

[17] Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. Brief announcement: atomic consistency and partition tolerance in scalable key-value stores. In *Proceedings of the 26th international conference on Distributed Computing*, DISC '12, pages 445–446, Berlin, Heidelberg, 2012. Springer-Verlag.

[18] Cosmin Arad, Tallat M. Shafaat, and Seif Haridi. CATS: Linearizability and partition tolerance in scalable and self-organizing key-value stores. Technical Report T2012:04, Swedish Institute of Computer Science, 2012.

[19] Cosmin Arad. Kompics Project. `http://kompics.sics.se/`, 2008-2013.

[20] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD dissertation, KTH The Royal Institute of Technology, Stockholm, Sweden, 2003.

[21] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. ISBN 193435600X.

[22] Muhammad Asif Fayyaz. Achieving robust self management for large scale distributed applications using management elements. Master's thesis, KTH The Royal Institute of Technology, Stockholm, Sweden, 2010.

[23] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[24] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004. ISBN 0471453242.

[25] Hagit Attiya and Jennifer L. Welch. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994.

[26] Peter Bailis, Shivaram Venkataraman, Michael J. Franklin, Joseph M. Hellerstein, and Ion Stoica. Probabilistically bounded staleness for practical partial quorums. *Proc. VLDB Endow.*, 5(8):776–787, April 2012.

[27] Ivo Balbaert. *The Way To Go: A Thorough Introduction To The Go Programming Language*. iUniverse, Inc., 2012. ISBN 9781469769165.

[28] Jerry Banks, John Carson, Barry L. Nelson, and David Nicol. *Discrete-Event System Simulation*. Pearson Prentice Hall, 4th edition, 2005. ISBN 0131446797.

[29] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, New York, NY, USA, 2003. ACM.

[30] Basho Riak. `http://wiki.basho.com/Riak.html/`, 2012.

[31] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[32] Ingmar Baumgart, Bernhard Heep, and Stephan Krause. OverSim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007*, pages 79–84, 2007.

[33] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale network services. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, NSDI'04, pages 19–19, Berkeley, CA, USA, 2004. USENIX Association.

[34] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 354–364, New York, NY, USA, 2011. ACM.

[35] BerkeleyDB Java Edition. `http://www.oracle.com/technology/products/berkeley-db/`, 2012.

[36] Philip A. Bernstein and Eric Newcomer. *Principles of Transaction Processing*. The Morgan Kaufmann Series in Data Management Systems. Elsevier Science, 2009. ISBN 9780080948416.

[37] Gautier Berthou and Jim Dowling. P2P VoD using the self-organizing gradient overlay network. In *Proceedings of the second international workshop on Self-organizing architectures*, SOAR '10, pages 29–34, New York, NY, USA, 2010. ACM.

[38] Kenneth P. Birman, Mark Hayden, Oznur Ozkasap, Zhen Xiao, Mihai Budiu, and Yaron Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2): 41–88, May 1999.

[39] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[40] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[41] Daniela Bordencea, Tallat M. Shafaat, Cosmin Arad, Seif Haridi, and Honoriu Valean. Efficient linearizable write operations using bounded global time uncertainty. In *International Symposium on Parallel and Distributed Computing*, ISPDC '13, Los Alamitos, CA, USA, 2013. IEEE Computer Society.

[42] Per Brand. *The design philosophy of distributed programming systems : the Mozart experience*. PhD thesis, KTH, Electronic, Computer and Software Systems, ECS, 2005. QC 20100928.

[43] Eric A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, PODC '00, page 7, New York, NY, USA, 2000. ACM.

[44] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal component model and its support in Java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, September 2006.

[45] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX.

[46] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer Publishing Company, Incorporated, 2nd edition, 2011. ISBN 9783642152597.

[47] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[48] Tushar D. Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

[49] Tushar D. Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, March 1996.

[50] K. Mani Chandy and Jayadev Misra. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Trans. Softw. Eng.*, 5(5):440–452, September 1979.

[51] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[52] Barbara Chapman. The multicore programming challenge. In *Proceedings of the 7th international conference on Advanced parallel processing technologies*, APPT '07, pages 3–3, Berlin, Heidelberg, 2007. Springer-Verlag.

[53] Derek Chen-Becker, Tyler Weir, and Marius Danciu. *The Definitive Guide to Lift: A Scala-based Web Framework*. Apress, Berkely, CA, USA, 2009. ISBN 1430224215.

[54] Shigeru Chiba. Load-time structural reflection in java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 313–336, London, UK, 2000. Springer-Verlag.

[55] Shigeru Chiba and Muga Nishizawa. An easy-to-use toolkit for efficient java bytecode translators. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 364–376, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[56] Gregory V. Chockler, Seth Gilbert, Vincent Gramoli, Peter M. Musial, and Alexander A. Shvartsman. Reconfigurable distributed storage for dynamic networks. *J. Parallel Distrib. Comput.*, 69:100–116, January 2009.

[57] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4): 427–469, December 2001.

[58] Bram Cohen. Incentives build robustness in BitTorrent. In *Proceedings of the 1st Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, 2003.

[59] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[60] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[61] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI '12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[62] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011. ISBN 9780132143011.

[63] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1): 1–42, February 2008.

[64] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, ATC '12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[65] Eric Dashofy, Hazel Asuncion, Scott Hendrickson, Girish Suryanarayana, John Georgas, and Richard Taylor. Archstudio 4: An architecture-based meta-modeling environment. In *Companion to the proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 67–68, Washington, DC, USA, 2007. IEEE Computer Society.

[66] Susan B. Davidson, Hector Garcia-Molina, and Dale Skeen. Consistency in a partitioned network: a survey. *ACM Comput. Surv.*, 17(3):341–370, September 1985.

[67] 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 *Proceedings of Twenty-first Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[68] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4): 372–421, December 2004.

[69] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 1–12, New York, NY, USA, 1987. ACM.

[70] Peter L. Deutsch and Jean-loup Gailly. Zlib compressed data format specification version 3.3. `http://zlib.net/`, 1996.

[71] Michael Dory, Adam Parrish, and Brendan Berg. *Introduction to Tornado*. O'Reilly Media, 2012. ISBN 1449309070.

[72] Jim Dowling and Vinny Cahill. The K-Component architecture meta-model for self-adaptive software. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 81–88, London, UK, 2001. Springer-Verlag.

[73] Jim Dowling and Amir H. Payberah. Shuffling with a croupier: Nat-aware peer-sampling. In *Proceedings of the 32nd IEEE International Conference on Distributed Computing Systems*, ICDCS '12, pages 102–111, Washington, DC, USA, 2012. IEEE Computer Society.

[74] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[75] Amr El Abbadi, Dale Skeen, and Flaviu Cristian. An efficient, fault-tolerant protocol for replicated data management. In *Proceedings of the fourth ACM SIGACT-SIGMOD symposium on Principles of database systems*, PODS '85, pages 215–229, New York, NY, USA, 1985. ACM.

[76] Patrick T. Eugster, Rachid Guerraoui, Anne-Marie Kermarrec, and Laurent Massoulié. Epidemic information dissemination in distributed systems. *Computer*, 37(5):60–67, May 2004.

[77] Patrick Th. Eugster, Rachid Guerraoui, Sidath B. Handurukande, Petr Kouznetsov, and Anne-Marie Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, November 2003.

[78] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Oper. Syst. Rev.*, 40(4):177–190, 2006.

[79] Panagiota Fatourou and Maurice Herlihy. Read-modify-write networks. *Distrib. Comput.*, 17(1):33–46, February 2004.

[80] Alex Feinberg. Project Voldemort: Reliable distributed storage. In *ICDE '11*, 2011. Project site: `http://project-voldemort.com`.

[81] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985.

[82] Wojciech Galuba, Karl Aberer, Zoran Despotovic, and Wolfgang Kellerer. Protopeer: a P2P toolkit bridging the gap between simulation and live deployment. In *Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, Simutools '09, pages 60:1–60:9, ICST, Brussels, Belgium, 2009.

[83] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

[84] Benoit Garbinato and Rachid Guerraoui. Flexible protocol composition in Bast. In *Proceedings of the The 18th International Conference on Distributed Computing Systems*, ICDCS '98, pages 22–, Washington, DC, USA, 1998. IEEE Computer Society.

[85] Martin Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223(4):120–123, October 1970. The original description of Conway's Game of Life.

[86] Dennis M. Geels. *Replay Debugging for Distributed Applications*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2006.

[87] Ali Ghodsi. *Distributed k-ary System: Algorithms for Distributed Hash Tables*. PhD dissertation, KTH The Royal Institute of Technology, Stockholm, Sweden, October 2006.

[88] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, SOSP '79, pages 150–162, New York, NY, USA, 1979. ACM.

[89] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2): 51–59, June 2002.

[90] Seth Gilbert, Nancy A. Lynch, and Alexander A. Shvartsman. Rambo II: rapidly reconfigurable atomic memory for dynamic networks. In *International Conference on Dependable Systems and Networks*, DSN '03, pages 259–268, 2003.

[91] Lisa Glendenning, Ivan Beschastnikh, Arvind Krishnamurthy, and Thomas Anderson. Scalable consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28, New York, NY, USA, 2011. ACM.

[92] Kenneth J. Goldman. *Distributed algorithm simulation using input/output automata*. PhD thesis, Massachusetts Institute of Technology, 1990. AAI0570262.

[93] Jim Gray and Leslie Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, March 2006.

[94] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 3540288457.

[95] Krishna P. Gummadi, Stefan Saroiu, and Steven D. Gribble. King: estimating latency between arbitrary internet end hosts. In *Proceedings of the 2nd ACM SIGCOMM Workshop on Internet measurment*, IMW '02, pages 5–18, New York, NY, USA, 2002. ACM.

[96] Anjali Gupta, Barbara Liskov, and Rodrigo Rodrigues. Efficient routing for peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation - Volume 1*, NSDI '04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[97] Diwaker Gupta, Kashi Venkatesh Vishwanath, Marvin McNett, Amin Vahdat, Ken Yocum, Alex Snoeren, and Geoffrey M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Trans. Comput. Syst.*, 29(2):4:1–4:48, May 2011.

[98] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993. ISBN 0-201-62427-3.

[99] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[100] David Harel and Amnon Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5(4):293–333, October 1996.

[101] Mark Garland Hayden. *The Ensemble System*. PhD thesis, Cornell University, Ithaca, NY, USA, 1998. ISBN 0-591-69952-4.

[102] Apache HBase. `http://hbase.apache.org/`, 2012.

[103] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991.

[104] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 522–, Washington, DC, USA, 2003. IEEE Computer Society.

[105] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123973375.

[106] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3): 463–492, July 1990.

[107] Dieter Hildebrandt and Wilhelm Hasselbring. Simulation-based development of peer-to-peer systems with the RealPeer methodology and framework. *J. Syst. Archit.*, 54(9):849–860, September 2008.

[108] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985. ISBN 0-13-153271-5.

[109] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in Java. In *ECOOP*, pages 329–353, Berlin, Heidelberg, 2010. Springer-Verlag.

[110] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, ATC '10, pages 1–14, Berkeley, CA, USA, 2010. USENIX Association.

[111] Mahmoud Ismail. Nilestore: secure and fault tolerant distributed storage system. `https://code.google.com/p/nilestore/`, 2011.

[112] Márk Jelasity, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. The peer sampling service: experimental evaluation of unstructured gossip-based implementations. In *Proceedings of the 5th ACM/IFIP/USENIX international conference on Middleware*, Middleware '04, pages 79–98, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[113] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Trans. Comput. Syst.*, 23(3):219–252, August 2005.

[114] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. T-Man: Gossip-based fast overlay topology construction. *Comput. Netw.*, 53(13):2321–2339, August 2009.

[115] Márk Jelasity, Spyros Voulgaris, Rachid Guerraoui, Anne-Marie Kermarrec, and Maarten van Steen. Gossip-based peer sampling. *ACM Trans. Comput. Syst.*, 25(3), August 2007.

[116] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Commun. ACM*, 29(4):300–311, April 1986.

[117] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. ISBN 0-471-94148-4. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 2000.

[118] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011. ISBN 1420082795.

[119] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 41st International Conference on Dependable Systems & Networks*, DSN '11, pages 245–256, Washington, DC, USA, 2011. IEEE Computer Society.

[120] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, STOC '97, pages 654–663, New York, NY, USA, 1997. ACM.

[121] Charles E. Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *Proceedings of the 28th ACM conference on Programming language design and implementation*, PLDI '07, pages 179–188, New York, NY, USA, 2007. ACM.

[122] Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 223–234, New York, NY, USA, 2011. ACM.

[123] Maxwell Krohn, Eddie Kohler, and M. Frans Kaashoek. Events can make sense. In *Proceedings of the USENIX Annual Technical Conference*, ATC '07, pages 7:1–7:14, Berkeley, CA, USA, 2007. USENIX Association.

[124] Project Kryo. `https://code.google.com/p/kryo/`, 2013.

[125] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[126] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3(2):125–143, March 1977.

[127] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[128] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[129] Leslie Lamport. On interprocess communication – Parts I and II. *Distributed Computing*, 1(2):77–101, 1986.

[130] Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2): 133–169, May 1998.

[131] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[132] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *ACM SIGACT News*, 41(1):63–73, March 2010.

[133] Butler W. Lampson. How to build a highly available system using consensus. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, WDAG '96, pages 1–17, London, UK, 1996. Springer-Verlag.

[134] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI '09, pages 185–198, Berkeley, CA, USA, 2009. USENIX Association.

[135] Google's LevelDB. `http://code.google.com/p/leveldb/`, 2012.

[136] Jinyang Li, Jeremy Stribling, Robert Morris, M. Frans Kaashoek, and Thomer M. Gil. A performance vs. cost framework for evaluating DHT design tradeoffs under churn. In *the 24th Joint Conference of the IEEE Computer and Comm. Societies*, INFOCOM '05, pages 225–236, Miami, FL, March 2005.

[137] Karl J. Lieberherr. Formulations and benefits of the Law of Demeter. *SIGPLAN Not.*, 24(3):67–78, March 1989.

[138] Karl J. Lieberherr, Ian M. Holland, and Arthur J. Riel. Object-oriented programming: an objective sense of style. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '88, pages 323–334, New York, NY, USA, 1988. ACM.

[139] Michael Lienhardt, Alan Schmitt, and Jean-Bernard Stefani. Oz/K: a kernel language for component-based open programming. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, pages 43–52, New York, NY, USA, 2007. ACM.

[140] Shiding Lin, Aimin Pan, Rui Guo, and Zheng Zhang. Simulating large-scale P2P systems with the WiDS toolkit. In *Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '05, pages 415–424, Washington, DC, USA, 2005. IEEE Computer Society.

[141] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999. ISBN 0201432943.

[142] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. ACM.

[143] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. The SMART way to migrate replicated stateful services. In *Proceedings of the 1st EuroSys European Conference on Computer Systems*, EuroSys '06, pages 103–115, New York, NY, USA, 2006. ACM.

[144] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Trans. Softw. Eng.*, 21(9):717–734, 1995.

[145] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

[146] Nancy A. Lynch and Alexander A. Shvartsman. Robust emulation of shared memory using dynamic quorum-acknowledged broadcasts. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, FTCS '97, pages 272–281, Washington, DC, USA, 1997. IEEE Computer Society.

[147] Nancy A. Lynch and Alexander A. Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the 16th International Conference on Distributed Computing*, DISC '02, pages 173–190, London, UK, 2002. Springer-Verlag.

[148] Nancy A. Lynch and Mark R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, PODC '87, pages 137–151, New York, NY, USA, 1987. ACM.

[149] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[150] Pattie Maes. Computational reflection. In *GWAI '87: Proceedings of the 11th German Workshop on Artificial Intelligence*, pages 251–265, London, UK, 1987. Springer-Verlag.

[151] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, 2002. Springer-Verlag.

[152] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[153] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[154] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. ISBN 0387102353.

[155] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-65869-1.

[156] Hugo Miranda, Alexandre S. Pinto, and Luís Rodrigues. Appia: A flexible protocol kernel supporting multiple coordinated channels. In *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 707–710, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[157] Ahmad Mirzaei and Seyedeh Serveh Sadeghi. Adjustable, delay-based congestion control in a reliable transport protocol over UDP. Master's thesis, KTH The Royal Institute of Technology, Stockholm, Sweden, 2012.

[158] MongoDB. `http://www.mongodb.org/`, 2012.

[159] Alberto Montresor and Márk Jelasity. PeerSim: A scalable P2P simulator. In *Proceedings of the 9th International Conference on Peer-to-Peer Computing*, P2P '09, pages 99–100, Seattle, WA, USA, September 2009. IEEE.

[160] Alberto Montresor, Mark Jelasity, and Ozalp Babaoglu. Chord on demand. In *Proceedings of the Fifth International Conference on Peer-to-Peer Computing*, P2P '05, pages 87–94, Washington, DC, USA, 2005. IEEE Computer Society.

[161] Muhammad Amir Moulavi. Self tuning for elastic storage in cloud environment. Master's thesis, KTH The Royal Institute of Technology, Stockholm, Sweden, 2011.

[162] Muhammad Amir Moulavi, Ahmad Al-Shishtawy, and Vladimir Vlassov. State-space feedback control for elastic distributed storage in a cloud environment. In *The Eighth International Conference on Autonomic and Autonomous Systems*, ICAS '12, St. Maarten, Netherlands Antilles, March 2012.

[163] Project Netty. `http://netty.io/`, 2013.

[164] Salman Niazi and Jim Dowling. Usurp: distributed NAT traversal for overlay networks. In *Proceedings of the 11th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS '11, pages 29–42, Berlin, Heidelberg, 2011. Springer-Verlag.

[165] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM.

[166] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[167] OSGi Alliance. OSGi Release 5. `http://www.osgi.org/Release5`, 2012.

[168] Krzysztof Ostrowski, Ken Birman, Danny Dolev, and Jong Hoon Ahnn. Programming with live distributed objects. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 463–489, Berlin, Heidelberg, 2008. Springer-Verlag.

[169] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[170] Amir H. Payberah, Jim Dowling, Fatemeh Rahimain, and Seif Haridi. Distributed optimization of P2P live streaming overlays. *Computing*, 94(8-10): 621–647, 2012.

[171] Amir H. Payberah, Jim Dowling, and Seif Haridi. Glive: The gradient overlay as a market maker for mesh-based P2P live streaming. In *International Symposium on Parallel and Distributed Computing*, ISPDC '11, pages 153–162, Los Alamitos, CA, USA, 2011. IEEE Computer Society.

[172] Amir H. Payberah, Jim Dowling, and Seif Haridi. Gozar: NAT-friendly peer sampling with one-hop distributed NAT traversal. In *The 11th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS '11, pages 1–14, Berlin, Heidelberg, 2011. Springer-Verlag.

[173] Amir H. Payberah, Jim Dowling, Fatemeh Rahimian, and Seif Haridi. gradienTv: market-based P2P live media streaming on the gradient overlay. In *the 10th international conference on Distributed Applications and Interoperable Systems*, DAIS '10, pages 212–225, Berlin, Heidelberg, 2010. Springer-Verlag.

[174] Amir H. Payberah, Hanna Kavalionak, Vimalkumar Kumaresan, Alberto Montresor, and Seif Haridi. Clive: Cloud-assisted P2P live streaming. In *Proceedings of the 12th International Conference on Peer-to-Peer Computing*, P2P '12, pages 79–90. IEEE, 2012.

[175] Amir H. Payberah, Hanna Kavalionak, Alberto Montresor, Jim Dowling, and Seif Haridi. Lightweight gossip-based distribution estimation. In *The 15th International Conference on Communications*, ICC '13. IEEE, June 2013.

[176] Amir H. Payberah, Fatemeh Rahimian, Seif Haridi, and Jim Dowling. Sepidar: Incentivized market-based P2P live-streaming on the gradient overlay network. *International Symposium on Multimedia*, pages 1–8, 2010.

[177] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, April 1980.

[178] Michael Perrone. Multicore programming challenges. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 1–2, Berlin, Heidelberg, 2009. Springer-Verlag.

[179] Larry L. Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences building PlanetLab. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 351–366, Berkeley, CA, USA, 2006. USENIX Association.

[180] Larry L. Peterson, Norman C. Hutchinson, Sean O'Malley, and Mark B. Abbott. RPC in the x-Kernel: evaluating new design techniques. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 91–101, New York, NY, USA, 1989. ACM.

[181] Michael Piatek, Tomas Isdal, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. Do incentives build robustness in BitTorrent? In *Proceedings of 4th USENIX Symposium on Networked Systems Design & Implementation*, NSDI '07, Cambridge, MA, April 2007. USENIX Association.

[182] Gordon D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[183] Dick Pountain and David May. *A tutorial introduction to Occam programming*. McGraw-Hill, Inc., New York, NY, USA, 1987. ISBN 0-07-050606-X.

[184] Apache Maven Project. `http://maven.apache.org/`, 2002-2013.

[185] Apache MINA Project. `http://mina.apache.org/`, 2003-2012.

[186] Rackspace Cloud Servers. `http://www.rackspace.com/cloud/servers/`, 2012.

[187] Fatemeh Rahimian, Thinh Le Nguyen Huu, and Sarunas Girdzijauskas. Locality-awareness in a peer-to-peer publish/subscribe network. In *The 12th International conference on Distributed Applications and Interoperable Systems*, DAIS '12, pages 45–58, Berlin, Heidelberg, 2012. Springer-Verlag.

[188] Jun Rao, Eugene J. Shekita, and Sandeep Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proc. VLDB Endow.*, 4: 243–254, January 2011.

[189] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 161–172, New York, NY, USA, 2001. ACM.

[190] John C. Reynolds. The discoveries of continuations. *Lisp Symb. Comput.*, 6 (3-4):233–248, November 1993.

[191] Sean Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a DHT. In *In Proceedings of the USENIX Annual Technical Conference*, ATC '04, Berkeley, CA, USA, 2004. USENIX Association.

[192] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the 3rd IFIP/ACM International Conference on Distributed Systems Platforms*, Middleware '01, pages 329–350, London, UK, 2001. Springer-Verlag.

[193] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, Cambridge, MA, USA, 2004. ISBN 0262220695.

[194] Jan Sacha and Jim Dowling. A gradient topology for master-slave replication in peer-to-peer environments. In *Proceedings of the 2005/2006 International conference on Databases, information systems, and peer-to-peer computing*, DBISP2P'05/06, pages 86–97, Berlin, Heidelberg, 2007. Springer-Verlag.

[195] Jan Sacha, Jim Dowling, Raymond Cunningham, and René Meier. Discovery of stable peers in a self-organising peer-to-peer gradient topology. In *the 6th IFIP International conference on Distributed Applications and Interoperable Systems*, DAIS'06, pages 70–83, Berlin, Heidelberg, 2006. Springer-Verlag.

[196] Vivek Sarkar. Programming challenges for petascale and multicore parallel systems. In *Proceedings of the Third international conference on High Performance Computing and Communications*, HPCC '07, pages 1–1, Berlin, Heidelberg, 2007. Springer-Verlag.

[197] Alan Schmitt and Jean-Bernard Stefani. The kell calculus: a family of higher-order distributed process calculi. In *Proceedings of the 2004 IST/FET international conference on Global Computing*, GC'04, pages 146–178, Berlin, Heidelberg, 2005. Springer-Verlag.

[198] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[199] Russell Sears and Raghu Ramakrishnan. bLSM: a general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[200] Tallat M. Shafaat, Bilal Ahmad, and Seif Haridi. ID-replication for structured peer-to-peer systems. In *the 18th international conference on Parallel Processing*, Euro-Par '12, pages 364–376, Berlin, Heidelberg, 2012. Springer-Verlag.

[201] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with network partitions in structured overlay networks. *Peer-to-Peer Networking and Applications*, 2:334–347, 2009.

[202] Tallat M. Shafaat, Ali Ghodsi, and Seif Haridi. Dealing with bootstrapping, maintenance, and network partitions and mergers in structured overlay networks. In *Proceedings of 6th IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, SASO '12, pages 149–158, Washington, DC, USA, 2012. IEEE Computer Society.

[203] Tallat M. Shafaat, Monika Moser, Thorsten Schütt, Alexander Reinefeld, Ali Ghodsi, and Seif Haridi. Key-based consistency and availability in structured overlay networks. In *Proceedings of the 3rd international conference on Scalable information systems*, InfoScale '08, pages 13:1–13:5, ICST, Brussels, Belgium, 2008.

[204] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS '11, pages 386–400, Berlin, Heidelberg, 2011. Springer-Verlag.

[205] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ACM.

[206] Sriram Srinivasan and Alan Mycroft. Kilim: Isolation-typed actors for Java. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP '08, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[207] Wayne P. Stevens, Glenford J. Myers, and Larry L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, June 1974.

[208] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '01, pages 149–160, New York, NY, USA, 2001. ACM.

[209] Michael Stonebraker, Samuel Madden, Daniel Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[210] Daniel Stutzbach and Reza Rejaie. Understanding churn in peer-to-peer networks. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, IMC '06, pages 189–202, New York, NY, USA, 2006. ACM.

[211] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. ISBN 0201745720.

[212] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

[213] Chunqiang Tang. DSF: a common platform for distributed systems research and development. In *Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, Middleware '09, pages 414–436, Berlin, Heidelberg, 2009. Springer-Verlag.

[214] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, New York, NY, USA, 2nd edition, 2001. ISBN 0521794838.

[215] Douglas Terry, Marvin Theimer, Karin Petersen, Alan Demers, Mike Spreitzer, and Carl Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 172–182, New York, NY, USA, 1995. ACM.

[216] The Akka Message-Passing Framework. `http://akka.io/`, 2012.

[217] The Go Programming Language. `http://golang.org/`, 2013.

[218] The Grizzly Project. `http://grizzly.java.net/`, 2013.

[219] The Jetty Web Server. `http://www.eclipse.org/jetty/`, 2013.

[220] The Rust Programming Language. `http://www.rust-lang.org/`, 2013.

[221] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 1–12, New York, NY, USA, 2012. ACM.

[222] Peter Urban, Andre Schiper, and Xavier Defago. Neko: A single environment to simulate and prototype distributed algorithms. In *Proceedings of the The 15th International Conference on Information Networking*, ICOIN '01, pages 503–, Washington, DC, USA, 2001. IEEE Computer Society.

[223] Amin Vahdat, Ken Yocum, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeff Chase, and David Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 271–284, New York, NY, USA, 2002. ACM.

[224] Robbert van Renesse, Kenneth P. Birman, Roy Friedman, Mark Hayden, and David A. Karr. A framework for protocol composition in Horus. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 80–89, New York, NY, USA, 1995. ACM.

[225] Robbert van Renesse, Kenneth P. Birman, and Silvano Maffeis. Horus: a flexible group communication system. *Commun. ACM*, 39(4):76–83, April 1996.

[226] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, Simutools '08, pages 60:1–60:10, ICST, Brussels, Belgium, 2008.

[227] Spyros Voulgaris, Daniela Gavidia, and Maarten Steen. CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays. *Journal of Network and Systems Management*, 13(2):197–217, June 2005.

[228] Dean Wampler. Scala web frameworks: Looking beyond lift. *IEEE Internet Computing*, 15:87–94, 2011.

[229] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001. ISBN 1558605088.

[230] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 230–243, New York, NY, USA, 2001. ACM.

[231] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the 5th symposium on Operating systems design and implementation*, OSDI '02, pages 255–270, New York, NY, USA, 2002. ACM.

[232] Bernard P. Zeigler. Hierarchical, modular discrete-event modelling in an object-oriented environment. *Simulation*, 49(5):219–230, November 1987.

[233] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE J.Sel. A. Commun.*, 22(1):41–53, January 2004.

[234] Hubert Zimmermann. The ISO reference model for open systems interconnection. *IEEE Transactions on Communications*, 28:425–432, April 1980.

Acronyms

ABD Attiya, Bar-Noy, and Dolev. 46, 101, 102, 113, 117, 118, 128, 151, 159

ADL architecture description language. 86, 89

API application programming interface. ix, 60, 91, 166

CALM Consistency As Logical Monotonicity. 155

CAP Consistency, Availability, and network Partition tolerance. 99

CCM CORBA Component Model. 88

CCS Calculus of Communicating Systems. 90

CDN content distribution network. 50

CLI command-line interface. 130

COPS Clusters of Order Preserving Servers. 155

CORBA Common Object Request Broker Architecture. 88, 203

CPU central processing unit. 69, 71, 72, 161

CRDT Commutative Replicated Data Type. 156

CSP Communicating Sequential Processes. 90, 91

DES discrete-event simulation. 58, 61, 63

DEVS Discrete Event System Specification. 91

DHT distributed hash table. 49, 50, 98, 130, 134, 151, 153

DNS Domain Name System. 63

DSF Distributed Systems Foundation. 91

DSL domain-specific language. ix, 63, 65, 79, 161

EJB Enterprise JavaBeans. 88

FIFO First In, First Out. 19, 34, 44, 86

GC garbage collection. 56, 69, 71, 129

GDB GNU Debugger. 92

GNU GNU's Not Unix! 204

GUI graphical user interface. 76, 130

HTML HyperText Markup Language. 52, 129, 130, 136

HTTP HyperText Transfer Protocol. 52

I/O Input/Output. 56, 60, 90, 205

IDE integrated development environment. ix, 67, 161

IP Internet Protocol. 16, 28, 38, 59

ISO International Organization for Standardization. 157

JAR Java ARchive. 84

JRE Java Runtime Environment. 56, 61

JVM Java virtual machine. 59, 62, 66, 79, 145

LAN local area network. 39, 70–72, 133

MINA Multipurpose Infrastructure for Network Applications. 60, 90

MWMR multiple-writer multiple-reader. 102

NAT network address translation. 6, 50, 86, 158, 206

NIO New I/O. 60, 61, 90, 130

OMNeT++ Objective Modular Network Testbed in C++. 88, 91

OS operating system. 5, 28, 61, 67, 134

OSGi Open Services Gateway initiative. 88

OSI Open Systems Interconnection. 157

P2P peer-to-peer. 48, 50, 51, 58, 59, 63, 68–73, 86, 87, 91, 129

PEX Peer Exchange. 50

PNUTS Platform for Nimble Universal Table Storage. 155

POSIX Portable Operating System Interface. 86

RAM random-access memory. 70, 140

RAMBO Reconfigurable Atomic Memory for Basic Objects. 102, 154

RDS Reconfigurable Distributed Storage. 102, 154

RNG random number generator. 61, 65, 135

RPC remote procedure call. 33, 34, 68, 87, 160

RSM replicated state machine. 154

SEDA staged event-driven architecture. 89, 90

SLA service-level agreement. 144

SMART Service Migration And Replication Technique. 102, 154

SMP symmetric multiprocessor. 71

SMR state machine replication. 47

SON structured overlay network. 49, 50

STUN Session Traversal Utilities for NAT. 50

SUT system under test. 69, 70

SWMR single-writer multiple-reader. 101

TCP Transmission Control Protocol. 16, 38, 60, 73

UDP User Datagram Protocol. 6, 16, 38, 60, 73, 86, 159

VM virtual machine. 69, 71

VOD video on demand. 50

WAN wide area network. 70, 71

WiDS WiDS is Distributed Simulator. 91, 206

YCSB Yahoo! Cloud Serving Benchmark. 130, 131, 139–144

Swedish Institute of Computer Science

SICS Dissertation Series

1. Bogumil Hausman, *Pruning and Speculative Work in OR-Parallel PROLOG*, 1990.

2. Mats Carlsson, *Design and Implementation of an OR-Parallel Prolog Engine*, 1990.

3. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990.

4. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991.

5. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991.

6. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991.

7. Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1992.

8. Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1992.

9. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993.

10. Mats Björkman,*Architectures for High Performance Communication*,1993.

11. Stephen Pink, *Measurement, Implementation, and Optimization of Internet Protocols*, 1993.

12. Martin Aronsson, *GCLA. The Design, Use, and Implementation of a Program Development System*, 1993.

13. Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994.

14. Sverker Jansson, *AKL – A Multiparadigm Programming Language*, 1994.

15. Fredrik Orava, *On the Formal Analysis of Telecommunication Protocols*, 1994.

16. Torbjörn Keisu, *Tree Constraints*, 1994.

17. Olof Hagsand, *Computer and Communication Support for Interactive Distributed Applications*, 1995.

18. Björn Carlsson, *Compiling and Executing Finite Domain Constraints*, 1995.

19. Per Kreuger, *Computational Issues in Calculi of Partial Inductive Definitions*, 1995.

20. Annika Waern, *Recognising Human Plans: Issues for Plan Recognition in Human-Computer Interaction*, 1996.

21. Björn Gambäck, *Processing Swedish Sentences: A Unification-Based Grammar and Some Applications*, 1997.

22. Klas Orsvärn, *Knowledge Modelling with Libraries of Task Decomposition Methods*, 1996.

23. Kia Höök, *A Glass Box Approach to Adaptive Hypermedia*, 1996.

24. Bengt Ahlgren, *Improving Computer Communication Performance by Reducing Memory Bandwidth Consumption*, 1997.

25. Johan Montelius, *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*, 1997.

26. Jussi Karlgren, *Stylistic experiments in information retrieval*, 2000.

27. Ashley Saulsbury, *Attacking Latency Bottlenecks in Distributed Shared Memory Systems*, 1999.

28. Kristian Simsarian, *Toward Human Robot Collaboration*, 2000.

29. Lars-åke Fredlund, *A Framework for Reasoning about Erlang Code*, 2001.

30. Thiemo Voigt, *Architectures for Service Differentiation in Overloaded Internet Servers*, 2002.

31. Fredrik Espinoza, *Individual Service Provisioning*, 2003.

32. Lars Rasmusson, *Network capacity sharing with QoS as a financial derivative pricing problem: algorithms and network design*, 2002.

33. Martin Svensson, *Defining, Designing and Evaluating Social Navigation*, 2003.

34. Joe Armstrong, *Making reliable distributed systems in the presence of software errors*, 2003.

35. Emmanuel Frécon, *DIVE on the Internet*, 2004.

36. Rickard Cöster, *Algorithms and Representations for Personalised Information Access*, 2005.

37. Per Brand, *The Design Philosophy of Distributed Programming Systems: the Mozart Experience*, 2005.

38. Sameh El-Ansary, *Designs and Analyses in Structured Peer-to-Peer Systems*, 2005.

39. Erik Klintskog, *Generic Distribution Support for Programming Systems*, 2005.

40. Markus Bylund, *A Design Rationale for Pervasive Computing - User Experience, Contextual Change, and Technical Requirements*, 2005.

41. Åsa Rudström, *Co-Construction of hybrid spaces*, 2005.

42. Babak Sadighi Firozabadi, *Decentralised Privilege Management for Access Control*, 2005.

43. Marie Sjölinder, *Age-related Cognitive Decline and Navigation in Electronic Environments*, 2006.

44. Magnus Sahlgren, *The Word-Space Model: Using Distributional Analysis to Represent Syntagmatic and Paradigmatic Relations between Words in High-dimensional Vector Spaces*, 2006.

45. Ali Ghodsi, *Distributed k-ary System: Algorithms for Distributed Hash Tables*, 2006.

46. Stina Nylander, *Design and Implementation of Multi-Device Services*, 2007.

47. Adam Dunkels, *Programming Memory-Constrained Networked Embedded Systems*, 2007.

48. Jarmo Laaksolahti, *Plot, Spectacle, and Experience: Contributions to the Design and Evaluation of Interactive Storytelling*, 2008.

49. Daniel Gillblad, *On Practical Machine Learning and Data Analysis*, 2008.

50. Fredrik Olsson, *Bootstrapping Named Entity Annotation by Means of Active Machine Learning: a Method for Creating Corpora*, 2008.

51. Ian Marsh, *Quality Aspects of Internet Telephony*, 2009.

52. Markus Bohlin, *A Study of Combinatorial Optimization Problems in Industrial Computer Systems*, 2009.

53. Petra Sundström, *Designing Affective Loop Experiences*, 2010.

54. Anders Gunnar, *Aspects of Proactive Traffic Engineering in IP Networks*, 2011.

55. Preben Hansen, *Task-based Information Seeking and Retrieval in the Patent Domain: Process and Relationships*, 2011.

56. Fredrik Österlind, *Improving Low-Power Wireless Protocols with Timing-Accurate Simulation*, 2011.

57. Ahmad Al-Shishtawy, *Self-Management for Large-Scale Distributed Systems*, 2012.

58. Henrik Abrahamsson, *Network overload avoidance by traffic engineering and content caching*, 2012.

59. Mattias Rost, *Mobility is the Message: Experiment with Mobile Media Sharing*, 2013.

60. Amir Payberah, *Live Streaming in P2P and Hybrid P2P-Cloud Environments for the Open Internet*, 2013.

61. Oscar Täckström, *Predicting Linguistic Structure with Incomplete and Cross-Lingual Supervision*, 2013.

62. Cosmin Ionel Arad, *Programming Model and Protocols for Reconfigurable Distributed Systems*, 2013.

Colophon

This dissertation was typeset using the LaTeX typesetting system developed by Leslie Lamport, based on TeX created by Donald Knuth. It uses the kthesis document class written by Lars Engebretsen based on the memoir class by Peter Wilson. The body text is set in 11/14.4pt on a 30pc measure with Palatino fonts designed by Hermann Zapf.

The dissertation was produced for G5 paper using the microtype package, created by Robert Schlicht, to enable the micro-typographic extensions of character protrusion, font expansion, adjustment of interword spacing, and additional kerning. These extensions, initially contributed by Hàn Thế Thành with pdfTeX, enhance the appearance and readability of the document with a minimum of visual obtrusion.

The bibliography was prepared using BibTeX created by Oren Patashnik. All source code listings were generated using the minted package created by Konrad Rudolph. Scalable screenshots of web pages were generated using wkhtmltopdf written by Jakob Truelsen. The covers were produced using the PSTricks macros introduced by Timothy Van Zandt, and the geometry package by Hideo Umeki.

All plots were generated using MathWorks® MATLAB®. Most diagrams were drawn using Microsoft® PowerPoint®, exported to PDF, and cropped with pdfcrop developed by Heiko Oberdiek. The diagram on page 32 was generated programmatically using the Ti*k*Z and PGF packages developed by Till Tantau. The Kompics logo and the CATS logo were designed using Adobe® Illustrator®.

Kompics is a portmanteau of KTH, component, and SICS. Consequently, the Kompics logo borrows graphical elements from the KTH logo and the pre-2013 logo of SICS. Kompics is also a pun on Multics.

TRITA-ICT/ECS AVH 13:07
ISSN 1653-6363
ISRN KTH/ICT/ECS/AVH-13/07-SE
ISBN 978-91-7501-694-8

SICS Dissertation Series 62
ISSN 1101-1335
ISRN SICS-D-62-SE

COSMIN IONEL ARAD

Programming Model and Protocols for Reconfigurable Distributed Systems

KTH 2013

www.kth.se