Building global and scalable systems with Atomic Multicast

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Samuel Benz

under the supervision of
Fernando Pedone

January 2018

Dissertation Committee

Antonio Carzaniga	Università della Svizzera Italiana, Switzerland
Robert Soulé	Università della Svizzera Italiana, Switzerland
Alysson Bessani	University of Lisbon, Portugal
Benoît Garbinato	University of Lausanne, Switzerland

Dissertation accepted on 29 January 2018

Research Advisor	PhD Program Director
Fernando Pedone	**Walter Binder**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Samuel Benz
Lugano, 29 January 2018

To Mona

Make it work, make it right,
make it fast.

Kent Beck

Abstract

The rise of worldwide Internet-scale services demands large distributed systems. Indeed, when handling several millions of users, it is common to operate thousands of servers spread across the globe. Here, replication plays a central role, as it contributes to improve the user experience by hiding failures and by providing acceptable latency. In this thesis, we claim that atomic multicast, with strong and well-defined properties, is the appropriate abstraction to efficiently design and implement globally scalable distributed systems.

Internet-scale services rely on data partitioning and replication to provide scalable performance and high availability. Moreover, to reduce user-perceived response times and tolerate disasters (i.e., the failure of a whole datacenter), services are increasingly becoming geographically distributed. Data partitioning and replication, combined with local and geographical distribution, introduce daunting challenges, including the need to carefully order requests among replicas and partitions. One way to tackle this problem is to use group communication primitives that encapsulate order requirements.

While replication is a common technique used to design such reliable distributed systems, to cope with the requirements of modern cloud based "always-on" applications, replication protocols must additionally allow for throughput scalability and dynamic reconfiguration, that is, on-demand replacement or provisioning of system resources. We propose a dynamic atomic multicast protocol which fulfills these requirements. It allows to dynamically add and remove resources to an online replicated state machine and to recover crashed processes.

Major efforts have been spent in recent years to improve the performance, scalability and reliability of distributed systems. In order to hide the complexity of designing distributed applications, many proposals provide efficient high-level communication abstractions. Since the implementation of a production-ready system based on this abstraction is still a major task, we further propose to expose our protocol to developers in the form of distributed data structures. B-trees for example, are commonly used in different kinds of applications, including database indexes or file systems. Providing a distributed, fault-tolerant

and scalable data structure would help developers to integrate their applications in a distribution transparent manner.

This work describes how to build reliable and scalable distributed systems based on atomic multicast and demonstrates their capabilities by an implementation of a distributed ordered map that supports dynamic re-partitioning and fast recovery. To substantiate our claim, we ported an existing SQL database atop of our distributed lock-free data structure.

Acknowledgements

I would like to express my gratitude to all the persons who have supported me during my great time while I was working on this thesis. My advisor Fernando Pedone who supported my ideas and guided this research. Daniele Sciascia, Parisa Marandi and Leandro Pacheco for the uncountable and insightful discussions about distributed systems and the subtle details of algorithms.

Further, I would like to thank my committee members: Antonio Carzaniga, Robert Soulé, Alysson Bessani and Benoît Garbinato for their valuable feedback and the time they spent examining this thesis.

Much appreciation to all members of our research group: Ricardo Padilha, Daniele Sciascia, Alex Tomic, Leandro Pacheco de Sousa, Odorico Mendizabal, Tu Dang, Edson Camargo, Paulo Coelho, Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Theo Jepsen and Loan Ton.

Finally, I would like to take the chance to thank Georgios Kontoleon, the Panter AG, Christoph Birkholz, ImpactHub Zurich and Simon Leinen, Jens-Christian Fischer at SWITCH. Last but not least, a special thank to Laurence Feldmeyer and my family. Without you, this work would not have been possible.

Preface

The result of this research appears in the following publications:

S. Benz, P. J. Marandi, F. Pedone and B. Garbinato. Building global and scalable systems with Atomic Multicast. 15th International Middleware Conference (Middleware 2014)

P. J. Marandi, S. Benz, F. Pedone and K. Birman. The Performance of Paxos in the Cloud. 33rd International Symposium on Reliable Distributed Systems (SRDS 2014)

S. Benz, L. Pacheco de Sousa, and F. Pedone. Stretching Multi-Ring Paxos. 31st Annual ACM Symposium on Applied Computing (DADS 2016)

S. Benz and F. Pedone. Elastic Paxos: A Dynamic Atomic Multicast Protocol. 37th IEEE International Conference on Distributed Computing (ICDCS 2017)

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Problem statement

The rise of worldwide Internet-scale services demands large distributed systems. In little less than two decades, we have witnessed the explosion of worldwide online services (e.g., search engines, e-commerce, social networks). These systems typically run on some cloud infrastructure, hosted by datacenters placed around the world. Moreover, when handling millions of users located everywhere on the planet, it is common for these services to operate thousands of servers scattered across the globe. A major challenge for such services is to remain available and responsive in spite of server failures, software updates and an ever-increasing user base. Replication plays a key role here, by making it possible to hide failures and to provide acceptable response time.

While replication can potentially lead to highly available and scalable systems, it poses additional challenges. Indeed, keeping multiple replicas consistent is a problem that has puzzled system designers for many decades. Although much progress has been made in the design of consistent replicated systems [30], novel application requirements and environment conditions (e.g., very large user base, geographical distribution) continue to defy designers. Some proposals have responded to these new "challenges" by weakening the consistency guarantees offered by services. Weak consistency is a natural way to handle the complexity of building scalable systems, but it places the burden on the service users, who must cope with non-intuitive service behavior. Dynamo [40], for instance, overcomes the implications of eventual consistency by letting the application developers decide about the correct interpretation of the returned data. While weak consistency is applicable in some cases, it can be hardly generalized, which helps explain why we observe a recent trend back to strong consistency (e.g.,

[6, 13, 36, 106]).

In order to scale, services typically partition their state and strive to only order requests that depend on each other, imposing a partial order on requests. Sinfonia [6] and S-DUR [101], for example, build a partial order by using a two-phase commit-like protocol to guarantee that requests spanning common partitions are processed in the same order at each partition. Spanner [36] orders requests within partitions using Paxos and across partitions using a protocol that computes a request's final timestamp from temporary timestamps proposed by the involved partitions. This thesis claims that atomic multicast, with strong and well-defined properties, is the appropriate abstraction to efficiently design and implement globally scalable distributed systems.

Additionally, to cope with the requirements of modern cloud based "always-on" applications, replication protocols must further be able to recover from crashes under production workload, allow for elastic throughput scalability and dynamic reconfiguration; that is, on-demand replacement or provisioning of system resources. Nevertheless, existing atomic multicast protocols are *static*, in that creating new multicast groups at run time is not supported. Consequently, replicas must subscribe to multicast groups at initialization, and subscriptions and unsubscriptions can only be changed by stopping all replicas, redefining the subscriptions, and restarting the system. This thesis presents Elastic Paxos, the first *dynamic* atomic multicast protocol. Elastic Paxos allows replicas to dynamically subscribe to and unsubscribe from atomic multicast groups.

Scalable state machine replication has been shown to be a useful technique to solve the above challenges in building reliable distributed data stores [14, 21]. However, implementing a fully functional system, starting from the atomic multicast primitives, supporting required features like recovery or dynamic behavior is a challenging and error-prone task. Providing higher-level abstractions in the form of distributed data structures can hide this complexity from system developers. For example, given a distributed B-tree, services like distributed databases [5] or file systems [75] can be implemented in a distribution transparent manner. Therefore, another goal of this work is to implement a distributed ordered map as a ready-to-use data structure.

Existing distributed data structures often rely on transactions or distributed locking to allow concurrent access. Consequently, operations may abort, a behavior that must be handled by the application. We implemented a distributed ordered map (DMap) that does not rely on transactions or locks for concurrency control. Relying on atomic multicast, all partially ordered operations succeed without ever aborting. Additionally, DMap is scalable, fault-tolerant and supports consistent long-running read operations on snapshots to allow background

data analytics.

In this thesis, we contend that instead of building a partial order on requests using an ad hoc protocol, intertwined with the application code, services have much to gain from relying on a middleware to partially order requests, analogously to how some libraries provide total order as a service (e.g., [9]). Moreover, such a middleware must include support for service recovery and add dynamic reconfiguration, both non trivial requirements which should be abstracted from the application code. As a consequence, application developers should only be exposed to strong consistent geo-distributed data structures as building blocks instead of directly implementing low-level coordination protocols. The research question is: How to achieve scalability, fault tolerance and consistency in practical usable dynamic distributed systems?

1.2 Research contributions

The research conducted within this dissertation provides three major contributions:

URingPaxos This work has contributed an efficient implementation of an atomic multicast protocol [15]. We have shown that atomic multicast is a suitable abstraction to build global and scalable systems [14]. First, we propose an atomic multicast protocol capable of supporting at the same time *scalability* and *strong consistency* in the context of large-scale online services. The Multi-Ring Paxos protocol we describe in this work does not rely on network-level optimizations (e.g., IP-multicast) and allows services to recover from a wide range of failures. Further, we introduce two novel techniques, *latency compensation* and *non-disruptive recovery*, which improve Multi-Ring Paxos's performance under strenuous conditions. Second, we show how to design two services, MRP-Store and DLog, atop URingPaxos and demonstrate the advantages of our proposed approach. Third, we detail the implementation of URingPaxos, MRP-Store, and DLog. Finally, we provide a performance assessment of all these components while we set out to assess its performance under extreme conditions.

Elastic Paxos In today's cloud environments, adding resources to and removing resources from an operational system without shutting it down is a desirable feature. Atomic multicast is a suitable abstraction to build scalable distributed systems, but atomic multicast, as discussed previously, relies on static subscriptions of replicas to groups. Subscriptions are defined at initialization and can only be changed by stopping all processes, redefining the subscriptions, and restart-

ing the system. In this contribution, we motivate and define Elastic Paxos [16], a dynamic atomic multicast protocol. We show how Elastic Paxos can be used to dynamically subscribe replicas to a new multicast stream (i.e., a new partition), which let a replicated data store be repartitioned without service interruption. Further, we demonstrate how dynamic subscriptions offer an alternative approach to reconfiguring Paxos.

DMap To overcome the complexity of implementing dynamic scalable replication protocols from scratch, we claim that developers can gain much from distributed data structures. DMap makes the following contributions. First, we propose a lock-free distributed ordered map with strong consistency guarantees and which implements the Java SortedMap interface. Second, we show how DMap can be used to reliably distribute Java applications, like a transactional database. Third, we detail the implementation of DMap and highlight the underlying replication and ordering techniques. Finally, we provide a performance assessment of all these components.

1.3 Document outline

The remainder of this thesis is structured as follows:

Chapter 2 introduces the system model and formalizes some definitions. Chapter 3 demonstrate how atomic multicast can be used to build global and scalable distributed systems and how recovery under full system load can be achieved. Chapter 4 extends atomic multicast with dynamic behavior and evaluates a distributed key-value store in a highly dynamic cloud environment. Chapter 5 explains how a lock-free concurrent data structure can be distributed using the developed algorithms and how an existing SQL database can be run atop of it. Chapter 6 concludes this thesis. Appendix A details the implementation of the source code library developed within this thesis.

Chapter 2

Background

This chapter will furnish some theoretical background information on topics related to the thesis and introduce the algorithms it relies upon.

2.1 System Model

We assume a distributed system composed of a set $\Pi = \{p_1, p_2, ...\}$ of interconnected processes that communicate through point-to-point message passing. Processes may fail by crashing and subsequently recover, but do not experience arbitrary behavior (i.e., no Byzantine failures).

Processes are either *correct* or *faulty*. A correct process is eventually operational "forever" and can reliably exchange messages with other correct processes. This assumption is only needed to prove liveness properties about the system. In practice, "forever" means long enough for processes to make some progress (e.g., terminate one instance of consensus).

The protocols in this thesis ensure safety under both asynchronous and synchronous execution periods. The FLP impossibility result [48] states that under asynchronous assumptions consensus cannot be both safe and live. To ensure liveness, we assume the system is *partially synchronous* [44]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous, called the *Global Stabilization Time (GST)* [44], is unknown to the processes. When the system behaves asynchronously (i.e., before GST), there are no bounds on the time it takes for messages to be transmitted and actions to be executed; when the system behaves synchronously (i.e., after GST), such bounds exist but are unknown by the processes.

2.2 Consensus and Atomic Broadcast

A fundamental problem in distributed systems is reaching consensus among multiple processes [47]. In a crash failure model, consensus is defined as follows [28]:

Termination: Every correct process eventually decides some value.

Agreement: No two correct processes decide differently.

Uniform integrity: Every process decides at most once.

Uniform validity: If a process decides v, then v was proposed by some process.

The consensus problem is notoriously difficult to solve in the presence of process failures and message losses. How can process a be sure that process b has decided on the same value? In a synchronous system, in which we have the notion of time, the first process can wait on a response or a timeout and proceed based on whatever happens first. A crash of a process can be detected in a synchronous system. But in an asynchronous system there is no notion of time. Tolerating crashes while using asynchronous systems is exactly what we want in practice. One reason to build distributed systems is that we can tolerate failures. Since the synchronous model only shifts the problem and failures in the asynchronous model are not acceptable to reach consensus, we have either to weaken the problem or strengthen the model assumptions. By weakening the problem, we could for example tolerate at most k different values (*k-agreement*) [108]. Another solution to prevent processes in the asynchronous system from not making progress is to use failure detectors [2].

Chandra and Toueg [28] propose a class of algorithms which use failure detectors to solve consensus. Further, they implement atomic broadcast. Atomic broadcast has similar properties to consensus:

Validity: If a correct process AB-broadcasts a message m, then it eventually AB-delivers m.

Agreement: If a process AB-delivers a message m, then all correct processes eventually AB-deliver m.

Uniform integrity: For any message m, every process AB-delivers m at most once, and only if m was previously AB-broadcast by sender(m).

Total order: If two correct processes p and q deliver two messages m and m', then p delivers m before m' if and only if q delivers m before m'.

In fact, it turns out that atomic broadcast can be reduced to consensus and vice versa. To achieve consensus in a distributed system, we can simply atomic broadcast a value. Since we have total order, processes can decide on the first received value. On the other side, we can run a consensus protocol to decide on multiple independent instances of consensus. This sequence of consensus instances can be used to implement atomic broadcast [28]. One consequence of the reduction of atomic broadcast to consensus is that atomic broadcast is not solvable in an asynchronous system in the presence of process crashes.

Despite the difficulties to build consensus and atomic broadcast protocols, they are very important in practice, since the communication paradigm they provide is very powerful. For example, a lot of protocols require a leader, a master process which coordinates the protocol. Once we have leader election, the protocol implementations are trivial. In general, atomic broadcast can be used for many kinds of distributed coordination services, like mutual exclusion.

Another example where atomic broadcast protocols are required is state machine replication [99]. In this form of replication, the commands are sent through atomic broadcast to a set of replicas. Every replica executes the deterministic command in the same order. This results in the same state at every replica (see Section 2.7).

2.3 Paxos

Paxos is a distributed and failure-tolerant consensus protocol. It was proposed by Lamport [69],[70] and combines many properties which are required in practice. While Paxos operates in an asynchronous model and over unreliable channels, it can tolerate crash failures. By using stable storage, processes can recover from failures. To guarantee progress, Paxos assumes a leader-election oracle.

The protocol distinguishes three roles: proposers, acceptors and learners. The algorithm works as follows (Figure 2.1): In phase $1a$ a proposer sends a message with a unique number to all acceptors. If the acceptors never saw a higher number for this consensus instance, they confirm the reservation of the ballot by sending back a phase $1b$ message. If the proposer receives at least a quorum $\lceil (n+1)/2 \rceil$ of acceptor answers, where n is the number of acceptors, it can start

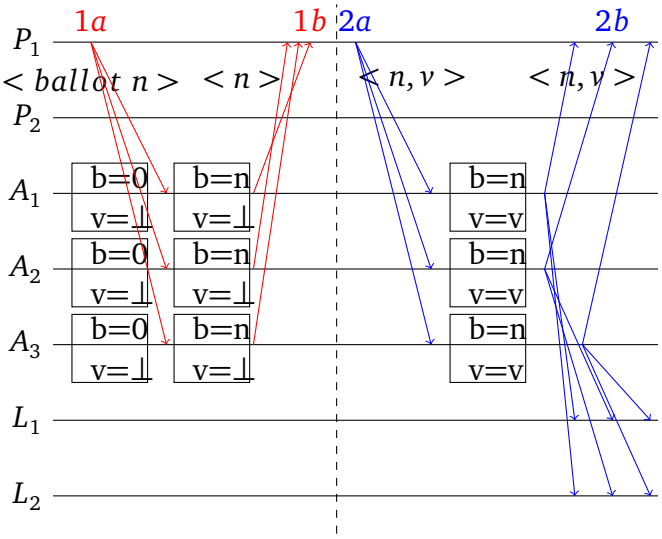

Figure 2.1. Paxos Algorithm

with phase $2a$. To get a quorum, a majority of acceptors must be alive. This means that Paxos requires $2f + 1$ acceptor nodes to tolerate up to f failures.

Phase $2a$ starts with a message, including the value to be proposed and the ballot number, from the proposer to all acceptors. If the ballot in the message corresponds to what the acceptors in phase 1 promised to accept, they will store the value. All acceptors will propagate their decision with a phase $2b$ message.

This is of course the most trivial case, in which no acceptor crashes and multiple proposers do not try to reserve the same consensus instance. None of these scenarios will affect the safety of the protocol. The later, however, could cause liveness problems. A liveness problem can prevent the algorithm from making progress, which would violate the termination property of consensus. Such a scenario can happen when P_1 receives a phase $1b$ message but before the acceptors receive its phase $2a$ message, a second proposer P_2 already increased the ballot with another $1a$ message. P_1 will re-try after a timeout while waiting for a $2b$ message and again send a message $1a$ with increased ballot. With unlucky timing, this can go on forever. Paxos solves the problem by assuming a leader election oracle, which selects one proposer only to execute phase 1.

The above Paxos algorithm solves only consensus for one instance. To use it as an atomic broadcast protocol, Paxos must be extended to run consensus for different incrementing instances.

Paxos is not the only protocol that can be used to implement atomic broadcast.

A good overview of other total order broadcast algorithms is provided in [41]. This survey identifies five categories of protocols. The fixed sequencers, where one process is elected for ordering, and the moving sequencer, which balances the work by transferring the sequencer role across different processes. Further there are privilege-based protocols where the senders can only propose values when they hold a token. The category of communication history-based protocols are like the privilege-based protocols coordinated by the senders. In the case of communication history, all processes can send in parallel. The ordering is achieved by using logical timestamps, like vector clocks [67].

2.4 Ring Paxos

While Paxos brings already a lot of interesting features in its original form, it is not very efficient. Ring Paxos [82] is a derivation of Paxos. It relies on the same safety and liveness properties as Paxos, but optimizes throughput. The new algorithm is based on a few observations in practice.

- The throughput of IP multicast scales constantly with the number of receivers, while IP unicast decreases proportional to the number of receivers.
- Minimizing packet loss by limiting the number of IP multicast senders to one.
- Limiting the number of incoming connections per host to one is more efficient than having many.

Concluding all of these observations, Ring Paxos has one coordinator which is also an acceptor and the multicast sender. Proposers send the values to this coordinator. Optimized phase 1 and phase 2 are executed in a ring of the acceptors. The decisions are multicast to all nodes.

While Ring Paxos can reach almost nominal network bandwidth (e.g., 1 Gbit/s) with a good average latency [82], it depends on IP multicast. In some environments (e.g., wide-area networks), however, IP multicast is not available. To overcome this shortcoming, multicast can be replaced by pipelined unicast connections. Unicast pipelining almost achieves the same throughput as multicast, and may introduce delays, a price which has to be payed to port Ring Paxos to WAN links.

The Ring Paxos algorithm implemented in this thesis is based on unicast connections only. In this case, all nodes form a ring, not only the acceptors (Figure 2.2).

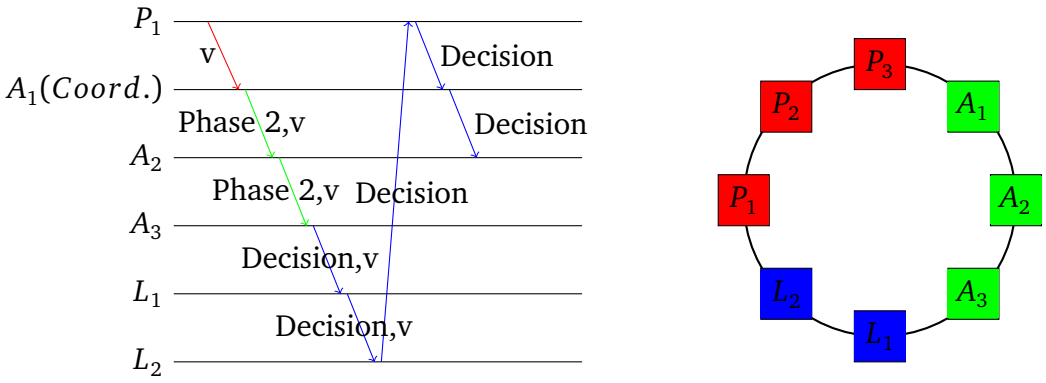

Figure 2.2. Unicast Ring Paxos Algorithm

Hereafter we call this protocol implementation URingPaxos. While Paxos uses two different messages per phase (a/b), URingPaxos uses several. Phase 1 has only one message with a vote count. Phase 2 has a message with a vote count and additionally a separate decision message. A proposer starts sending a message v to its ring successor. This node will store the value and forward the message until it reaches the coordinator. Where the roles are placed in the ring is not important for correctness, but it has an impact on the overall latency.

Once the coordinator receives a value, since it is also an acceptor, it starts learning the value with a phase 2 message. Phase 1 is not shown in the figure and can be done for multiple instances before a value is proposed.

When an acceptor receives a phase 2 message, it will increase the vote count in the message and store the value. At this point, the value is not yet decided. The decision message is issued by the last acceptor in the ring, if the vote count is larger than or equal to the quorum. The decision messages are forwarded in the ring until they reach the predecessor of the last acceptor.

Phase 2 and the decision message do not always include the full value. The algorithm ensures that every value is only transmitted once in the ring. This is possible because the value contains a unique identifier and an actual value. The later can be removed when not needed before forwarding in the ring.

2.5 Atomic Multicast

Ring Paxos solves the shortcoming of Paxos by making it fast in terms of throughput. However, the resulting protocol is not scalable. Scalability is the ability to increase the overall performance by adding more resources. Ring Paxos is not

scalable since all traffic must be submitted to all acceptors. Thus adding more acceptors, doesn't allow more messages to be ordered.

Atomic multicast [55] is an abstraction used by process groups to communicate. It defines two communication primitives: *multicast*(γ, m) and *deliver*(m). Client processes invoke *multicast*(γ, m) to submit requests, encoded in message m, to the replica processes associated with stream γ. Replicas subscribe to one or more multicast streams, and deliver client requests with primitive *deliver*(m). Atomic multicast is defined as follows:

Agreement: If a correct process delivers a message m, then every correct process in γ eventually delivers m.

Validity: If a correct process multicast a message m, then every correct process in γ eventually delivers m.

Integrity: For any message m, every correct process p deliver m at most once.

Partial order: If two correct processes p and q deliver two messages m and m', then p delivers m before m' iff q delivers m before m'.

2.6 Multi-Ring Paxos

Multi-Ring Paxos [81] is an atomic multicast algorithm designed for scalability. The core idea behind it is simple. By using multiple rings, the single-ring performance can be summed up. To guarantee total order, Multi-Ring Paxos uses a deterministic merge function to combine the output of multiple rings.

The merge function can be a simple round-robin procedure. First take m values from the first ring, than m values from the second ring and so on (Figure 2.3). This assumes that all rings make progress at the same speed. If this is not the case, then sooner or later, some of the learners in the faster rings will wait until the slower rings deliver enough values. To overcome this problem, the coordinator of every ring keeps track of its ring throughput. The maximum throughput of the fastest ring in the system is a configuration parameter (λ). Each coordinator compares its actual throughput with λ and issues enough skip messages every time interval Δt to match λ.

A skip message is a special *null* value, which means that the multi-ring learner can skip one value since there are not enough values proposed in this ring. Several skip messages are batched and learned in one single Paxos instance. This keeps the overhead of skip messages minimal.

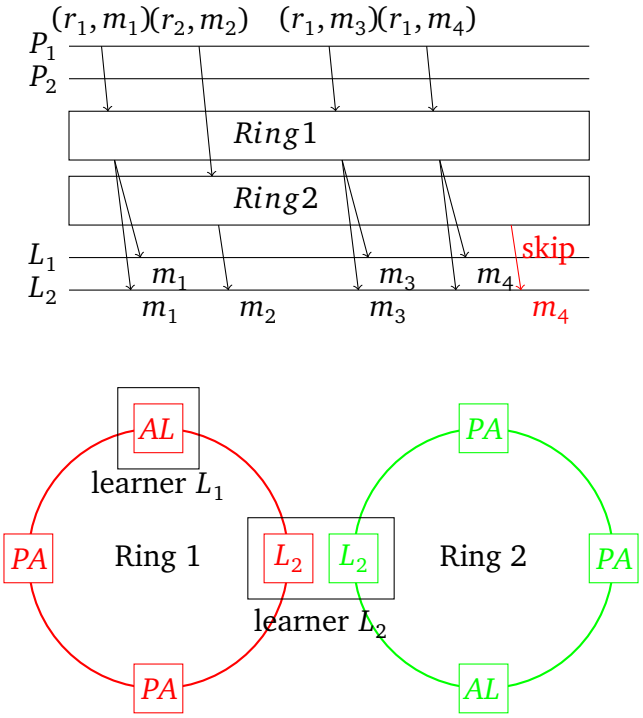

Figure 2.3. Multi-Ring Paxos Algorithm

With this approach, it is also possible to combine rings with different throughput. While one goal is to scale local disk writes, equally distribute ring throughput; another goal is to combine different WAN links. In the case of WAN links, fast local rings could be connected to slower but globally connected rings.

2.7 State Machine Replication

Sate machine replication, also called active replication, is a common approach to building fault-tolerant systems [99]. Replicas, which can be seen as deterministic state machines, receive and apply deterministic commands in total order [69]. Their state therefore evolves identically and an ensemble of multiple replicas form a multi-master data store.

With one exception, in this thesis we consider strongly consistent services that ensure linearizability. A concurrent execution is linearizable if there is a sequential way to reorder the client operations such that: (1) it respects the real-time semantics of the objects, as determined in their sequential specs, and (2) it respects the order of non-overlapping operations among all clients [59].

In Section 3.5.1 we use a weaker consistency criteria: sequential consistency. A concurrent execution is sequentially consistent if there is a sequential way to reorder the client operations such that: (1) it respects the semantics of the objects, as determined in their sequential specs, and (2) it respects the order of operations at the client that issued the operations [68].

State machine replication [67, 99] simplifies the problem of implementing highly available linearizable services by decomposing the *ordering* of requests across replicas from the *execution* of requests at each replica. Requests can be ordered using atomic broadcast and, as a consequence, service developers can focus on the execution of requests, which is the aspect most closely related to the service itself. State machine replication requires the execution of requests to be deterministic, so that when provided with the same sequence of requests, every replica will evolve through the same sequence of states and produce the same results.

State machine replication, however, does not lead to services that can scale throughput with the number of replicas. Increasing the number of replicas results in a service that tolerates more failures, but does not necessarily serve more clients per time unit. Several systems resort to state partitioning (i.e., sharding) to provide scalability (e.g., Calvin [106], H-Store [64]). Scalable performance and high availability can be obtained by partitioning the service state and replicating each partition with state machine replication. To submit a request for execution, the client atomically multicasts the request to the appropriate partitions [21]. Performance will scale as long as the state can be partitioned in such a way that most commands are executed by a single partition only. Atomic multicast helps design highly available and scalable services that rely on the state machine replication approach by ensuring proper ordering of both single- and multi-partition requests.

Chapter 3

Scalable and Reliable Services

3.1 Introduction

Internet-scale services are widely deployed today. These systems must deal with a virtually unlimited user base, scale with high and often fast demand of resources, and be always available. In addition to these challenges, many current services have become geographically distributed. Geographical distribution helps reduce user-perceived response times and increase availability in the presence of node failures and datacenter disasters (i.e., the failure of an entire datacenter). In these systems, data *partitioning* (also known as sharding) and *replication* play key roles.

Data partitioning and replication can lead to highly scalable and available systems, however, they introduce daunting challenges. Handling partitioned and replicated data has created a dichotomy in the design space of large-scale distributed systems. One approach, known as *weak consistency*, makes the effects of data partitioning and replication visible to the application.

Weak consistency provides more relaxed guarantees and makes systems less exposed to impossibility results [48, 52]. The tradeoff is that weak consistency generally leads to more complex and less intuitive applications. The other approach, known as *strong consistency*, hides data partitioning and replication from the application, simplifying application development. Many distributed systems today ensure some level of strong consistency by totally ordering requests using the Paxos algorithm [69], or a variation thereof. For example, Chubby [25] is a Paxos-based distributed locking service at the heart of the Google File System (GFS); Ceph [111] is a distributed file system that relies on Paxos to provide a consistent cluster map to all participants; and Zookeeper [60] turns a Paxos-like total order protocol into an easy-to-use interface to support group messaging and

distributed locking.

Strong consistency requires ordering requests across the system in order to provide applications with the illusion that state is neither partitioned nor replicated. Different strategies have been proposed to order requests in a distributed system, which can be divided into two broad categories: those that impose a total order on requests and those that partially order requests.

Reliably delivering requests in total and partial order has been encapsulated by atomic broadcast and atomic multicast, respectively [57]. We extend Multi-Ring Paxos, a scalable atomic multicast protocol introduced in [81], to (a) cope with large-scale environments and to (b) allow services to recover from a wide range of failures (e.g., the failures of all replicas). Addressing these aspects required a redesign of Multi-Ring Paxos and a new library called URingPaxos: Some large-scale environments (e.g., public datacenters, wide-area networks) do not allow network-level optimizations (e.g., IP-multicast [81]) that can significantly boost bandwidth. Recovering from failures in URingPaxos is challenging because it must account for the fact that replicas may not all have the same state. Thus, a replica cannot recover by installing any other replica's image.

We developed the URingPaxos library and two services based on it: MRP-Store, a key-value store, and DLog, a distributed log. These services are at the core of many internet-scale applications. In both cases, we could show that the challenge of designing and implementing highly available and scalable services can be simplified if these services rely on atomic multicast. Our performance evaluation assesses the behavior of URingPaxos under various conditions and shows that MRP-Store and DLog can scale in different scenarios. We also illustrate the behavior of MRP-Store when servers recover from failures.

This chapter makes the following contributions. First, we propose an atomic multicast protocol capable of supporting at the same time *scalability* and *strong consistency* in large-scale environments. Intuitively, URingPaxos composes multiple instances of Ring Paxos to provide efficient message ordering. The URing-Paxos protocol we describe in this chapter does not rely on network-level optimizations (e.g., IP-multicast) and allows services to recover from a wide range of failures. Further, we introduce two novel techniques, *latency compensation* and *non-disruptive recovery*, which improve URingPaxos's performance under strenuous conditions. Second, we show how to design two services, MRP-Store and DLog, atop URingPaxos and demonstrate the advantages of our proposed approach. Third, we detail the implementation of URingPaxos, MRP-Store, and DLog. Finally, we provide a performance assessment of all these components while we set out to assess their performance under extreme conditions. Our performance assessment was guided by our desire to answer the following ques-

tions.

- Can URingPaxos deliver performance that matches high-end networks (i.e., 10 Gbps)?
- How does a recovering replica impact the performance of operational replicas computing at peak load?
- URingPaxos ensures high performance despite imbalanced load in combined rings with a skip mechanism. Can URingPaxos's skip mechanism handle highly skewed traffic?
- How many combined rings in a learner are "too many"?
- Can URingPaxos deliver usable performance when deployed around the globe and subject to disasters?

3.2 Why Atomic Multicast

Two key requirements for current online services are (1) the immunity to a wide range of failures and (2) the ability to serve an increasing number of user requests. The first requirement is usually fulfilled through *replication* within and across datacenters, possibly located in different geographical areas.

The second requirement is satisfied through *scalability*, which can be "horizontal" or "vertical". Horizontal scalability (often simply *scalability*) consists in adding more servers to cope with load increases, whereas vertical scalability consists in adding more resources (e.g., processors, disks) to a single server. Horizontal scalability boils down to partitioning the state of the replicated service and assigning partitions (i.e., so-called shards) to the aforementioned geographically distributed servers.

Consistency vs. scalability. The partition-and-replicate approach raises a challenging concern: How to preserve service consistency in the presence of requests spanning multiple partitions, each partition located in a separate data center, in particular when failures occur? When addressing this issue, middleware solutions basically differ in how they prioritize *consistency vs. scalability*, depending on the semantic requirements of the services they support. That is, while some services choose to relax consistency in favor of scalability and low latency, others choose to tolerate higher latency, possibly sacrificing availability (or at least its perception thereof by end-users), in the interest of service integrity.

Prioritizing scalability. TAO, Facebook's distributed data store [24] is an example of a middleware solution that prioritizes scalability over consistency: with TAO, strong consistency is ensured within partitions and a form of eventual consistency is implemented across partitions. This implies that concurrent requests accessing multiple partitions may lead to inconsistencies in Facebook's social graph. To lower the chance of potential conflicts, data access patterns can be considered when partitioning data (e.g., entries often accessed together can be located in the same partition). Unfortunately, such optimizations are only possible if knowledge about data usage is known a priori, which is often not the case.

Some middleware solutions, such as S-DUR [101] and Sinfonia [6], rely on two-phase commit [17] to provide strong consistency across partitions. Scatter [53] on the other hand prohibits cross-partition requests and uses a two-phase commit protocol to merge commonly accessed data into the same partition. A common issue with storage systems that rely on atomic commitment is that requests spanning multiple partitions (e.g., cross-partition transactions) are not totally ordered and can thus invalidate each other, leading to multiple aborts. For example, assume objects x and y in partitions p_x and p_y, respectively, and two transactions T_1 and T_2 where T_1 reads x and updates the value of y, and T_2 reads y and updates the value of x. If not ordered, both transactions will have to abort to ensure strong consistency (i.e., serializability).

Prioritizing consistency. When it comes to prioritizing consistency, some proposals totally order requests before their execution, as in state machine replication [99], or execute requests first and then totally order the validation of their execution, as in deferred update replication [92]. (With state machine replication requests typically execute sequentially [1]; with deferred update replication requests can execute concurrently.) Coming back to our example of conflicting transactions T_1 and T_2, while approaches based on two-phase commit lead both transactions to abort, with deferred update replication only one transaction aborts [91], and with state machine replication both transactions commit. Many other solutions based on total order exist, such as Spanner [36] and Calvin [106].

The Isis toolkit [22] and later Transis [10] pioneered the use of totally ordered group communication at the middleware level. With Isis, total order is enforced at two levels: first, a consistent sequence of views listing the replicas considered alive is atomically delivered to each replica; then, messages can be totally ordered within each view, using an atomic broadcast primitive. In the

[1] Some proposals exploit application semantics to allow concurrent execution of commands in state machine replication (e.g., [38, 66, 79, 80]).

same vein, many middleware solutions rely on atomic broadcast as their basic communication primitive to guarantee total order.

The best of both worlds. We argue that atomic multicast is the right communication abstraction when it comes to combining consistency and scalability. Indeed, atomic broadcast implies that all replicas are in the same group and must thus receive each and every request, regardless its actual content, which makes atomic broadcast an inefficient communication primitive when data is partitioned and possibly spread across datacenters. With atomic multicast, on the contrary, each request is only sent to the replicas involved in the request, which is more efficient when data is partitioned and possibly distributed across datacenters. Compared to solutions that rely on atomic broadcast to ensure consistency within each partition and an ad hoc protocol to handle cross-partition requests, atomic multicast is more advantageous in that requests are ordered both within and across partitions. As a matter of fact, most existing middleware solutions rely on atomic broadcast only to ensure consistency within each partition, while ensuring cross-partition consistency in an ad hoc manner, i.e., without relying on a well-defined communication primitive.

Not only do we advocate atomic multicast as basic communication primitive to build middleware services, but we also believe that the traditional group addressing semantics should be replaced with one that better fits the context of large-scale Internet services. With traditional atomic multicast primitives (e.g., [42, 56, 95, 97, 98]), a client can address multiple *non-intersecting* groups of servers, where each server can only belong to a single group. Rather, we argue that clients should address one group per multicast and each server should be able to subscribe to any group it is interested in, i.e., any replication group corresponding to the shards the server is currently replicating, similarly to what IP multicast supports. As we shall see in Section 3.3, this somehow "inverted" group addressing semantics allows us to implement a scalable atomic multicast protocol.

Atomic Multicast and the CAP theorem [52]. Atomic multicast ensures consistency, in the form of a well-defined order property, is partition-tolerant, in the sense that partitions may happen, but violates availability: A ring is only available if a majority of acceptors remains in the same partition and can communicate. A learner will be available as long as all the rings it subscribes to remain available.

Recovering from failures. The ability to safely recover after a failure is an essential aspect of the failure immunity requirement of large-scale middleware services. Furthermore, fast crash recovery is of practical importance when in-

memory data structures are used to significantly decrease latency. Yet, similarly to what is done to ensure cross-partition consistency, existing middleware solutions tend to deal with recovery issues in an ad hoc manner, directly at the service level, rather than factor out the solution to recovery issues in the underlying communication layer. A different approach consists in relying on atomic multicast to orchestrate checkpointing and coordinate checkpoints with the trimming of the logs used by the ordering protocol. This is particularly important in the context of atomic multicast since recovery in partitioned systems is considerably more complex than recovery in single partition systems (see Section 3.4).

Architecture overview. Figure 3.1 presents an overview of our middleware solution based on atomic multicast, implemented by URingPaxos. Online services can build on atomic multicast's ordering and recovery properties, as described in the next two sections. As suggested by this figure, atomic multicast naturally supports state partitioning, an important characteristic of scalable services, and no ad hoc protocol is needed to handle coordination among partitions.

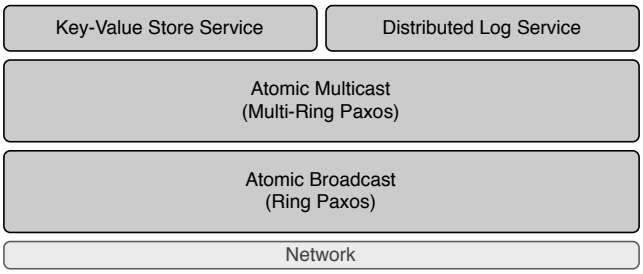

Figure 3.1. Architecture overview.

3.3 URingPaxos

Intuitively, URingPaxos turns an atomic broadcast protocol based on Ring Paxos into an atomic multicast protocol. That is, URingPaxos is implemented as a collection of coordinated Ring Paxos instances, or rings for short, such that a distinct multicast group is assigned to each ring. Each ring in turn relies on a sequence of consensus instances, implemented as an optimized version of Paxos.

URingPaxos is based on Multi-Ring Paxos which was introduced in [81]. In this section, we recall how URingPaxos works and describe a variation of Ring Paxos that does not rely on network-level optimizations (e.g., IP-multicast) to achieve high throughput. In the next section, we introduce URingPaxos's recovery.

Ring Paxos. Similarly to Paxos, Ring Paxos [82] differentiates processes as *proposers*, *acceptors*, and *learners*, where one of the acceptors is elected as the *coordinator*. All processes in Ring Paxos communicate through a unidirectional ring overlay, as illustrated in Figure 3.2 (a). Using a ring topology for communication enables a balanced use of networking resources and results in high performance.

Figure 3.2 (b) illustrates the operations of an optimized Paxos, where Phase 1 is pre-executed for a collection of instances. When a proposer proposes a value (i.e., the value is atomically broadcast), the value circulates along the ring until it reaches the coordinator. The coordinator proposes the value in a Phase 2A message and forwards it to its successor in the ring together with its own vote, that is, a Phase 2B message. If an acceptor receives a Phase 2A/2B message and agrees to vote for the proposed value, the acceptor updates Phase 2B with its vote and sends the modified Phase 2A/2B message to the next process in the ring. If a non-acceptor receives a Phase 2A/2B message, it simply forwards the message as is to its successor. When the last acceptor in the ring receives a majority of votes for a value in a Phase 2B message, it replaces the Phase 2B message by a decision message and forwards the outcome to its successor. Values and decisions stop circulating in the ring when all processes in the ring have received them. A process learns a value once it receives the value and the decision that the value can be learned (i.e., the value is then delivered). To optimize network and CPU usage, different types of messages for several consensus instances (e.g., decision, Phase 2A/2B) are often grouped into bigger packets before being forwarded. Ring Paxos is oblivious to the relative position of processes in the ring. Ring configuration and coordinator's election are handled with a coordination system, such as Zookeeper.

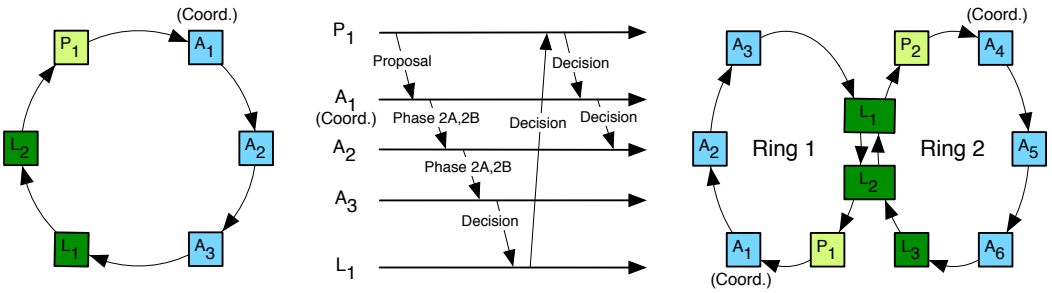

Figure 3.2. (a) The various process roles in Ring Paxos disposed in one logical ring; (b) an execution of a single instance of Ring Paxos; and (c) a configuration of URingPaxos involving two rings (learners L_1 and L_2 deliver messages from Rings 1 and 2, and leaner L_3 delivers messages from Ring 2 only).

URingPaxos. With URingPaxos, each Learner can subscribe to as many rings as it wants to and participates in coordinating multiple instances of Ring Paxos for those rings. In Figure 3.2 (c), we picture a deployment of URingPaxos with two rings and three learners, where learners L1 and L2 subscribe to rings 1 and 2, and learner L3 subscribes only to ring 2. The coordination between groups relies on two techniques, *deterministic merge* and *rate leveling*, controlled with three parameters: M, Δ, and λ.

Initially, a proposer multicasts a value to group γ by proposing the value to the coordinator responsible for γ. Then, Learners use a *deterministic merge* strategy to guarantee atomic multicast's ordered delivery property: Learners deliver messages from rings they subscribe to in round-robin, following the order given by the ring identifier. More precisely, a learner delivers messages decided in M consensus instances from the first ring, then delivers messages decided in M consensus instances from the second ring, and so on and then starts again with the next M consensus instances from the first ring.

Since multicast groups may not be subject to the same load, with the deterministic merge strategy described above replicas would deliver messages at the speed of the slowest multicast group, i.e., the group taking the longest time to complete M consensus instances. To counter the effects of unbalanced load, URingPaxos uses a *rate leveling* strategy whereby the coordinators of slow rings periodically propose to skip consensus instances. That is, at regular Δ intervals, a coordinator compares the number of messages proposed in the interval with the maximum expected rate λ for the group and proposes enough skip instances to reach the maximum rate. To skip an instance, the coordinator proposes null values in Phase 2A messages. For performance, the coordinator can propose to skip several consensus instances in a single message.

3.4 Recovery

For a middleware relying on URingPaxos to be complete and usable, processes must be able to recover from failures. More precisely, recovery should allow processes to (a) restart their execution after failures and (b) limit the amount of information needed for restart. URingPaxos's recovery builds on Ring Paxos's recovery. In the following, we first describe recovery in Ring Paxos (Section 3.4.1) and then detail the subtleties involving recovery in URingPaxos (Section 3.4.2).

3.4.1 Recovery in Ring Paxos

The mechanism used by a process to recover from a failure in Ring Paxos depends on the role played by the process. In a typical deployment of Ring Paxos (e.g., state machine replication [67, 99]), clients propose commands and replicas deliver and execute those commands in the same total order before responding to the clients. In this case, clients act as proposers and replicas as learners, while acceptors ensure ordered delivery of messages. In the following, we focus the discussion on the recovery of acceptors and replicas. Recovering clients is comparatively an easier task.

Acceptor Recovery. Acceptors need information related to past consensus instances in order to serve retransmission requests from recovering replicas. So, before responding to a coordinator's request with a Phase 1B or Phase 2B message, an acceptor must log its response onto stable storage. This ensures that upon recovering from a failure, the acceptor can retrieve data related to consensus instances it participated in before the failure. In principle, an acceptor must keep data for every consensus instance in which it participated. In practice, it can coordinate with replicas to trim its log, that is, to delete data about old consensus instances.

Replica Recovery. When a replica resumes execution after a failure, it must build a state that is consistent with the state of the replicas that did not crash. For this reason, each replica periodically checkpoints its state onto stable storage. Then, upon resuming from a failure, the replica can read and install its last stored checkpoint and contact the acceptors to recover the commands missing from this checkpoint, i.e., the commands executed after the replica's last checkpoint.

Optimizations. The above recovery procedure is optimized as follows. If the last checkpointed state of a recovering replica is "too old",[2] the replica builds an updated state by retrieving the latest checkpoint from an operational replica. This optimization reduces the number of commands that must be recovered from the acceptors, at the cost of transferring the complete state from a remote replica.

3.4.2 Recovery in URingPaxos

Recovery in URingPaxos is more elaborate than in Ring Paxos. This happens because in URingPaxos replicas may deliver messages from different multicast groups and thus evolve through different sequences of states. We call the set of

[2]That is, it would require the processing of too many missing commands in order to build an up-to-date consistent state.

replicas that deliver messages from the same set of multicast groups a *partition*. Replicas in the same partition evolve through the same sequence of states. Therefore, in URingPaxos, a recovering replica can only recover a remote checkpoint, to build an updated state, from another replica in the same partition.

As in Ring Paxos, replicas periodically checkpoint their state. Because a replica p's state may depend on commands delivered from multiple multicast groups, however, p's checkpoint in URingPaxos is identified by a tuple k_p of consensus instances, with one entry in the tuple per multicast group. A checkpoint identified by tuple k_p reflects commands decided in consensus instances up to $k[x]_p$, for each multicast group x that p subscribed to. Since entries in k_p are ordered by group identifier and replicas deliver messages from groups they subscribe to in round-robin, in the order given by the group identifier, predicate 3.1 holds for any state checkpointed by replica p involving multicast groups x and y:

$$x < y \Rightarrow k[x]_p \geq k[y]_p \tag{3.1}$$

Note that Predicate 3.1 establishes a total order on checkpoints taken by replicas in the same partition.

Periodically, the coordinator of a multicast group x asks replicas that subscribe to x for the highest consensus instance that acceptors in the corresponding ring can use to safely trim their log. Every replica p replies with its highest safe instance $k[x]_p$ to the coordinator, reflecting the fact that the replica has checkpointed a state containing the effects of commands decided up to instance $k[x]_p$. The coordinator waits for a quorum Q_T of answers from the replicas, computes the lowest instance number $K[x]_T$ out of the values received in Q_T and sends $K[x]_T$ to all acceptors. That is, we have that the following predicate holds for $K[x]_T$:

$$\forall p \in Q_T : K[x]_T \leq k[x]_p \tag{3.2}$$

Upon receiving the coordinator's message, each acceptor can then trim its log, removing data about all consensus instances up to instance $K[x]_T$.

A recovering replica contacts replicas in the same partition and waits for responses from a recovery quorum Q_R. Each replica q responds with the identifier k_q of its most up-to-date checkpoint, containing commands up to consensus instances in k_q. The recovering replica selects the replica with the most up-to-date checkpoint available in Q_R, identified by tuple K_R such that:

$$\forall q \in Q_R : k_q \leq K_R \tag{3.3}$$

If Q_T and Q_R intersect, then by choosing the most up-to-date checkpoint in Q_R, identified by K_R, the recovering replica can retrieve any consensus instances missing in the selected checkpoint since such instances have not been removed by the acceptors yet.

Indeed, since Q_T and Q_R intersect, there is at least one replica r in both quorums. For each multicast group x in the partition, from Predicates 3.1 and 3.3, we have $k[x]_r \leq K_R[x]$. Since r is in Q_T, from Predicate 3.2, we have $K_T[x] \leq k[x]_r$ and therefore:

$$K_T \leq k_r \leq K_R \tag{3.4}$$

which then results in:

$$K_T \leq K_R \tag{3.5}$$

Predicate 3.5 implies that for every multicast group x in the most up-to-date checkpoint in Q_R, the acceptors involved in x have trimmed consensus instances at most equal to the ones reflected in the checkpoint. Thus, a recovering replica will be able to retrieve any instances decided after the checkpoint was taken.

3.4.3 Latency compensation

The skip calculation described in Section 3.3 is very effective in networks subject to small latencies (e.g., within a datacenter). However, with large and disparate latencies (e.g., geographical deployments), a late skip message may delay the delivery of messages at a learner (see Figure 3.2(c)). This delay might happen even if the number of skip instances is accurately calculated to account for imbalanced traffic among rings.

We overcome this problem by revisiting the skip mechanism to take into consideration the approximate time skip messages need to reach their concerned learners. In equation (3.6), avg_delay is an approximated average of the delays between the ring coordinator and the ring learners. The intuition is to skip additional messages to make up for the time it takes for a skip message to arrive at the learners.

$$skips(t_{now}) = \lambda * (t_{now} - t_{ref} - avg\_delay) - skipped \tag{3.6}$$

3.4.4 Non-disruptive recovery

Recovering a failed learner in URingPaxos, as described before, boils down to (a) retrieving and installing the most recent service's checkpoint and (b) recovering and executing commands that are not included in the retrieved snapshot,

the *log tail*. While this procedure can be optimized in many ways [19], recovery in URingPaxos is inherently subject to a tradeoff that involves the frequency of checkpoints and the size of the log tail: frequent checkpoints result in small log tails and, conversely, infrequent checkpoints lead to large log tails.

Since checkpoints tend to slow down service execution, reducing the frequency of checkpoints seems desirable. However, restricting the log tail size is equally important because retrieving commands from the log during recovery has negative effects on the service's performance. This happens because acceptors must participate in new rounds of Paxos and at the same time retrieve values accepted in earlier rounds (i.e., the log tail). We have experimentally assessed that even under moderate load the recovery traffic drastically affects performance (see Section 3.7.6).

To minimize disruption of service performance during normal service execution and recovery of a learner, we revisited URingPaxos's original recovery mechanism. With the new method, a recovering learner starts by caching new ordered messages. This silent procedure does not place acceptors under additional stress. The replica then must retrieve a valid checkpointed state from another replica (or from remote storage), that is, a checkpoint that contains all commands that precede the cached commands. With a valid checkpoint, the replica can apply the cached commands not in the checkpoint and discard the ones already in the checkpoint. This procedure prioritizes performance during normal operation but it may increase the time needed to recover a learner.

3.5 Services

We have used two services, a key-value store and a distributed log, to illustrate the capabilities of URingPaxos. In this section we briefly discuss these services.

3.5.1 MRP-Store

MRP-Store implements a key-value store service where keys are strings and values are byte arrays of arbitrary size. The database is divided into l partitions $P_0, P_1, ..., P_l$ such that each partition P_i is responsible for a subset of keys in the key space. Applications can decide whether the data is hash- or range-partitioned [87], and clients must know the partitioning scheme. The service is accessed through primitives to read, update, insert, and delete an entry (see Table 3.1). Additionally we provide a range scan command to retrieve entries whose keys are within a given interval.

Table 3.1. MRP-Store operations.

Operation	Description
read(k)	return the value of entry k, if existent
scan(k, k')	return all entries within range $k..k'$
update(k, v)	update entry k with value v, if existent
insert(k, v)	insert tuple (k, v) in the database
delete(k)	delete entry k from the database

MRP-Store replicates each partition using the state machine replication approach [69], implemented with URingPaxos. A request to read, update, insert, or delete entry k is multicast to the partition where k belongs; a scan request is multicast to all partitions that may possibly store an entry within the provided range, if data is range-partitioned, or to all partitions, if data is hash-partitioned.

MRP-Store ensures sequential consistency [11], that is, there is a way to serialize client operations in any execution such that: (1) it respects the semantics of the objects, as determined in their sequential specifications and (2) it respects the order of non-overlapping operations submitted by the same client. Atomic multicast prevents cycles in the execution of multi-partitions operations, which would result in non-serializable executions.

3.5.2 DLog

DLog is a distributed shared log that allows multiple concurrent writers to append data to one or multiple logs atomically (see Table 3.2). Append and multi-append commands return the position of the log at which the data was stored. There are also commands to read from a position in a log and to trim a log at a certain position. Like MRP-Store, DLog uses state machine replication implemented with URingPaxos. Commands to append, read, and trim are multicast to the log they address and multi-append commands are multicast to all logs involved. A DLog server holds the most recent appends in-memory and can be configured to write data asynchronously or synchronously to disk.

3.6 Implementation

In this section, we discuss important aspects about the implementations of URingPaxos and the services we built on top of it.

Table 3.2. DLog operations.

Operation	Description
append(l, v)	append v to log l, return position p
multi-append($\mathscr{L}, v$)	append value v to logs in $\mathscr{L}$
read(l, p)	return value v at position p in log l
trim(l, p)	trim log l up to position p

3.6.1 URingPaxos

URingPaxos is implemented mostly in Java, with a few parts in C. All the processes in URingPaxos, independent of their roles, are multi-threaded. Threads communicate through Java's standard queues. A learner has dedicated threads per each ring it subscribes to. Another thread then deterministically merges the queues of these threads. Acceptors, when using in-memory storage, have access to pre-allocated buffers with 15k slots, each slot of size 32 kbytes. This allows the acceptors to handle re-transmission during approximately 3 seconds of execution time under the most strenuous conditions. Disk writes are implemented using the Java version of Berkeley DB. All communication within URingPaxos is based on TCP. Automatic ring management and configuration management is handled by Zookeeper. Applications can use URingPaxos by including it as a library or by running it standalone. In standalone mode, applications can communicate using a Thrift API.[3] URingPaxos is publicly available for download.[4]

3.6.2 MRP-Store

In our prototype, clients connect to proposers through Thrift and replicas implement the learner interface. The partitioning schema is stored in Zookeeper and accessible to all processes. Clients determine an entry's location using the partitioning information and send the command to a proposer of the corresponding ring. Clients may batch small commands, grouped by partition, up to 32 kbytes. Replicas reply to clients with the response of a command using UDP. There are multiple client threads per client node and each one only submits a new request after the first response from a replica in single-partition commands or for at least one response from every partition in scan operations is received.

Database entries are stored in an in-memory tree at every replica. Replicas comply with URingPaxos's recovery strategy (see Section 3.4.2) by periodically

[3] http://thrift.apache.org/

[4] https://github.com/sambenz/URingPaxos

taking checkpoints of the in-memory structure and writing them synchronously to disk. After a majority of replicas have written their state to stable storage, Paxos acceptors are allowed to trim their logs. A recovering replica will contact a majority of other replicas and download the most recent remote checkpoint.

3.6.3 DLog

Similarly to MRP-Store, DLog clients submit commands to replicas using Thrift. Multiple commands from one client can be grouped in batches of up to 32 kbytes. Replicas implement the learner's interface to deliver commands. Replicas append the most recent writes to an in-memory cache of 200 Mbytes and write all data asynchronously to disk. Results from the execution of commands are sent back to clients through UDP. A trim command flushes the cache up to the trim position and creates a new log file on disk.

3.7 Experimental evaluation

In this section, we experimentally assess various aspects of the performance of our proposed systems:

- We establish a baseline performance for URingPaxos, MRP-Store, and DLog.
- We measure vertical and horizontal scalability of MRP-Store and DLog in a datacenter and across datacenters.
- We evaluate the impact of recovery on performance.

Additionally, we assess the behavior of URingPaxos under a range of "extreme" conditions, including wide-area channels and high-performance links. Since we do not have access to an experimental environment that simultaneously accommodates all these characteristics, we conducted our experiments in different environments, as described next.

- We scale the number of rings to achieve high performance in a high-end 10 Gbps network.
- We stress URingPaxos skip mechanism with highly skewed traffic.
- We assess the impact of a global ring and a disaster failure in a geographically distributed deployment.

- We evaluate the impact of a recovering replica on the performance of operational replicas under peak load.

3.7.1 Hardware setup

The local-area network experiments (i.e., within a datacenter) were performed in two environments: (a) A cluster of 4 servers equipped with 32-core 2.6 GHz Xeon CPUs and 128 GB of main memory. These servers were interconnected through a 48-port 10-Gbps switch with round trip time of 0.1 millisecond. For persistency we use solid-state disks (SSDs) with 240 GB and 5 7200-RPM hard disks with 4 TB each. (b) A cluster of 24 Dell PowerEdge 1435 servers and 40 HP SE1102 servers connected through two HP ProCurve 2910 switches with 1-Gbps interfaces. The globally distributed experiments (i.e., across datacenters) were performed on Amazon EC2 with instances in 5 different regions. We used r3.large spot-instances, with 2 vCPU and 15 GB DRAM. To avoid disk bottlenecks, all experiments were executed with in-memory storage.

3.7.2 URingPaxos configuration

URingPaxos has three configuration parameters [81]: M, λ and Δ_t. M is the number of messages delivered (or skipped) contiguously from the same single ring; if not stated otherwise, we use $M = 1$.

We have empirically determined that λ, the virtually maximum throughput of a ring, should be set a bit higher than the actual maximum achievable performance. Too high λ values lead to wasted CPU cycles in the deterministic merge function; too low λ values cap performance.

Parameter Δ_t determines how often skip messages are proposed in a Paxos instance. In general, small values for Δ_t are preferred, to reduce the latency of actual messages; too low Δ_t values, however, waste Paxos instances and introduce additional overhead in the system.

3.7.3 Experimental setup

Within a datacenter, URingPaxos was initialized as follows: M =1, $\Delta = 5$ millisecond, and $\lambda = 9000$. Across datacenters, the following configuration was used: M =1, $\Delta = 20$ millisecond, and $\lambda = 2000$. We keep machines approximately synchronized by running the NTP service before the experiments. We used Berkeley DB version JE 5.0.58 as persistent storage. Unless stated otherwise, acceptors use asynchronous disk writes. When in synchronous mode,

batching was disabled, that is, instances were written to disk one by one. Each experiment is performed for a duration of at least 100 seconds.

3.7.4 Baseline performance

In this section, we evaluate the performance of a single multicast group in URing-Paxos with a "dummy service" (i.e., commands do not execute any operations) under varying request sizes and storage modes. We also compare the performance of MRP-Store and DLog to existing services with similar functionality.

URingPaxos

Setup. In this experiment there is one ring with three processes, all of which are proposers, acceptors, and learners, and one of the acceptors is the coordinator. Proposers have 10 threads, each one submitting requests whose size varies between 512 bytes and 32 kbytes. Batching is disabled in the ring. We consider five different storage modes: in-memory, synchronous and asynchronous disk writes using solid-state disks and hard disks.

Results. As seen in the top-left graph of Figure 3.3, regardless the storage mode, throughput increases as the request size increases. With synchronous disk writes, the throughput is limited by the disk's performance. With in-memory storage mode, the throughput is limited by the coordinator's CPU (bottom-left graph). The coordinator's CPU usage is the highest in asynchronous mode. This is due to Java's parallel garbage collection (e.g., 200% CPU). For in-memory storage, we allocate memory outside of Java's heap and therefore performance is not affected by Java's garbage collection. The bottom-right graph of Figure 3.3 shows the CDF of latency for 32 kbyte values. In synchronous disk write mode, more than 90% of requests take less than 10 milliseconds.

MRP-Store

Setup. In this experiment, we use Yahoo! Cloud Serving Benchmark (YCSB) [35] to compare the performance of MRP-Store against Apache's Cassandra and a single MySQL instance. These systems provide different consistency guarantees, and by comparing them we can highlight the performance implications of each guarantee. In the experiments with MRP-Store, we use three partitions, where participants in a partition subscribe to a ring local to the partition. Each ring is deployed with three acceptors, all of which write asynchronously to disk. We test

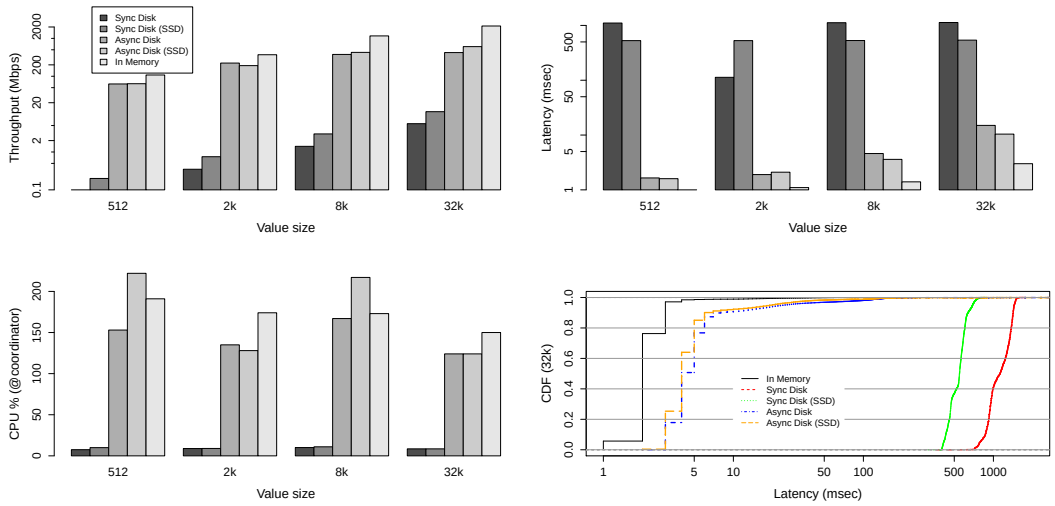

Figure 3.3. URingPaxos with different storage modes and request sizes. Four metrics are measured: throughput in mega bits per second (top-left graph), average latency in milliseconds (top-right graph), CPU utilization at coordinator (bottom-left graph), and CDF for the latency when requests are 32 KBytes (bottom-right graph). The y-axis for throughput and latency is in log scale.

configurations of MRP-Store where replicas in the partitions subscribe to a common global ring and where there is no global ring coordinating the replicas (in the graph, "independent rings"). All the rings are co-located on three machines and clients run on a separate machine. In the experiments with Cassandra, we initiate three partitions with replication factor three. MySQL is deployed on a single server. In all cases, the database is initialized with 1 GByte of data.

Results. With the exception of Workload E, composed of 95% of small range scans and 5% of inserts, Cassandra is consistently more efficient than the other systems since it does not impose any ordering on requests (see Figure 3.4). Ordering requests within partitions only (i.e., independent rings) is cheaper than ordering requests within and across the system. This happens because with independent rings, each ring can proceed at its own pace, regardless the load in the other rings. To a certain extent, this can be understood as the cost of ensuring stronger levels of consistency. In our settings, MRP-Store compares similarly to MySQL. As we show in the following sections, MRP-Store can scale with additional partitions while keeping the same ordering guarantees, something that is not possible with MySQL.

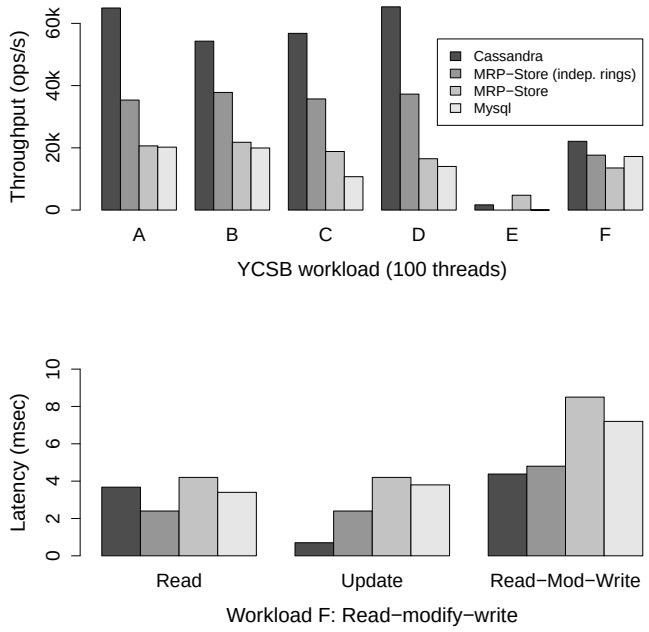

Figure 3.4. Performance of Apache's Cassandra, two configurations of MRP-Store, and MySQL, under Yahoo! cloud serving benchmark (YCSB). The graphs show throughput in operations per second (top) and average latency in msecs (bottom).

DLog

Setup. In this experiment, we compare the performance of our DLog service to Apache's Bookkeeper [63]. Both systems implement a distributed log with strong consistency guaranties. All requests are written to disk synchronously. The DLog service uses two rings with three acceptors per ring. DLog learners subscribe to both rings and are co-located with the acceptors. Bookkeeper uses an ensemble of the same three nodes. A multithreaded client runs on a different machine and sends append requests of 1 KBytes.

Results. Figure 3.5 compares the performance of our DLog service with Apache Bookkeeper. The DLog service consistently outperforms Bookkeeper, both in terms of higher throughput and lower latency. With 200 clients, DLog approaches the limits of the disk to perform writes synchronously. The large latency in Bookkeeper is explained by its aggressive batching mechanism, which attempts to maximize disk use by writing in large chunks.

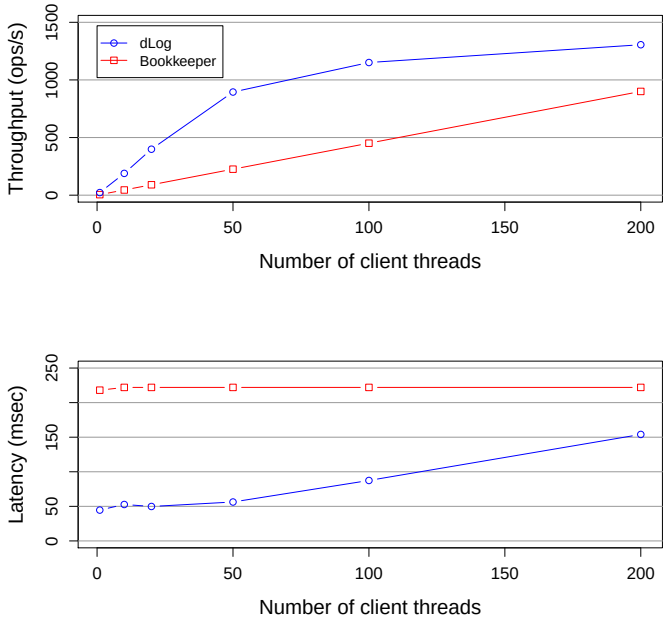

Figure 3.5. Performance of DLog and Apache's Bookkeeper. The workload is composed of 1 kbyte append requests. The graphs show throughput in operations per second (top) and average latency in msecs (bottom).

3.7.5 Scalability

In this section, we perform a set of experiments to assess the scalability of our proposed services. We consider vertical scalability with DLog and URingPaxos (i.e., variations in performance when increasing the resources per machine in a static set of machines) and horizontal scalability with MRP-Store (i.e., variations in performance when increasing the number of machines).

Vertical scalability of disk writes

Setup. In this experiment, we evaluate vertical scalability with the DLog service by varying the number of multicast groups (rings). Each multicast group (ring) is composed of three processes, one of which assumes the learner's role only and the others are both acceptors and proposers. We perform experiments with up to 5 disks per acceptor, where each ring is associated with a different disk. Therefore, by increasing the number of rings, we add additional resources to the acceptors. In each experiment, learners subscribe to k rings and to a common ring shared by all learners, where k varies according to the number of disks used in the experiment. Processes in the rings are co-located on three physical machines. Clients are located on a separate machine and generate 1 KByte requests, which are batched into 32 KByte packets by a proxy before being submitted to URingPaxos. The workload is composed of append requests only. Throughput is shown per ring. The reported latency is the average over all the rings.

Results. Figure 3.6 shows the throughput and latency of URingPaxos as the number of rings increases. Throughput improves steadily with the number of rings. The percentages show the linear scalability relative to the previous values. The latency CDF corresponds to the reported throughput for writes to disk 1.

Vertical scalability in a local 10 Gbps network

In this section, we evaluate the vertical scalability of URingPaxos in a local 10 Gbps network environment.

Setup. We perform two sets of experiments, one with 200-byte messages and another with 32-kbyte messages. For each message size, we increase the number of rings from 1 (i.e., Ring Paxos) up to 10. Four servers are involved: one server runs one proposer and one acceptor per ring, two other servers play the role of acceptors only, with one acceptor deployed per ring; the last server runs a learner, which subscribes to up to 10 rings. The proposer in each ring uses multiple threads (20), one thread per client. We report peak throughput, measured at the learner.

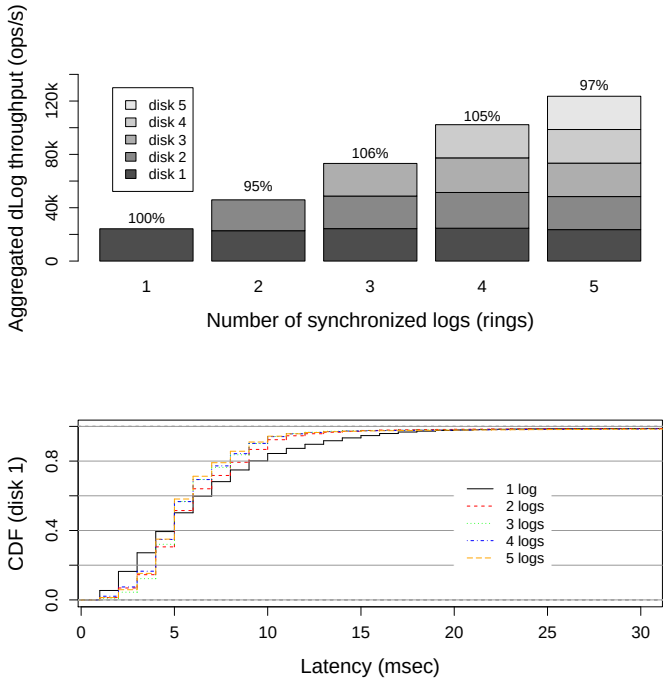

Figure 3.6. Vertical scalability of DLog in asynchronous mode. The graphs show aggregate throughput in operations per second (top), and latency CDF in msecs (bottom).

Results. Figure 3.7 on the left shows that URingPaxos reaches peak performance with 9 rings for large messages and with 8 rings for small messages. With large messages, URingPaxos reaches 8.41 Gbps, very close to 8.75 Gbps, the maximum usable TCP throughput (i.e., without TCP/IP headers) we could produce with *iperf*.[5] With small messages, URingPaxos achieves about 570 K messages per second. We also report the latency CDF, measured in 1-millisecond buckets, for the peak load (center graphs) and the CPU consumption at the learner (right graphs). The 90-th latency percentile under these conditions is below 5 milliseconds. The protocol is network-bound with large messages and CPU-bound with small messages. (Since there is one communication thread per ring at the learner, 10 rings can use up to 1000% CPU.)

In both experiments we can see on the left that as the number of rings a learner subscribes to increases, the throughput achieved by each ring decreases. This happens because the load in the learner's Java virtual machine increases

[5]http://iperf.sourceforge.net/

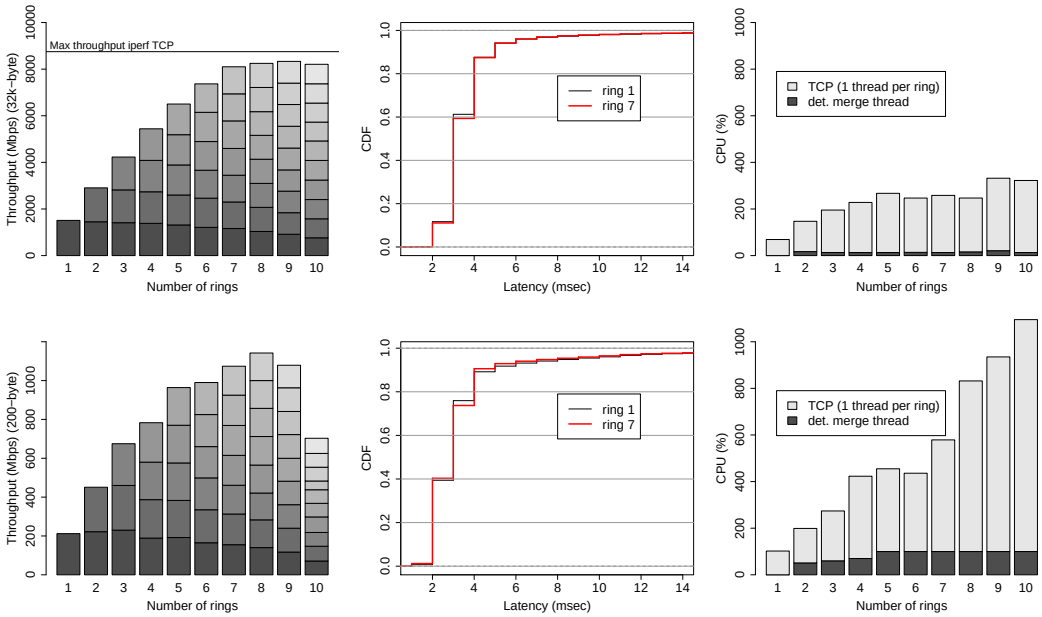

Figure 3.7. Scaling up URingPaxos in a 10 Gbps network. The graphs show the aggregate and per ring throughput in megabits per second for 32-kbyte (left top) and 200-byte (left bottom) messages; the latency CDF, measured in 1-millisecond buckets (center); the CPU usage (right). All measurements performed at the learner process.

with each new ring, slowing down the learner. In URingPaxos, a slow process reduces the overall traffic, as a result of flow control.

Horizontal scalability across the globe

Setup. In this experiment, we evaluate horizontal scalability with the MRP-Store service, globally deployed across four Amazon EC2 regions (one in eu-west, two in us-west, and one in us-east). In each region there is one ring composed of a replica with three proposers/acceptors, and one client running on a separate machine. Replicas from all the rings are also part of a global ring. Clients send 1 KByte commands to their local partitions (rings) only. Each client machine batches the requests into packets of 32 kbytes before sending them. The workload is composed of update requests only. Latency is measured in the us-west-2 region.

Results. Similarly to the DLog service, throughput increases as new parti-

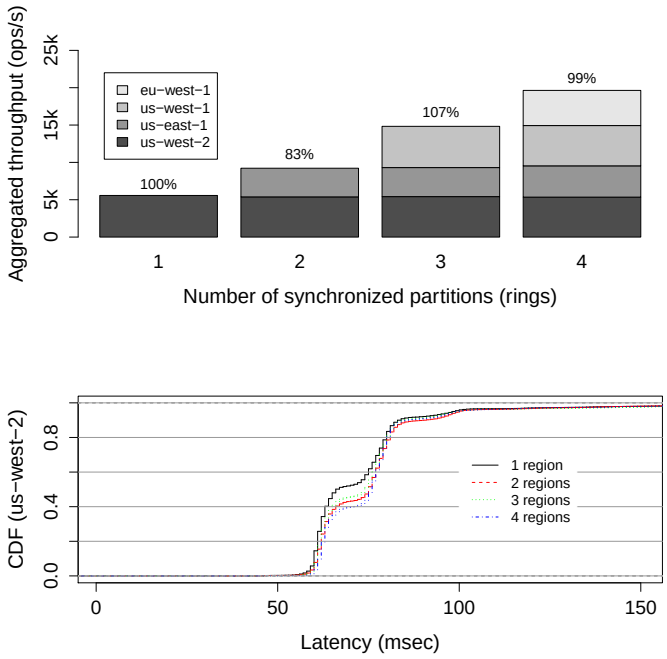

Figure 3.8. Horizontal scalability of MRP-Store in asynchronous mode. The graphs show aggregate throughput in operations per second (top) and latency CDF in msecs in us-west-1 (bottom).

tions are added to the collection (see Figure 3.8). As expected, latency is almost constant with the number of rings. We note that the local throughput of a region is not influenced by other regions, the reason for the scalability of the service. The percentages show the linear scalability relative to the previous values.

Data center fault tolerance

In this section, we evaluate the global scalability and fault tolerance of URing-Paxos. The goal is to show that having a large global ring, which allows to send ordered commands to geographically distributed partitions (local rings), does not slow down local traffic. We also evaluate the effect of a data center outage during runtime.

Setup. For this experiment, we used Amazon EC2 instances. We deployed 5 local rings, each in its own region: us-west-1 (N. California), us-west-2 (Oregon), eu-west-1 (Ireland), ap-southeast-1 (Singapore), ap-southeast-2 (Sydney). All nodes in each local ring are placed on the same availability zone. We also de-

ployed a global ring, composed of three acceptors (placed in separate regions) and all learners from each of the local rings. This deployment allows for progress even in the presence of a disaster taking down an entire datacenter. We simulated a datacenter outage by forcibly killing all processes belonging to one of the regions containing an acceptor of the global ring.

Results. We first evaluate the fault tolerance of URingPaxos. Figure 3.9 shows the throughput in each of the local rings, using messages of 32 kbytes. We can see that, despite the outage of a complete region (at around 25 seconds into the execution), the remaining rings maintain normal traffic after a short disruption caused by the global ring reconfiguration.

To assess the impact of a global ring on the performance of local rings, we conducted a few other experiments using the same deployment of 5 datacenters, each with a local ring. We consider a baseline case with local rings only (i.e., no global ring) and setups with a global ring synchronizing all nodes, with and without latency compensation (Section 3.4.3). We use the same load (number of clients) in all three cases, roughly 80% of the peak throughput for the case with compensation enabled, with 200-byte messages. Figure 3.10 shows the throughput obtained in each case and the latency CDF. The local throughput went down by around 23% with a global ring connecting all the nodes. The results also show that compensating the latency difference between rings is fundamental. The "steps" visible in the latency CDF for the scenario with no compensation reflect the latency difference across rings.

URingPaxos ring scalability

In URingPaxos, learners can subscribe to any combination of existing rings. Imbalanced traffic across rings is compensated with the skip mechanism. In this experiment, we assess the overhead of the skip mechanism on highly skewed traffic.

Setup. This experiment was conducted in a local cluster with a 1 Gbps network. In this experiment, a single learner subscribes to multiple rings. Each ring is composed of three acceptors and the learner. In order to assess the protocol's inherent latency without any queuing effects, we consider executions with a single client. We varied the number of rings from 1 up to 32. Except for the configurations with 16 and 32 rings, we deploy one acceptor per node. For the experiments with 16 rings, there are two acceptors per node; with 32 rings, there are four acceptors per node. To assess the efficacy of the skip mechanism, the client submits 200-byte messages to one of the rings; the other rings rely solely on the skip mechanism. In these experiments, Δ was set to 5 milliseconds.

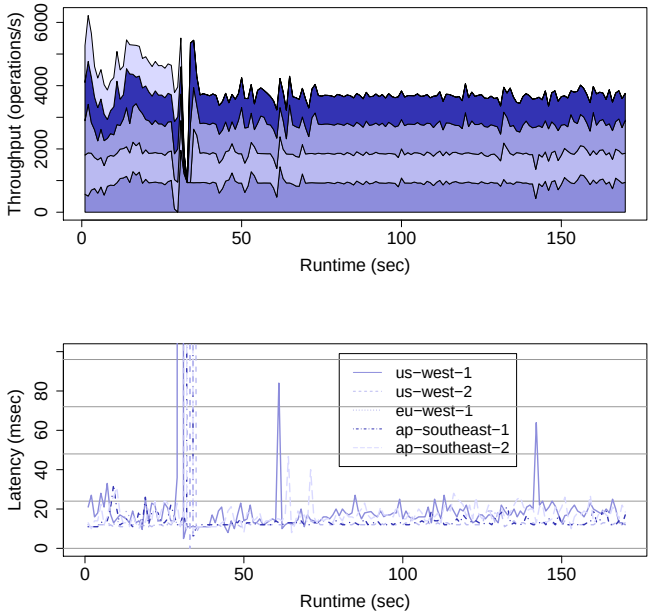

Figure 3.9. Impact of a data center outage after 25s into the execution in the performance of a global URingPaxos deployment.

Results. The most visible impact in Figure 3.11 is the transition from one to two rings. One ring is not constrained by any synchronization and can achieve the lowest latency. Additional rings introduce an overhead, that eventually increases linearly with the number of rings. Since we have one client only, from Little's law [61], the throughput is the inverse of the latency.

3.7.6 Recovery

Impact of recovery on performance

In this section, we evaluate the impact of failure recovery on the system's performance using the MRP-Store service.

Setup. We deploy one ring with three acceptors, all performing asynchronous disk writes, and three replicas in the local cluster. The system operates at 75% of its peak load and there is one client generating requests against the replicas. The replicas periodically checkpoint their in-memory data store synchronously to disk to allow the acceptors to trim their log. One replica is terminated after 20

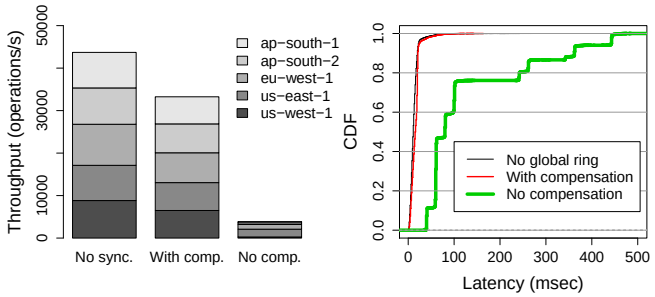

Figure 3.10. Impact of a global ring to local maximum throughput with and without latency compensated skip calculation.

seconds and restarts after 240 seconds, at which point it retrieves the most recent checkpoint from an operational replica. The instances that are not included in the checkpoint will be retrieved directly from the acceptors.

Results. Figure 3.12 shows the impact of recovery on performance. As seen in the graph, re-starting a terminated replica causes a short reduction in performance. Writing checkpoints synchronously to the disk does not disrupt the service. We note that checkpoints are not written to disk at the same time by all the replicas and that the client waits only for the first response form any replica. Performance is mostly affected by trimming the acceptor logs and also when the recovering replica retrieves and installs a checkpoint.

Non-disruptive recovery under peak load

We use MRP-Store to evaluate our optimized recovery procedure. Our key-value store service implements commands to insert and remove tuples of arbitrary size, read and update an existing entry, and query a range of tuples. Replicas use a copy-on-write data structure to allow checkpoints in parallel with the execution of commands.

Setup. The experimental setup uses a ring with 3 nodes, each acting as an acceptor and a learner (i.e., replica). Four clients (each with 150 threads) submit 1024-byte update requests to the replicas through YCSB [35]. Each replica executes every request and replies back to the client using UDP. Every replica periodically checkpoints its state into a distributed file system,[6] accessible to all replicas. The state checkpointed by a replica has 1.5 million entries (1.5 gbyte).

[6]http://www.xtreemfs.org/

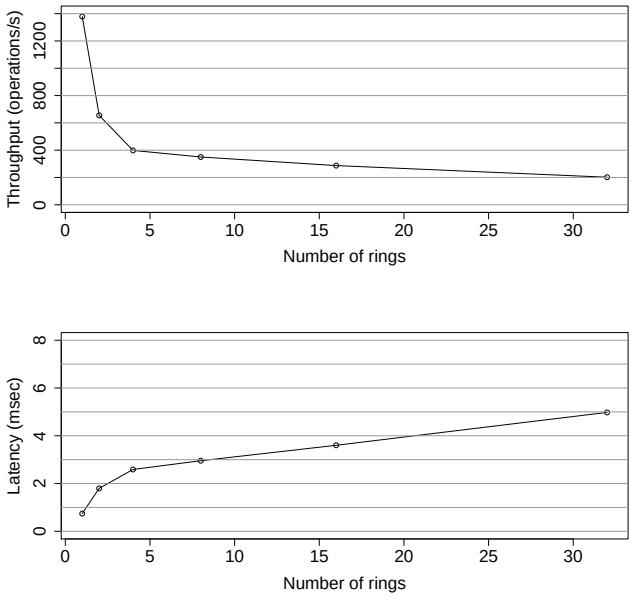

Figure 3.11. Impact of the number of groups (rings) a learner subscribes to on throughput and latency (since there is a single client, from Little's law throughput is the inverse of latency).

Results. Figure 3.13 shows the behavior of URingPaxos's new non-disruptive recovery under maximum load, which for 1024-byte values is around 800 Mbps. For comparison, we also depict the behavior of the classic recovery protocol under lower load, around 400 Mbps, since the classic protocol cannot sustain higher load. Around 45 seconds into the execution, we crash one of the replicas, which starts recovery around time 110. With the new recovery protocol, the average throughput during recovery is 78% of the throughput under normal operation. Performance troughs are due to garbage collection (events labelled "1" in the graph) and ring management (event with label "2"). Since processes communicate in a ring, a pause in any of the nodes (e.g., due to garbage collection) can have a visible effect on throughput. The fact that the recovering learner has to batch new commands and that replicas have to use multiple (in-memory) copy-on-write data structures forces us to use large heaps, which lead to longer and unpredictable garbage collection pauses.

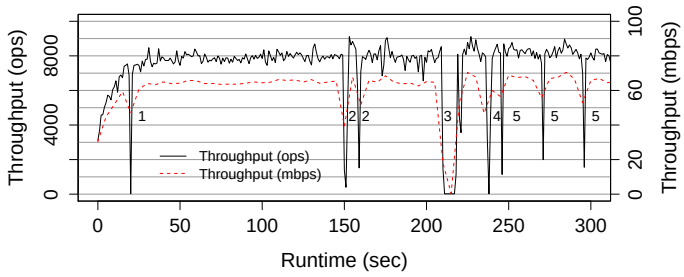

Figure 3.12. Impact of recovery on performance (1: one replica is terminated; 2: replica checkpoint; 3: acceptor log trimming; 4: replica recovery; 5: re-proposals due to recovery traffic).

3.8 Related work

In this section, we review related work on atomic multicast, geo-replication, distributed logging, and recovery.

Atomic multicast. The first atomic multicast protocol can be traced back to [23], where an algorithm was devised for failure-free scenarios. To decide on the final timestamp of a message, each process in the set of message addressees locally chooses a timestamp, exchanges its chosen timestamps, deterministically agrees on one of them, and delivers messages according to the message's final timestamp. As only the destinations of a message are involved in finalizing the message's timestamp, this algorithm is scalable. Moreover, several works have extended this algorithm to tolerate failures [50, 56, 95, 97], where the main idea is to replace failure-prone processes by fault-tolerant disjoint groups of processes, each group implementing the algorithm by means of state machine replication. The algorithm in [42] proposes to daisy-chain the set of destination groups of a message according to the unique group ids. The first group runs consensus to decide on the delivery of the message and then hands it over to the next group, and so on. Thus, the latency of a message depends on the number of destination groups.

While most works on multicast algorithms have a theoretical focus, Spread [9] implements a highly configurable group communication system, which supports the abstraction of process groups. Spread orders messages by means of interconnected daemons that handle the communication in the system. Processes connect to a daemon to multicast and deliver messages. To the best of our knowledge,

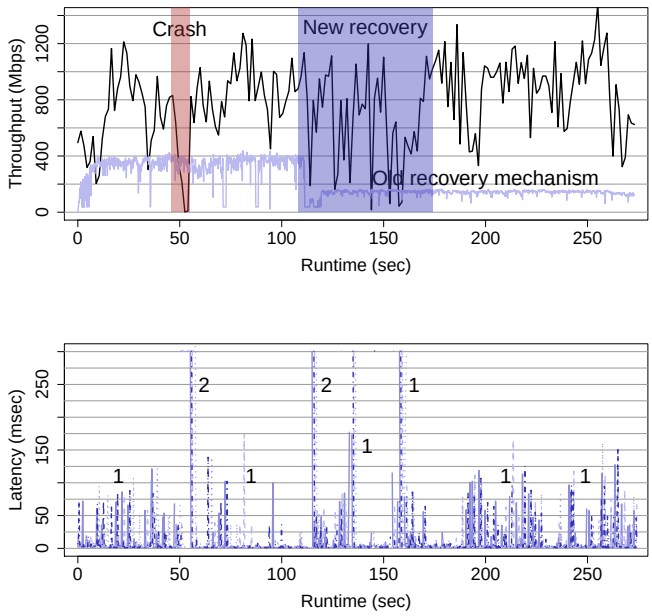

Figure 3.13. Recovery of a key-value store snapshot with 1.5 million entries. Throughput of URingPaxos's new and old recovery protocols (top) and latency of new recovery protocol (bottom, where "1" identifies garbage collection events and "2" identifies ring management events).

URingPaxos is the first high-performance atomic multicast library available for download. Similarly to Mencius [77], coordinators in URingPaxos account for load imbalances by proposing null values in consensus instances. Differently from Mencius, which is an atomic broadcast protocol, URingPaxos implements atomic multicast by means of the abstraction of groups. While the group abstraction is similar to the Totem Multi-Ring protocol [1], Totem uses timestamps to achieve global total order. URingPaxos's deterministic merge strategy is similar to the work proposed in [7], which totally orders message streams in a widely distributed publish-subscribe system.

Geo-replication There are different approaches to handling the high latency inherent of globally distributed systems. Some systems choose to weaken consistency guarantees (e.g., Dynamo [40]), while others cope with wide-area round trip times. Mencius [77], WHEAT [104] and EPaxos [85] are latency optimized. Both protocols implement atomic broadcast and therefore do not scale. P-store [98]

relies on atomic multicast. In order to scale, it partitions the service state and strives to order requests that depend on each other, imposing a partial order on requests. Sinfonia [6] and S-DUR [101] build a partial order by using a two-phase commit-like protocol to guarantee that requests spanning common partitions are processed in the same order at each partition. Spanner [36] orders requests within partitions using Paxos and across partitions using a protocol that computes a request's final timestamp from temporary timestamps proposed by the involved partitions.

Chain replication. Conceptually, chain replication [110] looks similar to URing-Paxos. All processes are organized in an overlay. The two replication techniques however, differ significantly. In chain replication, all write requests must be sent to the first replica in the chain and all read request must be sent to the last process in the chain. This does not apply to URingPaxos. All commands can be sent to any processes in the ring, since the order property is achieved by the Paxos protocol and not the position in the ring. Further, URingPaxos can recover naturaly from failures (e.g., lost messages), while chain replication requires an external oracle in case of a process failure. Recovering in chain replication is similar to virtual synchrony, where the view of the system is changed to exclude a faulty replica.

Distributed logging. Atomic broadcast is not the only solution to totally order requests in a distributed environment. Distributed logging is an alternative approach, where appending a log entry corresponds to executing a consensus instance in an atomic broadcast protocol. CORFU [76] implements a distributed log with a cluster of network-connected flash devices, where the log entries are partitioned among the flash units. Each log entry is then made fault-tolerant using chain replication and a set of flash devices. New data is always appended to the end of the distributed log. To append a message, a client of CORFU (e.g., application server) retrieves and reserves the current tail of the distributed log through a sequencer node. Although appends are directly applied to the flash devices, the scalability of retrieving the log's next available offset is determined by the centralized sequencer's capacity. In our DLog service, the increasing append load is smoothly absorbed by adding new rings to the ensemble, and is not subject to central components. Disk Paxos [51] is another implementation of a distributed log that does not rely on a sequencer. However, Disk Paxos is not network efficient since appending new data clients leads to contention over the log entries. An advantage to CORFU and similar systems [58] is that the distribution of appends among the storage units can be balanced. Tango [13], builds on CORFU to implement partitioned services, where a collection of log entries is

allocated to each partition. The replicas at each partition only execute the subset of the log entries corresponding to their partitions, and skip the rest. Globally ordering the entire set of log entries simplifies ensuring consistency with cross partition queries. However, the number of partitions a service can be divided into is limited by the log's capacity at handling the appends. In our DLog service, an unbounded number of partitions can be created by adding new rings; moreover, queries concerning disjoint partitions are not globally ordered.

Recovery. Recovery protocols often negatively affect a system's performance. Several optimizations can be applied to the logging, checkpointing, and state transfer to minimize the overhead of recovery as we discuss next.

Optimized logging. A common approach to efficient logging is to log requests in batches [19, 26, 33, 49, 103]. Since stable storage devices are often block-based it is more efficient to write a batch of requests into one block rather than to write multiple requests on many different blocks. Another optimization is to parallelize the logging of batches [19]. Parallel logging benefits most applications in which the time for processing a batch of requests is higher than the time required for logging a batch. The overhead of logging can be further reduced by using solid-state disks (SSD) or raw flash devices instead of magnetic disks [94]. Similarly, in our DLog service we support both hard disks and SSDs, and synchronous and asynchronous disk writes to enable batched flushes to the disk.

Optimized checkpointing. Checkpoints are often produced during the normal operation of a system, while processing of the requests is halted [26, 69, 94, 103]. Not handling requests during these periods makes the system unavailable to clients and reduces performance. If instead processes take checkpoints at non-overlapping intervals, there will always be operational processes that can continually serve the clients. Building on this idea, in [19] processes schedule their checkpoints at different intervals. As the operation of a quorum of processes is sufficient for the system to make progress, a minority of processes can perform checkpointing while the others continue to operate. Another optimization is to use a *helper* process to take checkpoints asynchronously [32]. In this scheme, two threads, primary and the helper, execute concurrently. While the primary processes requests, the helper takes checkpoints periodically. Similarly, in our DLog service replicas can take snapshots at different non-overlapping intervals.

Optimized state transfer. State transfer has its own implications on performance. During state transfer, a fraction of the source processes' resources (e.g., CPU, network) are devoted to the transmission of the state, which is not to the advantage of performance. To protect performance, state transfer can be delayed to a moment in which the demand on the system is low enough that both

the execution of new requests and the transfer of the state can be handled [60]. Another optimization is to reduce the amount of transferred information. Representing the state through efficient data structures [26], using incremental checkpoints [27, 32], or compressing the state are among these techniques. In [19], the authors propose a collaborative state transfer protocol to evenly distribute the transfer load across replicas.

Chapter 4

Dynamic Atomic Multicast

4.1 Introduction

Today's on-demand computing resources, common in public cloud environments, provide operators of distributed systems with the possibility to react quickly to changes in application workload. Starting up new webservers once increased traffic is detected or switching off low utilized servers to save costs are common operations. Dynamically adding or removing resources when servers are stateful (e.g., databases), however, is much more challenging than reconfiguring stateless servers (e.g., webservers). In fact, building fault-tolerant (replicated) distributed services that provide strong consistency and scalable performance is a daunting task in itself. Further requiring these services to dynamically scale up and down resources introduces additional complexity.

Services are typically made scalable and fault-tolerant by means of state partitioning (sharding) and replication (e.g., [36, 6, 90]). But handling sharded and replicated data in a distributed environment is challenging if services are not willing to give up strong consistency. Strong consistency requires client requests to be ordered across partitions and replicas. Atomic multicast is a communication abstraction that can help the design of scalable and highly available stateful services [21, 14] by consistently ordering requests. Therefore, much of the complexity involved in designing scalable and fault-tolerant services is encapsulated by atomic multicast.

Nevertheless, existing atomic multicast protocols are *static*, in that creating new multicast groups at run time is not supported. Consequently, replicas must subscribe to multicast groups at initialization, and subscriptions and unsubscriptions can only be changed by stopping all replicas, redefining the subscriptions, and restarting the system. This chapter presents Elastic Paxos, the first *dynamic*

atomic multicast protocol. Elastic Paxos allows replicas to dynamically subscribe to and unsubscribe from atomic multicast groups.

Dynamic subscriptions in Elastic Paxos should not be confused with dynamic reconfiguration. In dynamic reconfiguration (e.g., [23, 71, 72, 73, 74]), the goal is to change the set of participants of a system (e.g., group membership). Elastic Paxos seeks to allow replicas to dynamically change the multicast groups they subscribe to, while the membership of the system may remain constant. Interestingly, we show in this chapter that one can use dynamic subscriptions to reconfigure a system.

In brief, our dynamic atomic multicast protocol composes multiple sequences of Paxos [69, 39], where each sequence is referred to as an *atomic multicast stream*, to provide efficient message ordering. The protocol ensures that no two replicas order the same requests in different orders and allows to add and remove additional streams during run time. To illustrate the design of a scalable and highly available prototypical service, we consider a storage service (i.e., a key-value store). The storage is partitioned into disjoint partitions and each partition is replicated by a group of replicas. There is one atomic multicast stream per partition, which the replicas of the partition subscribe to, and one atomic multicast stream that is shared by all replicas. The storage service supports two types of operations: single-partition operations (i.e., get and put on a single key) and multi-partition operations (i.e., a consistent get range operation that returns all keys in a specified interval). Single-partition operations are multicast to the replicas of the partition that contain the accessed key; multi-partition operations are multicast to all replicas, using the shared atomic multicast stream.

This chapter makes the following contributions.

- We introduce Elastic Paxos, an atomic multicast protocol that supports dynamic subscriptions.
- We show how Elastic Paxos can be used to design strongly consistent, scalable and highly available dynamic services.
- We detail the implementation of our new protocol.
- We evaluate the performance of Elastic Paxos

4.2 Motivation

Atomic multicast is a suitable abstraction to build scalable distributed systems. But creating new groups during run time is not supported by existing atomic

multicast systems. In this section, we motivate and define dynamic atomic multicast and explain why we need a new protocol to implement dynamic atomic multicast.

Atomic multicast, as discussed in the previous chapters, relies on static subscriptions of replicas to groups (streams), that is, subscriptions are defined at initialization and can only be changed by stopping all processes, redefining the subscriptions, and restarting the system.

In today's cloud environments, adding resources to and removing resources from an operational system without shutting it down is a desirable feature [34]. Combining the benefits of atomic multicast and dynamic subscriptions at run time allows several practical use cases, as we describe next.

Vertical elasticity. Although atomic broadcast is typically implemented with a single message stream, it can be also implemented with multiple streams, as long as all processes subscribe to all streams. When implemented with a single stream, the performance of atomic broadcast will be typically limited by the performance of the coordinator (CPU) or the acceptors (disk write performance) of the stream. However, replicas can increase the throughput of atomic broadcast by dynamically subscribing to multiple streams.

In doing so, each stream contributes to the aggregated throughput of atomic broadcast. (Section 4.7.3)

Horizontal elasticity. Scaling out a key-value store service can be achieved by horizontally partitioning (sharding) the service state. Partitioned state introduces the problem of how to ensure consistency of cross-partition queries. Paxos and other atomic broadcast algorithms ensure total order of commands within one partition (e.g., *get* and *put* commands), consistent cross-partition operations (e.g., *getrange*) must be coordinated using additional mechanisms, such as two-phase commit and synchronized clocks (e.g., [36]). Atomic multicast offers an alternative by ordering both single-partition and cross-partition commands, as needed (i.e., partial order). If replicas can dynamically subscribe to a new stream (i.e., a new partition), then a replicated data store can be repartitioned without service interruption. (Section 4.7.4)

Reconfiguration. Reconfiguration means changing the set of processes in a distributed system. It is used, for example, to replace a failed server or a server whose disk is full. Reconfiguring a replicated state machine has been considered before (e.g., [23, 71, 72, 73, 74]). In general, existing solutions consist in stopping processes in the current configuration (i.e., the running state machine), redefining the set of processes in the new configuration, and re-starting

the processes in the new configuration [73].

In Paxos, the real challenge is reconfiguring the set of acceptors since these are the processes that store the state of Paxos (e.g., accepted values). Moreover, processes must know the set of acceptors of each consensus instance (i.e., system membership). Lamport [69] suggests to manage membership by making the set of acceptors part of the state of the system and handling membership changes as commands, which must also be ordered by consensus. Such a mechanism, however, prevents multiple consensus instances from executing concurrently, which limits performance [74].

Dynamic subscriptions offer an alternative approach to reconfiguring the acceptors in a single stream S_i. We first create a new stream S_i' with the new set of acceptors, then have the learners subscribe to S_i', and finally unsubscribe from S_i. Note that this approach does not impose any constraints on the intersection between S_i and S_i' (e.g., S_i and S_i' can be disjoint sets).

4.3 Dynamic Atomic Multicast

After arguing for dynamic subscriptions in atomic multicast, we extend the atomic multicast interface with two additional primitives: *subscribe_msg*(G,S) and *unsubscribe_msg*(G,S), which replicas in replication group G can use to subscribe to and unsubscribe from stream S. After replicas subscribe to stream S, they will eventually deliver messages multicast to S. Similarly, if replicas unsubscribe from S, they will eventually stop delivering messages multicast to S. In both cases, atomic multicast guarantees acyclic ordered delivery (see Section 2.5).

A simple (but incorrect) solution. Conceptually, one can easily reconfigure a replicated state machine (atomic broadcast) by proposing a special *new_conf*(C) command that starts a new configuration C [69, 23, 73]. The position of the decision of *new_conf*(C) in the sequence of consensus instances defines the new configuration. Consensus instances decided before the instance in which *new_conf*(C) is decided use the current configuration; instances that succeed the instance in which *new_conf*(C) is decided use the new configuration C. Could a similar simple approach be used to handle dynamic subscriptions in partially replicated state machines (atomic multicast)?

Consider a replica R_1 in replication group G_1 that currently subscribes to stream S_1 and wishes to subscribe to stream S_2.[1] To ensure that replicas in G_1

[1] Note that subscribing to the very first stream is trivial since it does not involve coordination with any other streams.

agree on the instance in S_1 and S_2 where the merge occurs, so that they can deliver messages from both streams without violating acyclic order, R_1 proposes a subscription request, denoted by $sub(G_1, S_2)$, in S_1 and in S_2. Once ordered in S_1, the subscription request will determine when messages from both streams will be merged. The subscription request in S_2 determines what messages in the new stream will start to be delivered by replicas in G_1 (see Figure 4.1). After the $sub(G_1, S_2)$ is ordered in both streams, the deterministic merge starts to deliver messages from each stream in round-robin, starting with S_1. Note that the subscription messages ordered at streams S_1 and S_2 are only used by replicas to agree on how to merge the streams; the subscription messages are not passed to the application.

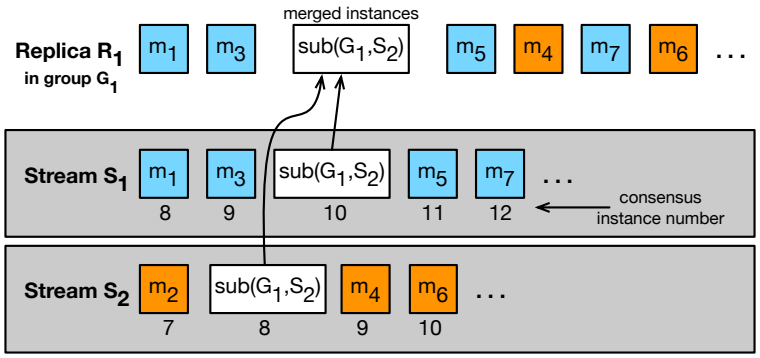

Figure 4.1. A simple scheme to dynamically subscribe to a stream.

Consider now a more complex case in which replica R_1 in G_1 initially subscribes to S_1 and wishes to subscribe to S_2, and R_2 in G_2 with a subscription to S_2 wishes to subscribe to S_1 (see Figure 4.2). This would be the case in our key-value store service, for example, if two partitions were merged as a single partition. To determine the instance in which the merge occurs, R_1 proposes message $sub(G_1, S_2)$ in streams S_1 and S_2, and R_2 proposes message $sub(G_2, S_1)$ in S_1 and S_2. But since subscription requests can be delivered in S_1 and S_2 in any order, it may happen that after both replicas subscribe to streams S_1 and S_2, the merged streams violate acyclic order of atomic multicast. In the example in Figure 4.2, R_1 orders m_6 before m_7, and R_2 orders m_7 before m_6.

The example above shows that simply having a deterministic scheme for replicas to merge two or more streams is not enough to ensure consistent order of messages. In the next section, we introduce a more sophisticated technique, which ensures that replicas in a group deliver the same sequence of messages, and replicas in the same group and in different groups do not violate acyclic order.

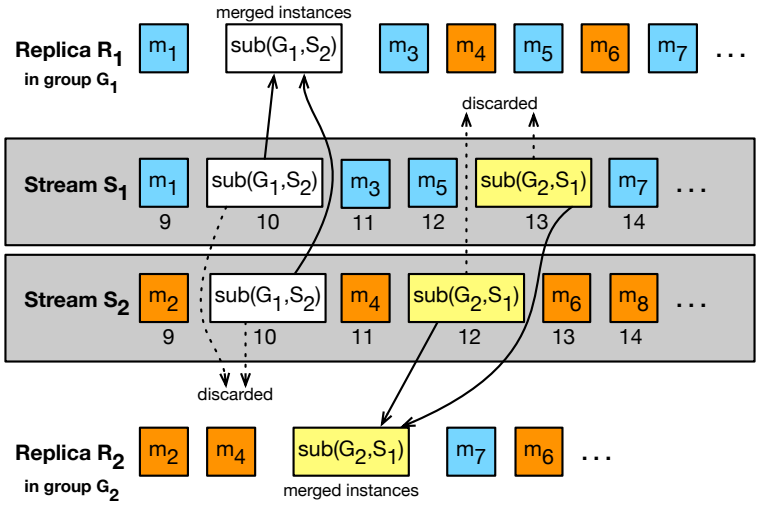

Figure 4.2. Example of order violation with simple scheme (i.e., m_6 and m_7).

4.4 Elastic Paxos

In this section, we present an overview of the Elastic Paxos protocol, describe Elastic Paxos in detail, introduce a few optimizations, and argue about the correctness of Elastic Paxos.

4.4.1 Overview

We seek decentralized solutions that properly coordinate dynamic subscriptions in atomic multicast without relying on a single entity, such as an oracle that oversees all subscribe and unsubscribe requests. In the following, we provide an overview of our solution. We describe how a replica R in replication group G can subscribe to and unsubscribe from a stream.

Every replica in G starts with a subscription to a default stream, S_G. In order for R to subscribe to a new stream S_N, R must atomically broadcast request *subscribe_msg*(G, S_N) to (a) the new stream S_N; and (b) a stream S that R currently subscribes to (e.g., the default stream). Upon delivering the subscription request from S, the deterministic merger that executes at R spawns a new learner task at R for stream S_N. The new learner starts by recovering all messages in S_N until it reaches the subscribe request *subscribe_msg*(G, S_N).

When the subscribe request is ordered in both streams S and S_N, the merger determines the "merge point", that is, the instance after which the replica will start combining messages from the new stream with messages from the cur-

rently subscribed streams. To avoid order violations, Elastic Paxos uses the *same instance* in both streams, computed as the maximum between the instances in which the subscribe request was delivered in each stream (see Figure 4.3). Intuitively, this works because the merge point is "aligned" at all subscribed streams.

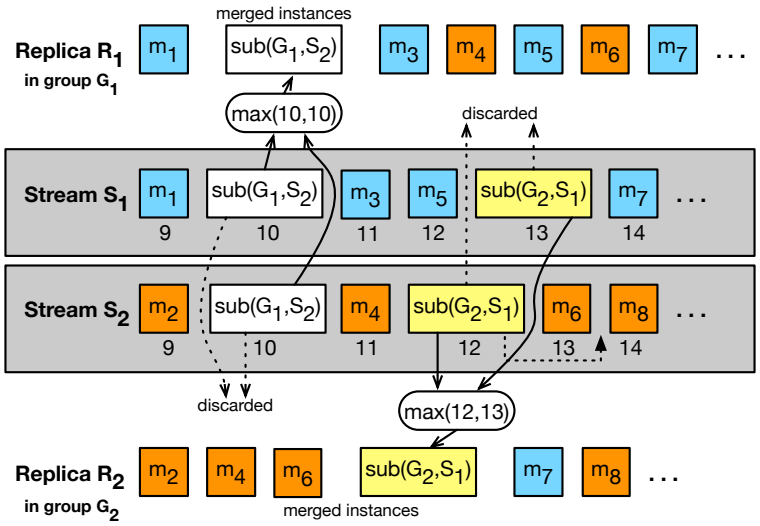

Figure 4.3. How Elastic Paxos ensures acyclic ordering.

Unsubscriptions are simpler than subscriptions because there is already a total order among messages in all subscribed streams. Therefore, it is enough to broadcast a single *unsubscribe_msg(group,stream)* request to any of the subscribed streams. As soon as the request is delivered, the dMerge task removes the requested stream from the set of streams the replica subscribes to.

4.4.2 Detailed protocol

Algorithm 1 details how a replica R in replication group G subscribes to a new stream S_N. Every replica consists of multiple tasks. There is one dMerge task, and one learner task per subscribed stream. The dMerge task orders messages from the various streams a replica subscribes to and handles subscription and unsubscription requests. dMerge holds an array of stream queues (Q), from which it deterministically (round-robin) delivers decided values. Every stream queue is filled by a background learner task. When a replica subscribes to a new stream, one more learner task is created. This new learner will recover (Section 4.6) all decided values and put them in Q.

Algorithm 1 Replica R in G subscribes to stream S_N

```
1:  Initialization:
2:    Q[1..max_stream][1..max_instance] ← ⊥, ⊥, ...
3:    start task dMerge                                   {init deterministic merge}
4:  task dMerge                                           {Deterministic merge}
5:    Initialization:
6:      Σ ← {S_G}                                         {set of subscribed streams, with default stream}
7:      start task Learner(S_G)                           {start the first learner}
8:      S ← S_G                                           {set first stream}
9:      ptr[S] ← 0                                        {next instance in a stream}
10:   while forever do                                    {round-robin delivery}
11:     ptr[S] ← ptr[S] + 1                               {set pointer to next message in S}
12:     wait until Q[S][ptr[S]] ≠ ⊥
13:       v ← Q[S][ptr[S]]
14:     if v = subscribe_msg(G_x, S_x) and G_x = G then
15:       S_N ← S_x
16:       start task Learner(S_N)
17:       while Q[S_N][ptr[S_N]] ≠ v do                   {find same subscribe...}
18:         ptr[S_N] ← ptr[S_N] + 1                       {...msg in both streams}
19:       merge_ptr ← max_ptr(ptr) + 1
20:       while ptr[S_N] < merge_ptr do                   {align stream}
21:         ptr[S_N] ← ptr[S_N] + 1                       {skip}
22:     else
23:       if v ≠ subscribe_msg(G_x, S_x) then
24:         deliver v                                     {v is ordered, pass it to the application}
25:     if ∀S ∈ Σ : ptr[S] = merge_ptr then
26:       Σ ← Σ ∪ {S_N}                                   {update current subscriptions}
27:       S ← first(Σ)                                    {after subscription start from first group}
28:     else
29:       S ← next(Σ)                                     {next group for round-robin delivery}
30: procedure max_ptr(ptr)
31:   // return maximum ptr[S] for all streams S in Σ
32:   x ← 0
33:   for each S ∈ Σ do
34:     if ptr[S] > x then x ← ptr[S]
35:   return x
36: procedure first(Σ)
37:   // return the first S in Σ
38: procedure next(Σ)
39:   // return the next (cyclic) S in Σ
40: task Learner(S)                                       {Learner of stream S}
41:   Initialization:
42:     ptr[S] ← 0
43:     for i from 1 to max decided instance in S do
44:       Q[S][i] ← recover(i)                            {recover all decided instances}
45:   upon deliver(v) do
46:     Q[S][i] ← v                                       {fill queue while Paxos instances get decided}
47:     i ← i + 1
```

For every queue, dMerge keeps a pointer per stream (ptr) with the position of the last ordered value in the stream that has already been delivered to the application. The subscription point is the maximum stream position of the two subscription messages (i.e., the new stream and the currently subscribed stream). Round-robin delivery from the new stream will start in the round after the maximum stream position.

For the sake of simplicity, in Algorithm 1 a stream position corresponds to a Paxos instance. In our prototype, the stream position is not related to the decided Paxos instances. Since multiple values or skip messages can be decided in one Paxos instance (batching), in our prototype the pointer refers to a value, after discarding skip messages.

4.4.3 Extensions and optimizations

A subscription request *subscribe_msg*(G, S_N) must be broadcast to S_N and to a currently subscribed stream S. Since this requires two invocations to atomic broadcast, it is possible that a process fails in between invocations, in which case the replica would block. To cope with such cases, if the dMerge of a replica does not deliver the subscription request from the second stream after some time, it broadcasts the request in the stream. Duplicated subscription requests are discarded by the replicas.

As Algorithm 1 shows, after receiving a subscribe request, the dMerge task interrupts the handling of messages until the same request is received in the new stream. Since the dMerge task does not know where in the stream the missing subscription request is, the simplest approach is to scan all previous messages. This procedure can be optimized if the process that triggers a subscription first broadcasts a hint to learners. Upon receiving such a hint (*prepare_msg*(G, S_N)), learners start scanning the new stream for subscription requests.

4.4.4 Correctness

Atomic multicast is a generalization of atomic broadcast and implements the abstraction of groups $\Gamma = S_1, ..., S_n$, also known as streams, where for each $S \in \Gamma, S \subseteq \Pi$. Processes may belong to one or more streams. If process $p \in S$, we say that p subscribes to stream S. Atomic multicast is defined by the primitives $multicast(S, m)$ and $deliver(m)$ (Section 2.5).

Elastic Paxos resembles URingPaxos in the absence of subscriptions and unsubscriptions. However, Elastic Paxos introduces the ability to add and remove subscriptions dynamically. Algorithm 1 describes how a replica R in replication

group G subscribes to a new stream S_N (i.e., by atomically broadcasting request $subscribe(G, S_N)$ to the new stream S_N and to R's default stream). In the following, we show that dynamic subscriptions do not violate any of the above specified properties of Atomic Multicast.

Proposition 1 *Validity. If a correct process multicasts a message m to S, then all correct processes in S will eventually deliver m.*

PROOF: It follows from the correctness of the Ring Paxos instance implementing S that m will be eventually in the decision of a consensus instance executed by all correct processes in S. Consequently, from an argument similar to that of uniform agreement, all such correct processes eventually deliver m. □

Proposition 2 *Uniform agreement: If a process delivers message m multicast to S_i, then all correct processes in S_i deliver m.*

PROOF: Assume p and q subscribe to S_i and q delivers m multicast to S_i . From the correctness of the Ring Paxos instance responsible for S_i, if p is correct, it will eventually decide on an instance that contains m. We claim that p will eventually deliver m. If p only subscribes to S_i, this is obviously true. Thus, assume that p also subscribes to stream S_j , where $j < i$. Process p will deliver m after having delivered M messages from each S_j. There could simply not be so many messages multicast to S_j. If so, the coordinator of the Ring Paxos instance responsible for S_j eventually times out and submits enough skip instances to reach the optimum in the interval. Thus, eventually enough application messages or skip messages will be decided to complete M, and eventually m is delivered by p. □

Proposition 3 *Uniform partial order. If processes p and q deliver messages m and m', then they deliver them in the same order.*

PROOF: Two cases must be considered:

(a) m and m' were multicast to the same stream S;

(b) m and m' were multicast to streams S_i and S_j, respectively, where $i < j$.

We assume that both processes p and q are correctly subscribed to S, respectively, S_i and S_j (Proposition 4).

Once p and q are subscribed to S, in case (a), it is simple to see from Algorithm 1 that both messages are delivered in the same order by all processes. Partial

order also holds for case (b), where processes p and q both are subscribed to both streams S_i and S_j. This, from the fact that processes order streams in the same way and first deliver M (round-robin messages, e.g. $M = 1$) messages from one stream and then deliver M from the other. Assume m is delivered in consensus instance k_i and m' in consensus instance k_j. If $k_i \leq k_j$, then both p and q will deliver m first and then m'. If $k_i > k_j$, then both processes will deliver m' first and then m. □

Proposition 4 *Dynamic subscriptions ensures acyclic ordering. If processes p and q subscribe both to streams S_i and S_j, then they deliver eventually the same suffix of messages.*

PROOF: By propositions 1 and 2, every process p that belongs to the same replication group G will eventually receive a subscription message in the old stream S_i and one in the new stream S_j. Subscription messages are proposed like any other messages and therefore decided in a consensus instance; k_i in S_i and k_j in S_j. The maximum instance number of k_i and k_j defines the subscription point α for all p in G, therefore, every replica in G will start delivering messages after k_α from the new stream S_j.

If two processes p and q, belonging to G and G', concurrently subscribe to S_i and S_j, that is, p subscribes to S_j and q adds S_i, they will eventually deliver the same suffix of messages. Since every subscription message will be decided in a unique consensus instance ($k_i \neq k'_i$ and $k_j \neq k'_j$), the two processes will calculate $\alpha \neq \alpha'$. Accordingly, the common suffix of delivered messages start after $max(\alpha, \alpha')$, since both replication groups G and G', will start deterministically deliver messages beginning from the $k_{\alpha,\alpha'}$ consensus instance from the first stream S. □

4.5 Scalable services with Elastic Paxos

Designing services that are highly available and capable to scale throughput without giving up strong consistency is a daunting task. In this chapter, we extended the MRP-Store (Section 3.5.1) to be a strongly consistent service that ensures linearizability and supports dynamic subscriptions.

Figure 4.4 illustrates a key-value store service developed with atomic multicast. There are commands to read and write single entries in the store (*get* and *put*) and to query multiple entries (*getrange*). For simplicity, we partition the store in two partitions, although the approach trivially generalizes to a higher

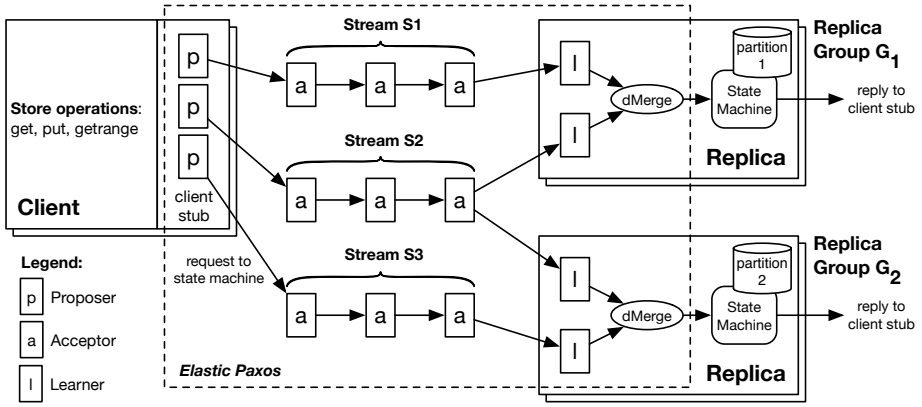

Figure 4.4. Architecture overview of a highly available and scalable store service developed with elastic multicast.

number of partitions. Replicas in G_1 subscribe to streams S_1 and S_2 and replicas in G_2 subscribe to streams S_2 and S_3. Atomic multicast is implemented with Elastic Paxos, which pipelines acceptors in a stream. The streams that a replica subscribes to are combined by the dMerge component.

Client commands are multicast to the replicas by the client stub, which receives the response, possibly combines the answer from different replica groups and returns the answer to the client. *get* and *put* commands are multicast to the stream that reaches the required partition, either S_1 or S_3; *getrange* commands are multicast to S_1 or S_3, if the command involves a single partition, and to S_2 if the command involves both partitions.

In the case of a multi-partition *getrange* command, replicas coordinate the execution to ensure linearizability [21]. Without coordination, single-partition and multi-partition commands can interleave in ways that may violate linearizability. We use the coordination technique proposed in [21] to guarantee linearizable execution of commands.

4.6 Implementation

To evaluate the capabilities of Elastic Paxos, we extended the URingPaxos library[2] to handle dynamic subscriptions. URingPaxos implements Ring Paxos [82], a high throughput atomic broadcast protocol based on TCP. Further, it implements atomic multicast by combining multiple instances of Ring Paxos [81]. The library

[2]https://github.com/sambenz/UringPaxos

is written in Java with some performance critical sections in C (JNI). URingPaxos uses ZooKeeper [60] to store ring management and protocol configuration data. Elastic Paxos replaces the static deterministic merge procedure of URingPaxos with a new procedure (Algorithm 1).

To demonstrate Elastic Paxos in a real application, we extended a partitioned key-value store service (MRP-Store) with operations to handle *subscribe* and *unsubscribe* events and support for dynamic scalability. Clients can submit *put*, *get*, and *getrange* commands to replicas. Replicas execute the commands to their in-memory data store and reply back directly to the client. Every replica belongs to one hash-partitioned partition of the whole state and every partition has a dedicated Paxos stream to order commands. To achieve linearizability for multi-partition operations, the replicas coordinate their executions with direct *signal* messages [21].

An important part of Elastic Paxos is recovery. The URingPaxos library has several mechanisms built in to recover and trim Paxos acceptors log and coordinate replica checkpoints and state transfer [14, 15].

Further, we added support to OpenStack. A controller or a client can create or destroy virtual machines, forming additional streams depending on the currently measured application throughput. Adding a new stream from newly created virtual machines (three acceptors) takes approximately 60 seconds.

4.7 Experimental evaluation

In this section, we describe our experimental environment, explain our goals and methodology, and evaluate Elastic Paxos.

4.7.1 Experimental setup

All experiments were performed on SWITCHengines,[3] an IaaS cloud service for academics. The platform uses OpenStack to provide virtual machines and Ceph as a distributed parallel block storage, serving the virtual machines.

The hardware consists of 32 physical machines; 16 are dedicated for compute nodes and 16 act as storage nodes. Every node (Intel S2600GZ) has 256 GB of main memory. The distributed file system uses 128 4 TB (WD4000F9YZ) spinning drives and a replication factor of 3. During our experiments, approximately 500 other virtual machines were running on the cluster.

[3] http://www.switch.ch/services/engines/

All virtual machines used in the experiments have 2 vCPU and 2 GB of memory. The network between these VMs is virtualized and tunneled between the physical nodes. Paxos acceptors and replicas are scheduled to different physical machines using the OpenStack anti-affinity host groups. Since the virtual machines do not provide local storage on real disk devices, all experiments were run in memory only.

URingPaxos has two important parameters, λ and Δ_t. λ defines the maximum virtual system throughput per stream, measured in Paxos instances per second. Δ_t defines the sampling interval to compare the actual throughput in a stream and λ. In all experiments, λ is set to 4000 and Δ_t to $100ms$.

4.7.2 Objectives and methodology

We assess the behavior of Elastic Paxos under a range of different practical deployments, as described next.

- We evaluate the performance of Elastic Paxos when multiple Paxos streams are added dynamically to a set of replicas. This is important in practice whenever the ordering protocol is the bottleneck in a SMR setup.
- We assess how Elastic Paxos can be used with a partitioned key-value store application to dynamically re-partition the replicas under load. Re-partitioning is required whenever the replicas are the bottleneck (e.g., due to CPU saturation).
- We demonstrate how a set of Paxos acceptors can be reconfigured under full system load. This is useful to replace a failed acceptor or an acceptor that runs out of disk storage.
- We demonstrate how Elastic Paxos can be used as an atomic multicast protocol to send consistent cross-partition commands, like creating partial snapshots. Consistent multi-partition commands are required for any task that requires total order across partitions.

4.7.3 Vertical elasticity

In this experiment we demonstrate how Elastic Paxos can be used to dynamically add multiple streams to a single set of replicas.

Setup. We start the experiment with a client VM (5 threads per stream) that sends 32 kbyte values to two replica VMs. We limited the single stream throughput to 30% not to saturate the replicas at the beginning of the experiment. Every

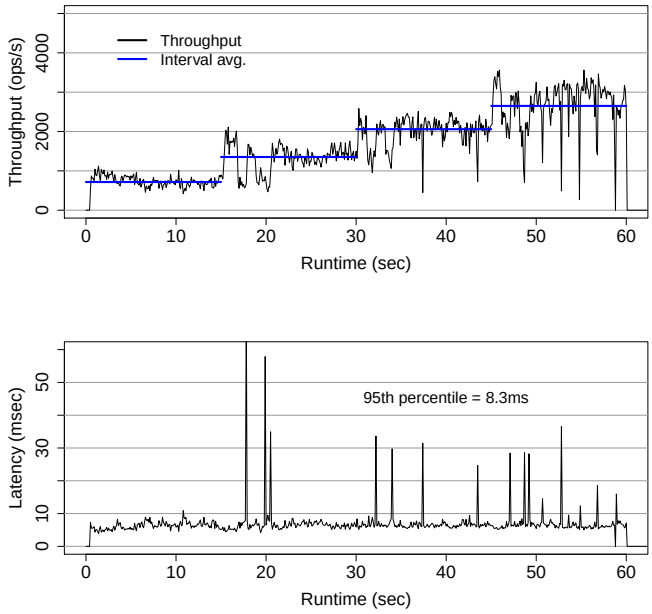

Figure 4.5. Dynamically adding streams to a set of replicas to scale up the coordination layer. Every 15 seconds replicas subscribe to a new stream.

15 seconds replicas subscribe to a new stream and immediately deliver new commands from the added stream. Every stream contains 3 acceptor VMs which are deployed as OpenStack Heat-AutoScaling groups. In this experiment, all VMs are started up from the beginning, but Heat-AutoScaling allows clients to boot up or shutdown the virtual machines that participate in the streams.

Results. Figure 4.5 shows the aggregated throughput at the replicas. The most visible impact is right after the subscribe message. This is due to the fact that we intentionally do not use the *prepare_msg* request (see 4.4.3) to inform replicas about the changes. During recovery of the new stream, a number of messages are queued up in memory at the replicas and delivered right after the subscription process is over. The interval averages increasing from 735, 1498, 2391 to 2660 ops/s by adding additional streams. With 4 streams, this corresponds to an increase of 3.62 times the system throughput.

4.7.4 Horizontal elasticity

In this section we evaluate how Elastic Paxos can be used to dynamically scale out a partitioned key-value store. For this experiment we use the partitioned key-value store described in Section 4.6.

Setup. We start the experiment with a client VM (100 threads) that sends 1024-byte put commands to random keys. Two replica VMs apply these commands to their local in-memory storage and send back a command response to the client thread. Initially only one partition is present in the system and serves every request. Every partition is coordinated by a stream of 3 acceptor VMs. At 30 seconds, one of the replicas subscribes to a new stream with additional 3 acceptors and informs the whole system 5 seconds later about the partition change. The client is notified about the change in the partitioning by ZooKeeper and starts sending random commands to both partitions.

Results. Figure 4.6 shows the system throughput during re-partitioning under 75% peak load. The duration of the re-partitioning is 1 second and mainly caused by a client timeout. Commands from clients which are received by the wrong partition after the split are discarded. The clients will resend them after a timeout to the correct partition. The throughput after splitting the partition is half at every replica. Further, also the CPU consumption at every replica drops after the re-partitioning event. Therefore, both partitions could now clearly handle 100% more operations per second.

4.7.5 Reconfiguration

In this experiment we show how Elastic Paxos can be used to reconfigure a state machine under full system load. Since reconfiguration of atomic broadcast is a sub problem of reconfigure atomic multicast, we use dynamic subscriptions to replace the set of acting acceptors. Changing the set of acceptors is required, if for example they run out of disk space, one acceptor stable storage is not recoverable or to tolerate more failures (e.g., 5 instead of 3 acceptors). The goal of this experiment is to show that dynamic subscription is an efficient solution to state machine reconfiguration.

Setup. We start the experiment with a client VM (60 threads) that sends 32 kbyte values to two replica VMs. These two replicas subscribe to the first stream which contains 3 acceptor VMs. After 40 seconds, we inform the replicas that we will add a second stream (with a *prepare_msg* request). After 45 seconds we let the replicas subscribe to the new stream containing 3 different acceptor VMs. Right after the subscribe message we submit a unsubscribe message to the

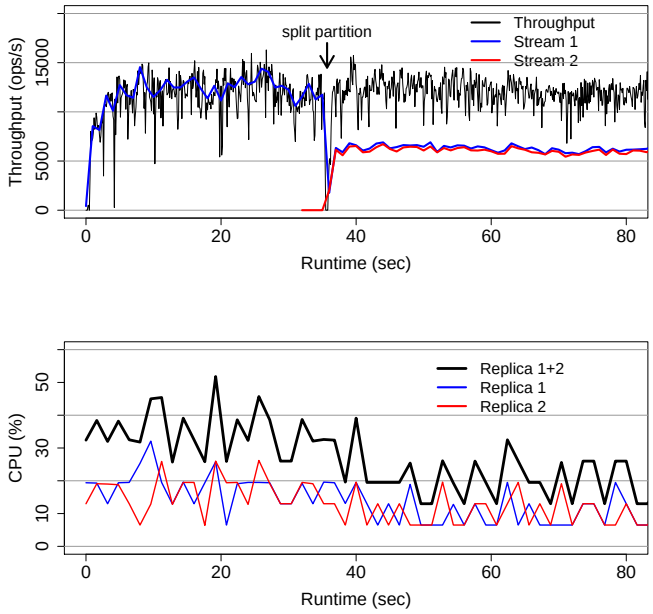

Figure 4.6. Re-partitioning of a key-value store (75% peak load). After 35 seconds the throughput and CPU consumption at both replicas decreased.

original stream.

Results. Figure 4.7 shows the reconfiguration under full load of 550 Mbps. Since the replicas received a *prepare_msg* (see 4.4.3), they can start up and recover the new stream in the background without blocking the main message execution. With this optimization, reconfiguration introduces almost no overhead.

4.7.6 Consistent cross-partition commands

We now show how Elastic Paxos can link any subset of partitions to send consistent cross-partition commands.

Setup. We start the experiment with a client VM (70 threads) sending 1024-byte put commands to random keys. Four replica VMs serve each 1 partition, apply these commands to their local in-memory storage and send back a response (using a UDP) to the client thread. All partitions are coordinated by a stream of 3 acceptor VMs each. Every 12 seconds we subscribe different partitions to an additional fifth stream and send a getrange command to it. Right after sending

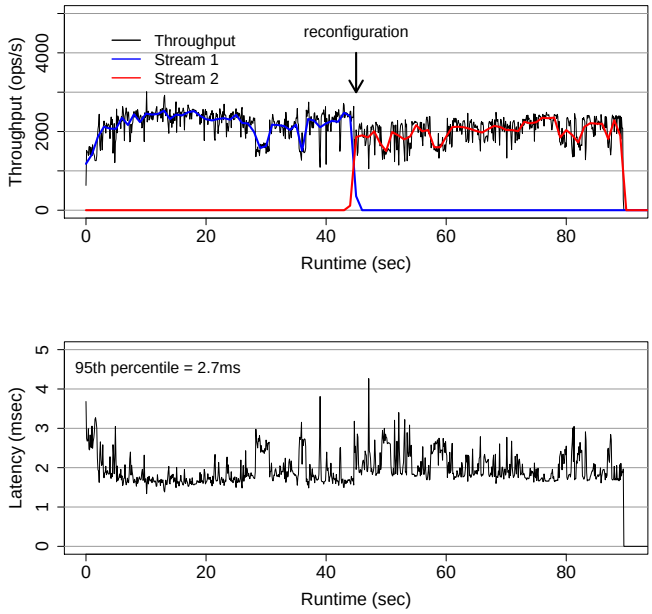

Figure 4.7. State machine reconfiguration under full system load. At 45 seconds we replace the set of active acceptors with a new one.

the command, we unsubscribe both replicas from the common stream.

Results. Figure 4.8 shows the throughput at the client that sends put commands to all 4 partitions. The performance impact, visible when any subset of partitions is connected, is due to the increased latency during re-configuration. The fixed amount of threads block and depending on how many partitions are involved or randomly addressed the aggregated throughput drops during a short period. The latency of the getrange commands is less than 100 msec but higher than the one of the put commands. This is because they arrive at the replicas during the subscription process.

4.8 Related work

In this section, we briefly review related work on atomic multicast, group membership and state machine reconfiguration.

Atomic multicast. An overview of atomic multicast algorithms is provided in Section 3.8. Elastic Paxos is based on URingPaxos which is iteslef based on Multi-

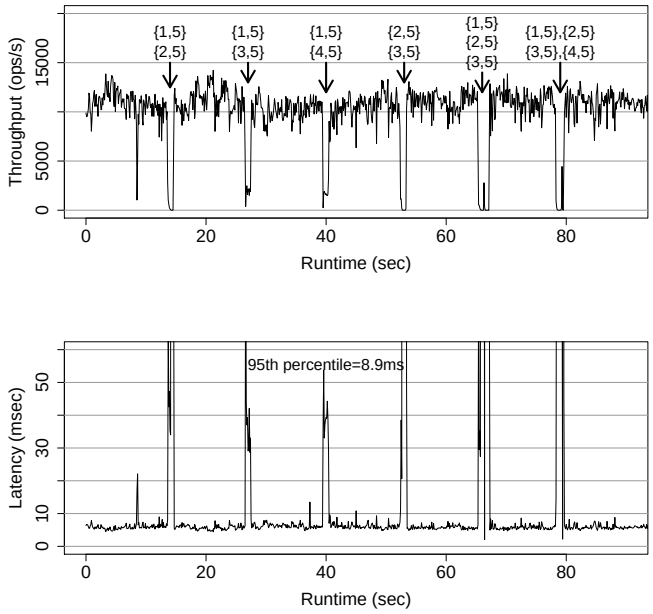

Figure 4.8. Use Elastic Paxos to send consistent cross-partition commands. The different subset of partitions are created dynamically at runtime.

Ring Paxos [81]. The algorithm of Elastic Paxos to reconfigure the deterministic merge function is similar to [23]. The chosen timestamps for subscribe and unsubscribe messages are the stream positions which are persisted in the streams themselves. Therefore for reconfiguration, Elastic Paxos requires each involved stream (group) to be able to reach consensus.

E-Cast [109] addresses similar problems like Elastic Paxos does. Compared to Elastic Paxos, E-Cast defines multicast as a stateful routing problem. E-Cast uses replicated routers (sequencers) to partially order messages and reconfigure the system, while Elastic Paxos uses the deterministic merge function in each replica. Therefore, Elastic Paxos does not require a global sequencer to order messages and reconfigure the system.

Group membership. Group membership has been an active field of research for decades, in the context of group communication protocols [31]. While in atomic broadcast total order is achieved by a sequence of individual consensus rounds, group communication protocols are based on a sequence of view changes. Group membership is a special case of the set membership problem, in which all pro-

cesses decide on which non-faulty processes belong to the current set (view) [96].

In Elastic Paxos, the round-robin delivery order can be seen as a dynamic set of changing streams. While the total order within a stream is based on atomic broadcast, the deterministic merge function is based on a sequence of subscription changes, similar to view changes. Compared to group communication protocols, Elastic Paxos does not use view changes to remove faulty replicas, but to dynamically scale. Additionally, the subscriptions in Elastic Paxos are persisted in the streams, every recovering replica can re-learn all subscription changes.

Rollup [54] is a protocol designed for fast cluster membership updates. The main goal is to avoid disruptive behavior when the master or leader of a protocol is replaced. Since Elastic Paxos is based on Paxos, frequent changes of the coordinator have an impact on performance. Compared to Rollup, Elastic Paxos is designed to scale atomic multicast groups rather than addressing fast replacement of the Paxos leader.

State machine reconfiguration. Changing the set of acting acceptors is discussed in [71, 73]. Elastic Paxos uses a different approach. It does not change the set of the acceptors itself, rather it replaces all of them by a new set (i.e., new stream).

Group communication protocols reconfigure the system to tolerate failures (e.g., process crashes). In general they use a fault-tolerant consensus algorithm to coordinate the view change. As already described, Elastic Paxos uses a similar way to add and remove new streams.

Similar to Elastic Paxos, SMART [74] uses different independent Paxos streams to reconfigure a replicated state machine. But, while SMART changes the set of replicas, Elastic Paxos keeps the replication group constant and changes the subscriptions. This allows Elastic Paxos, additionally to reconfiguration, also to scale by adding multiple Paxos streams to a single replication group. Adding a new replica to a replication group is part of Elastic Paxos's recovery procedure.

Elastic SMR [86] optimizes static splits and merges of partitions. Instead of implementing ad-hoc state transfer protocols and performing scaling operations as background tasks, it proposes a modular partition transfer protocol for creating and destroying such partitions at runtime. The view manager of BFT-SMaRt [20] uses robust algortithms that tolerate byzantine failures.

Eve [65] implements scalable state machine replication on multi-core servers, but it is static and does not allow reconfiguration. DynaStore [4] allows reconfiguration without consensus and can operate in a completely asynchronous system [48]. However, compared to Elastic Paxos, DynaStore considers a strictly weaker model (i.e., read/write register instead of an arbitrary state machine).

Chapter 5

Distributed Atomic Data Structures

5.1 Introduction

Most modern cloud services are distributed systems. Today's on-demand computing resources, common in public cloud environments, provide operators of these systems with the possibility to provision as many servers as needed by the service and to react quickly to changes in application workload. Starting up new servers once increased traffic is detected and shutting down low utilized servers to save costs are common operations. While it is relatively easy to reconfigure stateless components (e.g., application servers), dynamically provisioning stateful components (e.g., storage) is complicated.

Major effort has been spent in the recent years to improve the performance, scalability and reliability of distributed data stores. But when it comes to using research results in real applications, existing solutions are often not sufficient. Implementing applications that support strong consistency, elastic scalability and efficient recovery is a daunting task.

Scalable state machine replication has been shown to be a useful technique to solve the above challenges in building reliable distributed data stores [14, 21]. However, implementing a fully functional system, starting from the atomic multicast primitives, supporting required features like recovery or dynamic behavior is a challenging and error-prone task. Providing higher-level abstractions in the form of distributed data structures can hide this complexity from system developers. For example, given a distributed B-tree, services like distributed databases [5] or file systems [75] can be implemented in a distribution transparent manner. In this chapter we discuss how to implement a distributed ordered map as a ready-to-use data structure.

Existing distributed data structures often rely on transactions or distributed

locking to allow concurrent access. Consequently, operations may abort. A behavior which must be handled by the application. We implemented a distributed ordered map (DMap) that does not rely on transactions or locks for concurrency control. Relying on atomic multicast, all partially ordered operations succeed without ever aborting. Additionally, DMap is scalable, fault-tolerant and supports consistent long-running read operations on multi-partition snapshots to allow background data analytics.

This chapter makes the following contributions. First, we propose a lock-free distributed ordered map with strong multi-partition consistency guarantees that implements the Java SortedMap interface. Second, we show how DMap can be used to reliably distribute any Java application, like a transactional database. Third, we detail the implementation of DMap and highlight the underlying replication and ordering techniques. Finally, we provide a performance assessment of all these components.

5.2 DMap Service

DMap is a distributed sorted key-value store which implements the full Java SortedMap (Table 5.1,5.2) and the ConcurrentMap interface (Table 5.3).

It is generic in the sense that it allows arbitrary Java objects as keys and values. For example, one can define a SortedMap that uses Integer objects as keys and String objects as values:

```
SortedMap<Integer,String> m;
```

or a map that uses String objects as keys and holds other complete Java maps as values. DMap also supports user generated objects, as long as they are Java serializable:

```
SortedMap<String,Map<String,YourObject>> n;
```

DMap can be used to distribute any local Java application relying on a SortedMap (or Map) by simply replacing the interface implementation. For example,

by replacing:
`SortedMap<K,V> m = new TreeMap<K,V>();`
by:
`SortedMap<K,V> m = new DistributedOrderedMap<K,V>(...);`

DMap uses dynamic atomic multicast (Chapter 4) to implement a lock-free concurrent data structure. All operations in DMap are strongly consistent and ensure linearizability (Section 2.7). This includes multi-partition commands like *size()* or *subMap()* (range).

Linearizability is important, since DMap is built to replace local data structures. Because we can not know the guarantees a Java application expects from the underlying data structure, either implicit or explicit, DMap must provide the strongest form of consistency.

Therefore, DMap runs with any existing code. For example, iterating over all entries in the data structure can be achieved as follows.

```
Iterator i = dmap.entrySet().iterator();
while(i.hasNext()){
   Entry e = i.next();
   System.out.println(e);
}
```

Iterators operate on consistent multi-partition snapshots (Section 5.3.2) and never throw a *ConcurrentModificationException*. The ordered set of entries is streamed to the client (Section 5.3.1) as it uses it. This implementation allows long-running data analytics operations over huge data sets.

5.3 System Architecture

In this section, we give an overview of DMap, detail how the dynamic ordering protocol is used, explain the replicated database and highlight the implemented recovery techniques.

5.3.1 DMap overview

DMap achieves scalability trough hash partitioning, supports dynamic re-partitioning, recovery and uses scalable state machine replication to provide fault-tolerance.

Clients use the Apache Thrift [1] RPC framework to communicate with the DMap servers. To initialize a DMap client, a map identifier and a connection

[1] https://thrift.apache.org/

Table 5.1. DMap operations (Java Map interface).

Interface **Map<K,V>**
/* Removes all of the mappings from this map. */ void **clear()** /* Returns true if this map contains a mapping for the specified key. */ boolean **containsKey(K key)** /* Returns true if this map maps one or more keys to the specified value. */ boolean **containsValue(V value)** /* Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key. */ V **get(K key)** /* Returns true if this map contains no key. */ boolean **isEmpty()** /* Associates the specified value with the specified key in this map. */ V **put(K key, V value)** /* Removes the mapping for a key from this map if it is present. */ V **remove(K key)** /* Returns the number of key-value mappings in this map. */ int **size()**
Interface **Iterator<E>** (from keySet(),values(),entrySet())
/* Returns true if the iteration has more elements. */ boolean **hasNext()** /* Returns the next element in the iteration. */ E **next()**

to a ZooKeeper [2] server are required. Zookeeper is used to look-up at least one DMap server, which will be used to download the initial system partition map.

The partition map is a mapping between a 32 bit integer token, indicating the position in a hash ring, and a set of DMap servers responsible for the corre-

[2] https://zookeeper.apache.org/

Table 5.2. DMap operations (Java SortedMap interface).

Interface **SortedMap**<K,V> extends Map<K,V>
/* Return a view of the portion of this map whose keys range from fromKey to toKey: */ SortedMap<K,V> **subMap**(K fromKey, K toKey); /* Return a view of the portion of this map whose keys are strictly less than toKey: */ SortedMap<K,V> **headMap**(K toKey); /* Return a view of the portion of this map whose keys are greater than or equal to fromKey: */ SortedMap<K,V> **tailMap**(K fromKey); /* Return the first (lowest) key currently in this map: */ K **firstKey**(); /* Return the last (highest) key currently in this map: */ K **lastKey**(); /* Return a Set view of the keys contained in this map: */ Set<K> **keySet**(); /* Return a Collection view of the values contained in this map: */ Collection<V> **values**(); /* Return a Set view of the mappings contained in this map: */ Set<Entry<K, V>> **entrySet**();

sponding token. Every DMap client holds a cached version of the partition map, including their version number in memory. DMap servers are contacted directly for a specific command on a key (e.g., *put(K,V)*); or any server can be contacted to send multi-partition commands (e.g., *size()*). Commands include the partition map version. If a server detects a command with a outdated version number, the client will be notified and can install the most recent partition map. Figure 5.1 shows the Client-Server RPC communication. Among the servers responsible for a partition, a client chooses randomly one for each command.

Client commands, received trough Thrift at one server, are atomically multi-

Table 5.3. DMap operations (Java ConcurrentMap interface).

Interface **ConcurrentMap<K,V>**
/* If the specified key is not already associated with a value, associate it with the given value. */
V **putIfAbsent(K key, V value)**
/* Removes the entry for a key only if currently mapped to a given value. */
boolean **remove(Object key, Object value)**
/* Replaces the entry for a key only if currently mapped to some value. */
V **replace(K key, V value)**
/* Replaces the entry for a key only if currently mapped to a given value. */
boolean **replace(K key, V oldValue, V newValue)**

casted to all involved servers; executed by all involved servers and the response is sent back to the client by the one server that originally received the command.

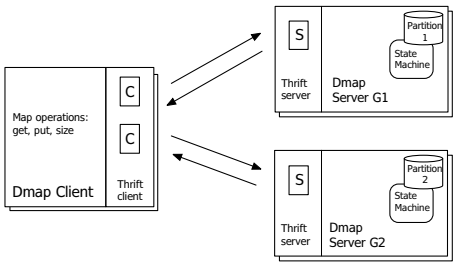

Figure 5.1. DMap Client-Server communication.

5.3.2 Multi-Partition Snapshots

To ensure the ordered delivery of entries while iterating over the hash partitioned map, DMap clients proceed as follows: First, they create a global consistent in-memory snapshot at all partitions. Second, they stream the snapshot in parallel from every partition a couple of entries at a time. Third, they deliver to the application the lowest entry of all partitions until all entries are delivered. This

procedure allows to iterate over a huge amount of data, since only some of them are kept in memory.

The key to implementing such efficient iterators over a hash-partitioned system is the ability to create multi-partition snapshots. Creating such snapshots is complicated since the partitions (or even the processes) do not share a common clock [29]. DMap relies on atomic multicast to create in-memory snapshots at the replicas. Atomic multicast, as described below, allows to send partially, or in this case totally, ordered commands to be executed at every replica. As shown in [14], such global messages do not impact commands sent to a single partition.

5.3.3 DMap replicated database

DMap uses atomic multicast to order all commands for implementing scalable state machine replication (Figure 5.2). Atomic multicast is a communication abstraction defined by the primitives *multicast*(S,m) and *deliver*(m), where m is a message and S is a multicast stream. Processes choose from which multicast groups they wish to deliver messages. If process p chooses to deliver messages multicast to stream S, we say that p *subscribes to* stream S.

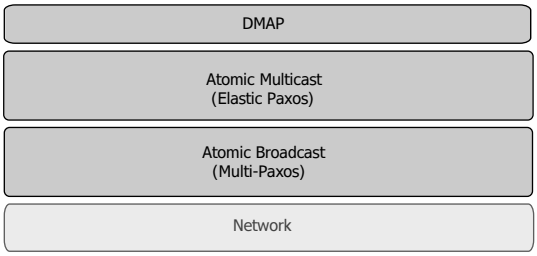

Figure 5.2. Atomic multicast protocol stack.

Atomic multicast, as discussed above, relies on static subscriptions of replicas to streams, that is, subscriptions are defined at initialization and can only be changed by stopping all processes, redefining the subscriptions, and restarting the system. To let DMap dynamic re-partition the state, it relies on Elastic Paxos (Chapter 4). Elastic Paxos allows to dynamically add and remove resources to / from an online partially replicated state machine.

Elastic Paxos extends the atomic multicast interface with two additional primitives: *subscribe_msg*(G,S) and *unsubscribe_msg*(G,S), which replicas in replication group G can use to subscribe to and unsubscribe from stream S. After replicas subscribe to stream S, they will eventually deliver messages multicast to S.

Similarly, if replicas unsubscribe from S, they will eventually stop delivering messages multicast to S. In both cases, atomic multicast guarantees acyclic ordered delivery.

The design of DMap is similar to the one of MRP-Store introduced in Section 4.5. There are however, some important differences between MRP-Store and DMap. First, DMap has a fully replicated partition map which is part of the system itself. While clients in MRP-Store rely on ZooKeeper to locate replicas, DMap clients can retrieve the partition map from every replica. Further, changes in the partitioning schema of DMap are partially ordered with all other requests in the system. Second, in MRP-Store all replicas send back an UDP message as a response to an executed command. In DMap, the client-server communication is entirely handled by Thrift (TCP) and therefore only the randomly chosen replica is answering to a client. This improves the overall throughput, since not all replicas are required to respond to a command. Moreover, UDP showed poor performance in some cloud environments. Third, DMap uses global in-memory snapshots to optimize the state transfer during recovery and re-partitioning. After a checkpoint is created, the data can be streamed by multiple replicas in parallel.

5.3.4 Recovery

The mechanism used by a process to recover from a failure depends on the role played in the server. In a typical deployment of Paxos (e.g., state machine replication, clients broadcast commands and replicas deliver and execute those commands in the same total order before responding to the clients. In this case, clients act as proposers and replicas as learners, while acceptors ensure ordered delivery of messages.

Acceptors need information related to past consensus instances to serve retransmission requests from recovering replicas. So, before responding to a coordinator's request with a Phase 1B or Phase 2B message, an acceptor must log its response onto stable storage. This ensures that upon recovering from a failure, the acceptor can retrieve data related to consensus instances it participated in before the failure.

Learners can always recover by requesting a retransmission of decided instances from the acceptors. However, such retransmission negatively impact the system throughput [19]. Therefore, each replica periodically checkpoints its state onto stable storage. Upon resuming from a failure, the replica retrieves and installs its last stored checkpoint and recovers from the acceptors the commands missing in this checkpoint (i.e., the commands executed after the replica's last

checkpoint). Acceptors can coordinate with replicas to delete data about old Paxos instances. If all replicas have saved a checkpoint that reflects messages decided in the i-th Paxos instance, then acceptors can delete data related to instances prior to i [14].

DMap can be configured to recover from stable storage like explained above. However, it supports also a full in-memory mode. In such a configuration, Paxos acceptors keep only the last 15k instances in memory and replicas do not checkpoint to stable storage. A recovering replica will first subscribe to all required multicast groups. Elastic Paxos ensures that after subscribing to all streams, the message ordering is guaranteed. Followed by requesting the most recent partition map and a snapshot of the current data on all replicas. To download and install of these checkpoints a recovering replica behaves like a DMap client. The version of the partition map and the snapshot id are the unique values of the Paxos instance in which the commands are decided. Therefore, the recovering replica can skip learned commands before the snapshot id and start applying commands with ids right after the snapshot. To finish recovery, a replica adds itself to the system partition map. After this point, clients will start sending command to the recovered replica.

5.4 H2 database on DMap

To demonstrate how useful a distributed data structure like DMap is, we implemented a transactional database on top of it. We replaced the storage engine (MVStore) of the H2 [3] database by DMap.

H2 has a modular design, which encapsulates the SQL query processor from the storage, B-tree, layer. Assuming a distributed storage engine, multiple independent H2 query processor instances can run simultaneous on the same distributed data.

The core of H2 is MVStore. MVStore allows to create multiple independent sorted maps. The whole database relies on this storage abstraction. All database schema information, primary and secondary indexes, even the undo log is persisted in this layer. Therefore, replacing the MVMap used by MVStore with DMap distributes the whole database. H2 with MVStore supports read-commited transactions. Even tough DMap is linearizable, it can not provide stronger guarantees than MVStore itself.

We needed less than 500 lines of source code to achieve our goal and run multiple H2 instances on top of DMap. Moreover, the modular design of H2

[3] http://www.h2database.com

and the expressive interface of DMap allows us to use all special database operations, like: creating or altering tables, creating indexes or using transactions without further modifications. The new system supports distributed transactions, based on a distributed undo log, and online database schema altering (e.g., creating tables) which are immediate visible to all query processors. However, since some query optimizers rely on data local to the query processors, such operations would require additional work to distribute the required information.

By adding one additional Java class to H2, we could not only distribute the whole database, but due to the properties of DMap, we could implement a scalable (partitioned) and fault-tolerant (active replicated) system.

5.5 Experimental evaluation

In this section, we experimentally assess various aspects of the performance of our proposed systems:

- We measure the baseline performance and horizontal scalability of DMap.
- We evaluate the impact of recovery on performance.
- We evaluate the performance under splitting a partitioning.
- We use TPC-C [4] to evaluate the performance of H2 running on DMap.

5.5.1 Hardware setup

All the experiments were performed in a cluster of 10 HP SE1102 servers, equipped with 2x 2.5 GHz Intel Xeon CPUs and 8 GB of main memory. These servers were interconnected through a HP ProCurve 2910 switch with 1 Gbps interfaces. The round trip time is 0.1 millisecond between the nodes. In all the experiments, clients and servers were deployed on separate machines. Elastic Paxos was initialized as follows: $\Delta = 5$ millisecond, $\lambda = 15$k and use in-memory storage. We keep the machines approximately synchronized by running a NTP service.

[4] http://www.tpc.org

5.5.2 Throughput and Latency

Scalability of DMap

Setup. In this experiment 60 clients per partition send *put()* commands to random keys in a closed loop. The values are strings of approximately 380 bytes each. We use up to 3 partitions. Every partition is served by 3 replicas running on one server each.

Results. Figure 5.3 (left) shows the throughput increase of the overall system while adding new partitions. The scalability is linear in the number of partitions, while it still offers the ability to execute consistent cross-partition commands. Figure 5.3 (right) shows the aggregated throughput and latency over time to three partitions. Avg. throughput is 33170 operations per second with an average latency of 5.3 ms.

Figure 5.4 shows the cumulative distribution function of the latency for all requests. Commands to one partition show a sharp CDF around the average latency. Increasing the number of involved partitions also increases the coordination overhead of Elastic Paxos. In Elastic Paxos, inbalances of client loads are compensated every Δ time interval. In this case 5 ms. Adding more partitions increases the probability that one partition must wait until it can proceed with the next executions. This is visible in the CDF with a bend in the curve around 5 ms.

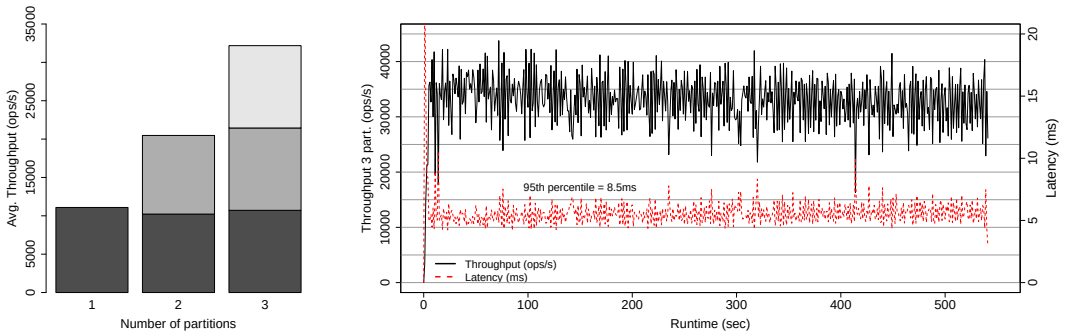

Figure 5.3. Throughput scalability (left) of DMap with 3 partitions. Runtime behavior (right) of throughput and latency with 3 partitions.

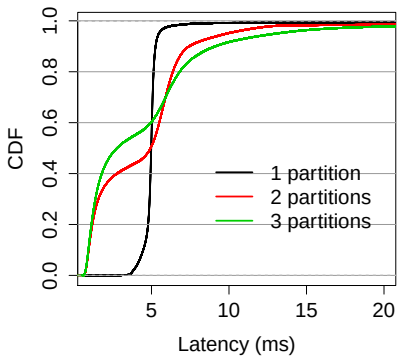

Figure 5.4. Cumulative distribution function of the command executions for 1 to 3 partitions.

Performance of Iterators

Setup. In this experiment, an increasing number of clients create an iterator (snapshot) in DMap with 3 partitions. We measure how fast every client can iterate over the whole distributed data set. The data set was previously provisioned with 1.2 million key-value pairs.

Results. As seen in Figure 5.5, the iterators show a better performance than the single command throughput. Initially, creating a snapshot is slow (200 ms), but once a snapshot (iterator) is created, every client can stream the data parallel from every replica in all partitions. A single iterator achieves 50k entries per second while the number of parallel iterators scales almost linearly up to 50 clients.

Yahoo! Cloud System Benchmark

Setup. We evaluate the performance of DMap using the Yahoo! Cloud System Benchmark (YCSB [35]). To evaluate a baseline performance, we compare with an unreplicated server, using only the Thrift interface. Both systems are deployed with 3 partitions and we use 180 clients. The data set was previously provisioned with 1.2 million key-value pairs. All 6 core workloads are evaluated.

Workload A (Update heavy workload): This workload has a mix of 50/50 reads and writes. An application example is a session store recording recent actions.

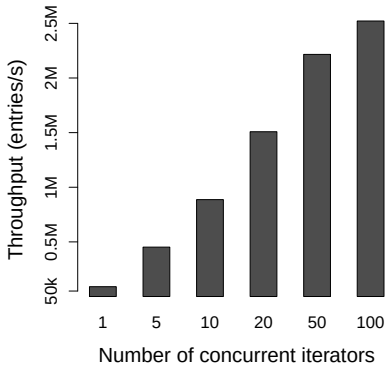

Figure 5.5. Performance of retrieving entries of an DMap iterator for 1 to 100 parallel clients.

Workload B (Read mostly workload): This workload has a 95/5 reads/write mix. An application example is photo tagging where adding a tag is an update, but most operations are reads.

Workload C (Read only): This workload is 100% read. An application example is a user profile cache, where profiles are constructed elsewhere (e.g., Hadoop).

Workload D (Read latest workload): In this workload, new records are inserted, and the most recently inserted records are the most popular. An application example is user status updates, where people want to read the latest.

Workload E (Short ranges): In this workload, short ranges of records are queried, instead of individual records. An application example is a threaded conversations, where each scan is for the posts in a given thread (assumed to be clustered by thread id).

Workload F (Read-modify-write): In this workload, the client will read a record, modify it, and write back the changes. An application example is a user database, where user records are read and modified by the user or the user activities are recorded.

Results. The YCSB throughput of all workloads is shown in Figure 5.6. Workload B,C and D correspond to the baseline performance of single partition commands. Workload A and F send update and read-modify-write commands. The way YCSB is implemented in DMap, such commands are composed of a read, followed by a write command. YCSB is a multi-map which allows to update a single entry in the value. DMap must first read the value as map, update field and put again the

whole value.

Workload E shows the performance of small scans. Retrieve a scan in DMap, creates an iterator and loops over a small amount of values. Since the cost in DMap is creating iterators (snapshots) and not looping over iterators, the overall performance in case E is only 290 scans per second.

In all workloads, except E, the unreplicated Thrift implementation is faster. This is obvious, since all partitions run independent from each other (consistent scans are not possible) and there is no latency overhead of atomic multicast.

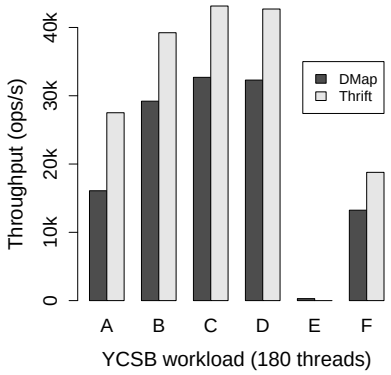

Figure 5.6. Yahoo! Cloud Serving Benchmark for A:update heavy, B:read mostly, C:read only, D:read latest, E:short ranges, F:read-mod-write workloads.

5.5.3 Recovery

Setup. As in the previous experiments, we use 180 clients to generate load on 3 partitions with 3 replicas each. The data set was previously provisioned with 1.2 million key-value pairs. After 20 s we kill one replica on one partition. At 40 s we bring back the killed replica, which immediate starts to recover (Section 5.3.4).

Results. Figure 5.7 shows the system throughput over time while recovery is active. Annotation (1) indicates the kill of one replica.

The performance drops to almost zero, since all commands to the failed replica are timing out. Additionally, all clients have to update their locally cached partition map. The partition map got updated, because the killed replica was removed. At (2), the replica starts recovering. Point (3) indicates the end of recovery. The recovered replica updates the partition map with the information

that it is operational. Clients will install a new partition map; but compared to (1), no Thrift connections are invalidated. State transfer while recovering is very fast, since it uses the iterators evaluated in Figure 5.5.

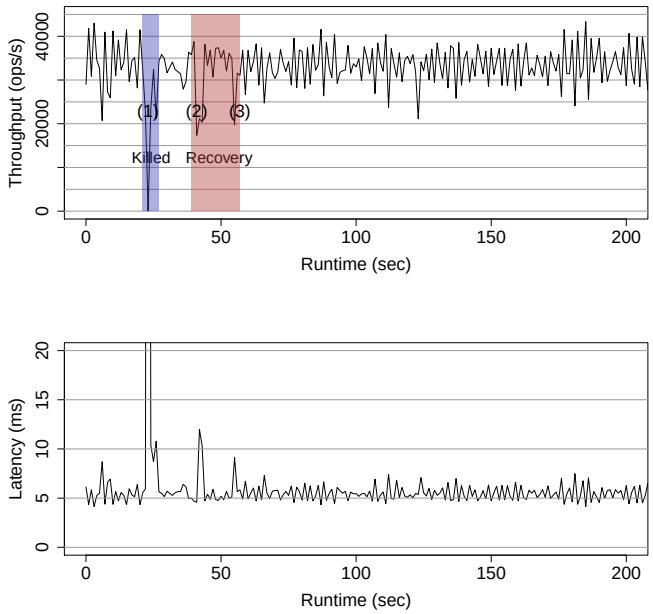

Figure 5.7. Impact on client throughput due to recovery of a DMap replica under full system load.

5.5.4 Re-Partitioning

Setup. In this experiment we start with 2 partitions (P1, P3) and after 20 s we dynamically add a third one (P2). We use 180 clients generating load and the data set was previously provisioned with 1.2 million key-value pairs. The new set of 3 replicas first recover the state from the currently available partition (not shown in this experiment), reconfigure all involved atomic multicast streams and later update the system partition map.

Results. Subscribing to and unsubscribing from multicast streams have no visible impact, as seen in Figure 5.8. The overall throughput drops during re-partitioning for a short period to 50% (half of the clients are re-assigned to the new partition). After re-partitioning, the overall throughput increases. The splitted partition (P1) was responsible for 2/3 of the hash space and therefore over-

loaded. After re-partitioning, every partition is responsible for 1/3 of the keys, which explains why the average latency decreases.

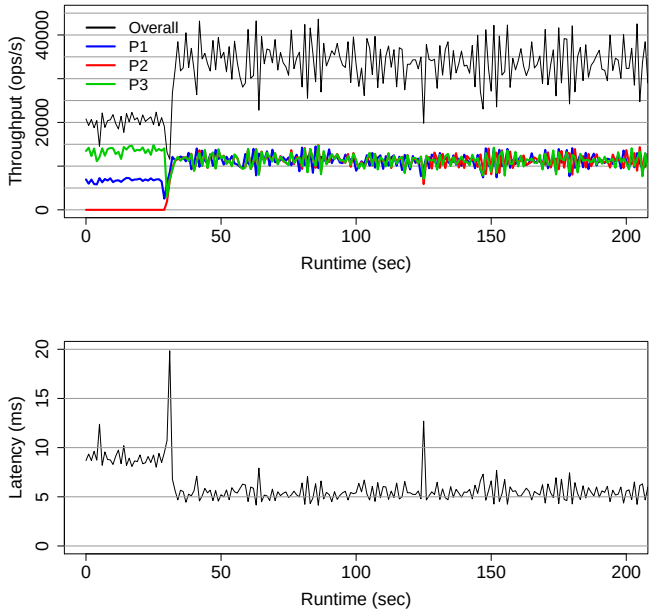

Figure 5.8. Impact on performance while splitting a partition in DMap.

5.5.5 Performance of H2 database running on DMap

Setup. In this experiment we use DMap with 3 partitions which act as distributed storage for the H2 database. To evaluate the performance we run the TPC-C online transaction processing benchmark.

TPC-C simulates a computer system to fulfill orders from customers. The company sells 10000 items and keeps its stock in 2 warehouses. Each warehouse has 10 sales districts and each district serves 300 customers. TPC-C involves a mix of five concurrent transactions (20 threads[5]) of different types and complexity:

- **New-order**: receive a new order from a customer: 43% (9 SQL statements)
- **Payment**: update the customers balance to record a payment: 43% (9 SQL statements)

[5]MVStore provides read-commited as isolation level. To execute TPC-C correctly the multi-programming level should be set to one thread..

- **Delivery**: deliver orders asynchronously: 4% (8 SQL statements)
- **Order-status**: retrieve the status of customer's most recent order: 4% (6 SQL statements)
- **Stock-level**: return the status of the warehouse's inventory: 4% (3 SQL statements)

Results. In TPC-C, throughput is defined by executed New-Order transactions per minute while the system executes all transaction types. H2 on DMap achieves 480 New-Order transactions per minute. This is 5.4 times slower than a replicated H2 instance accessed over TCP.

Figure 5.9 shows all operations the database executes on DMap during the execution of TPC-C. Single-partition commands run in parallel and can be scaled by adding new replica sets. The all-partition commands must be executed by every replica and are not scalable. The create range commands are due to select queries of a range. H2 executes more than 100 DMap operations per second. But, the TPC-C throughput of the New-Order transaction is only about 8 per second. This can be explained while analyzing how many DMap operations a SQL statement requires (Table 5.4). On average, every SQL statement requires 10 DMap operations to update the undo log twice and setting the locked and final value to the table. The H2/DMap integration is not optimized to use a transaction cache. Such a cache could possibly reduce the number of DMap operations. The current implementation to use DMap in H2 consists of only one Java class. The goal was to demonstrate the simplicity of integration and not to result in a fast database.

Table 5.4. Overview of H2 SQL queries and resulting DMap operations.

H2 :	"insert into test values (1,'String')"
DMap:	4*GET, PUT, PUTIFABSENT, 2*GET, PUT, REMOVE
H2 :	"select * from test where id=1"
DMap:	GET
H2 :	"update test set value name='XYZ' where id=1"
DMap:	4*GET, PUT, REPLACE, 3*GET, PUT, REPLACE, 2*GET, PUT, REMOVE, 2*GET, PUT, REMOVE
H2 :	"select * from test"
DMap:	SIZE, CREATERANGE
H2 :	"delete from test where id=1"
DMap:	4*GET, PUT, REPLACE, 2*GET, 2*REMOVE

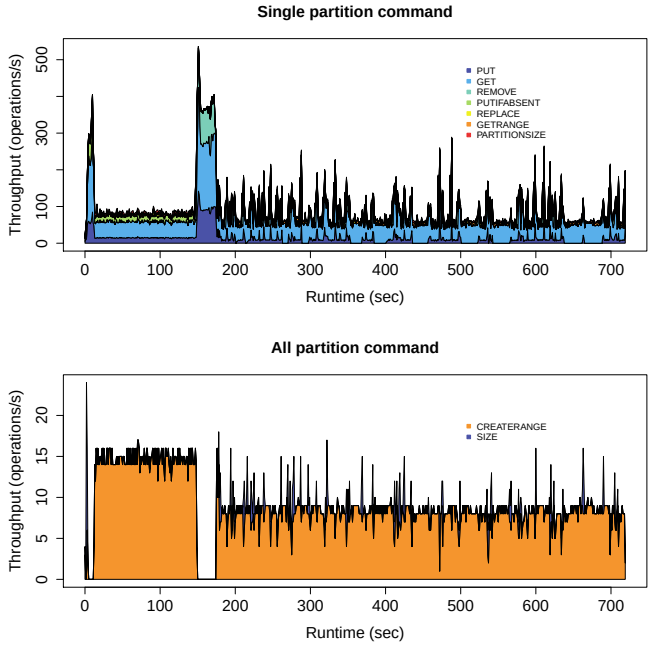

Figure 5.9. H2 operations on DMap while performing the TPC-C benchmark.

5.6 Related work

In this section, we review related work on distributed data structures, atomic multicast, and recovery.

Distributed data structures. There exists a variety of systems that implement distributed data structures. An overview is shown in Table 5.5. They provide different interfaces, consistency guarantees or are built for specific optimized use-cases. To the best of our knowledge, no system implements a generic Java interface and provides scalable, consistent range queries.

One of the first distributed data structures similar to our ordered map was a B-tree algorithm based on a B-link tree proposed in [62]. However, the tree was designed for distributed memory architectures and not high latency networks. Even in the modern literature, not many distributed tree structures exist. SD-RTree [43] is a scalable distributed R-tree designed for networks. This data structure is based on a binary tree and optimized for spatial objects. The first distributed B-tree that tries to address similar requirements to the ones described in this thesis is presented in [3]. The concurrency control is based on transactions and not locking, which was common in B-trees for distributed memory.

Table 5.5. Overview of existing distributed data structures.

System	Generic Java Iface	Type	Consistency	Partitioned
DMap	SortedMap	SortedMap	strong	yes
Yesquel	no	SortedMap	strong	yes
HBase	no	SortedMap	strong	yes
Cassandra	no	Map	weak	yes
Redisson	ConcurrentMap	Map	weak	yes
	SortedSet	SortedSet	weak	yes
Hazlecast	ConcurrentMap	Map	weak	yes
Dynamo	no	Map (w/ Scan)	weak	yes
Hyperdex	no	Map (w/ Scan)	strong	yes
SimpleDB	no	Map (w/ Scan)	consistent reads	yes
Ignite	JCache	Map	strong	replicated or partitioned
Atomix	no	Map	strong	no

Minuet, a scalable distributed multiversion B-tree [105] addresses the problem of long-running data analytics workloads in the context of short-living transactions. Minuet is based on Sinfonia [6] but provides an optimistic concurrency control mechanism to scale parallel inserts and updates. Further, it implements consistent snapshots and copy-on-write tree branches. Recent work on distributed data structures also proposes to use skip lists to implement efficient range queries for dictionaries [8]. Compared to the work presented in this thesis, it uses a hardware level message passing interface (MPI).

Several other publications propose B-trees to build distributed systems. Boxwood [75] uses a distributed B-tree to implement a file system. The tree operations are coordinated by a distributed lock service. Hyder [18] implements an index structure based on a binary tree on a shared flash log. Similar to Hyder, Tango [13] generalizes distributed data structures on append only logs. Both use the log for transaction control and append a new version of the changed index to the log.

HyperDex [46] and Yesquel [5] are two systems that are the most related to the work proposed here. HyperDex implements a partitioned key-value store which allows efficient search functions and secondary indexes based on a novel multi-dimensional hash function. Yesquel implements a distributed B-tree and proposes several optimizations to use the tree for a distributed SQL database. The architecture and concurrency control used in Yesquel is very similar to Sinfonias's mini-transactions. Both systems implement a rich API. However, compared to DMap, their interfaces are not compatible to existing well-know Java interfaces.

Distributed databases. The idea of running multiple independent query processors on a distributed data store is not new. MoSql [107] implements a distributed storage engine for the MySQL database. Compared to H2/DMap, it uses deferred update replication to certify concurrent transaction before commit. Yesquel [5] also replaces the local B-tree implementation of SQlite with their distributed balanced tree. F1 [102] is a distributed SQL database which drives the Google ad-words business. The storage engine used by F1 is Spanner [36].

Chapter 6

Conclusion

The rise of worldwide Internet-scale services demands large distributed systems. Indeed, when handling several millions of users, it is common to operate thousands of servers spread across the globe. Here, replication plays a central role, as it contributes to improve the user experience by hiding failures and by providing acceptable latency. In this thesis, we claim that atomic multicast, with strong and well-defined properties, is the appropriate abstraction to efficiently design and implement globally scalable distributed systems.

In this thesis is, we contend that instead of building a partial order on requests using an ad hoc protocol intertwined with the application code, services have much to gain from relying on a middleware to partially order requests. Moreover, such a middleware must include support for service recovery and add dynamic reconfiguration, both non trivial requirements which should be abstracted from the application code. For that reason, application developers should only be exposed to strong consistent geo-distributed data structures as building blocks instead of directly implementing low-level coordination protocols.

6.1 Research assessment

The research conducted within this dissertation provides three major contributions: (i) We have shown that atomic multicast is a suitable abstraction to build global and scalable systems. (ii) We could demonstrate how Elastic Paxos can be used to dynamically reconfigure atomic multicast, which let a replicated data store be repartitioned without service interruption. (iii) We could show how a distributed data structure middleware based on Elastic Paxos (DMap) can be used to reliable distribute any Java application, like a full transactional database.

URingPaxos. With an efficient implementation of an atomic multicast algorithm, we could demonstrate the capabilities of such an abstraction to support at the same time *scalability* and *strong consistency* in the context of large-scale online services. URingPaxos scales not only in local-area environments up to the maximum network line speed of 10 Gbit/s but also on globally distributed wide area networks. It allows to build scalable partial ordered data stores. Further, we could contribute a novel technique to recover atomic multicast, even under full system load, and proposed new mechanism to reduce latency of global commands.

Elastic Paxos. In today's cloud environments, adding resources to and removing resources from an operational system without shutting it down is a desirable feature. As we could show, atomic multicast is a suitable abstraction to build scalable distributed systems. But atomic multicast, as discussed above, relies on static subscriptions of replicas to groups, that is, subscriptions are defined at initialization and can only be changed by stopping all processes, redefining the subscriptions, and restarting the system. Therefore, we designed a dynamic atomic multicast algorithm, Elastic Paxos. We could show how Elastic Paxos can be used to dynamically subscribe replicas to a new multicast group (i.e., a new partition), which let a replicated data store be repartitioned without service interruption. Further, we demonstrate how dynamic subscriptions offer an alternative approach to reconfiguring a Paxos replicated state machine.

DMap. Scalable state machine replication has been shown to be a useful technique to solve the challenges in building reliable distributed data stores. However, implementing a fully functional system, starting from the atomic multicast primitives, supporting required features like recovery or dynamic behavior is a challenging and error-prone task. Providing higher-level abstractions in the form of distributed data structures can hide this complexity from system developers. For that reason, we proposed that system developers can gain much from distributed data structures, instead of implementing low level abstractions. We implemented DMap, a lock-free concurrent ordered map, supporting dynamic re-partitioning and recovery, exposed as a well known Java interface. Different to existing distributed data structures, which often rely on transactions or distributed locking to allow concurrent access, DMap relies on Elastic Paxos to partially order all operations. Further, it supports global consistent snapshots which allows long-running read operations for background data analytics. Finally, to strengthen our claims, we ported a transactional database on top of DMap.

6.2 Future directions

Latency in globally distributed environments. While we could show that atomic multicast is a suitable abstraction to build global distributed data stores, our algorithms could be improved in terms of latency in global distributed environments. Mult-Ring Paxos and Elastic Paxos are designed to maximize throughput. The underlying ring topology, however, introduces additional latency. While this effect can be neglected in local-area networks, spawning instances around the globe causes unnecessary latencies. One direction of future work could be to replace the ring topology for global commands with latency optimal algorithms [69].

Atomic Multicast in other research domains. Much of the studies in this thesis focus on distributed data stores. There are however many other fields of research and practical systems which could benefit from strong ordering guaranties and scalability of atomic multicast. One such field could be for example distributed composite event detection. Many industries are confronted with the challenges of pervasive sensor data (e.g., internet of things). But when it comes to their analysis, many platforms are still implemented in a centralized manner and batch processing principles (e.g., MapReduce jobs). Such systems assume that one can store all data in memory or on disk and filter the relevant part afterward. Once the incoming traffic to store grows, it is more efficient to implement such systems in a distributed control loop.

Atomic multicast could be used to distribute composite event detection. The execution of the pattern matching should be close to the event sources to handle the ever-increasing amount of data. Existing solutions differ in the expressiveness of their languages. One approach is to run complex queries on top of aggregated data windows, another is to let data change the states in an automaton. Query data windows let the language allow data aggregation and joins. Esper is a good example of such an implementation. The drawback of this approach is the scalability, since all implementations we know of use a single-server approach. Detection based on finite state automaton seems to be well studied [93] [100]. While other works to detect patterns also exist [37], the approaches to scale are similar. Pietzbuch [93] split the automaton into sup-expressions and deploy the execution of them to different servers. Sub-expressions may be reused in other queries. This requires a clever placement of the sub-automaton to have minimal client latency and maximal automaton reusing. Cugola [37] splits the execution of more trivial expressions along a hierarchically shortest path tree of all message brokers.

Appendix A

URingPaxos Library

URingPaxos is a is publicly available library [1] developed in the contect of this thesis and at the core of Chapters 3 through 5.

Since URingPaxos has been used in research projects and publications outside of the scope of this thesis [89, 78, 21, 88, 12, 83, 84], this Appendix will describe the implementation of the core library in more details.

The complete library presented in this thesis is written in Java and has approximately 25000 lines of code. Some performance critical storage parts, however, are implemented in C, using the Java native interface (JNI). The decision to use Java instead of C or C++ was mainly due to the better code readability and maintainability of Java. While C is fast, the benefits that Java brings, for example, in the collections and concurrency frameworks, outweigh the small performance penalty we have to pay.

A.1 Core Algorithm

The core algorithm implemented in URingPaxos is a unicast version of Ring Paxos [82]. Scalability is achieved by combining multiple rings with Multi-Ring Paxos or Elastic Paxos. The following description will point out some important details of the Ring Paxos core.

A.1.1 Proposer

A proposer is one of the simplest roles in the Paxos algorithm. In its original form, it can propose a command (value) by executing phase 1 followed by phase 2. As

[1] `https://github.com/sambenz/URingPaxos`

soon as we have more than one proposer in the system, this could cause liveness problems in the algorithm [69]. An optimized proposer will send the commands to a coordinator (i.e., leader).

While in Paxos the values can be sent through multicast to the coordinator, in Ring Paxos, the commands are forwarded along the ring until they reach the leader (Figure 2.2). Ring Paxos introduces a new message type for this purpose.

Proposers are usually also learners. This is required to detect if an actual command has really got learned or to throttle down the generated load. While in Ring Paxos all decisions are forwarded to all participants, proposers have to subscribe to the multicast group of the learners to receive phase 2b messages.

In this Ring Paxos implementation a proposer has two operation modes. An internal one which reads commands from standard input and an external one which can be used to directly embed the proposer in an application. Embedded values can be proposed using the following *propose()* method:

```
public FutureDecision propose(byte[] b);
```

The argument is a byte array. This command or value is wrapped in a *Value* object. A *Value* object has the actual byte array and a unique identifier (ID), which is generated in the proposer. This ID enables indirect consensus [45] and opens the possibility to remove the byte array from following messages. For performance reasons, this ID is not a real UUID but it is a combination of time in nano seconds and the unique ring position of the proposer.

The return type of *propose()* is a *FutureDecision*. A future decision will contain eventually a *Decision*, once the byte array is learned. It contains also a *CountDownLatch* on which the calling thread can wait until the decision is set. By waiting on this lock, the caller can implement a blocking call, while the overall nature of propose is still non-blocking. This feature is for example used to measure the latency of a proposed value.

A.1.2 Coordinator

The coordinator is the process that starts phase 1. It is elected out of the group of acceptors. While the algorithm can tolerate multiple coordinators and still guarantee safety, it is more efficient when a single leader exists. Coordinator election is done with Apache Zookeeper. The first acceptor in the ring acts as the coordinator.

After a coordinator is elected, it starts a new thread which is responsible for phase 1. While Paxos has a dedicated phase 1a/b message, Ring Paxos uses only one message with an additional vote count (Figure A.2). Whenever a coordinator successfully reserves a ballot for an instance, it generates a *Promise* object which contains an instance number and a corresponding ballot and stores it in a *BlockingQueue*. While the instance numbers are continuously increasing, the ballot numbers are composed out of a counter and the last digit of the coordinator ID. This guarantees that ballot numbers are always unique in the whole system. If the proposed ballot for an instance is smaller than what the acceptors already promised to accept, then they will answer with a *nack* message or by sending nothing (timeout at coordinator). If the instance was already decided, the acceptor will respond with a *Value*. In this case, the coordinator must re-propose the *Value* with a higher ballot by starting phase 2.

The promise queue is not bounded to a fixed value of instances. A thread ensures that at any time the *Promises* in the queue are more than half of the $p1\_preexecution\_number$. The reservation of multiple instances can be done in one message. This implementation, however, is conservative: Only when the smallest instance number in such a range message is higher than the highest instance the acceptors have ever seen, the range message is accepted. Otherwise, URingPaxos falls back to a standard phase 1 messages for every instance.

When a coordinator receives a *Value* to propose, it starts executing phase 2. First it takes the next *Promise* from the queue and generates a *Proposal*. A *Proposal* is mainly a *Value* and a timestamp. The later is used to detect timeouts while proposing. Phase 2 is started by composing the *Promise* and the *Proposal* into a phase 2 message. Further, the coordinator keeps track of what is decided to remove the *Proposal* from an internal map or re-propose the same *Value* with a new *Promise*.

In the case of URingPaxos or Elastic Paxos, the coordinator has also the additional task to measure the ring throughput. The throughput per time interval is compared to a global maximum of λ messages. Whenever the actual throughput is smaller than λ, additional skip messages will be sent. Skip messages are special values, but normal Paxos instances. They are proposed and learned like every other value in Paxos, but the content is only evaluated in the *MultiRingLeaner*s or *ElasticLearner*s.

A.1.3 Acceptors

The most complex part of Paxos is implemented in the acceptors. Every acceptor has to keep track of which ballot it has promised to accept. Further, once a value

is learned, it creates a *Decision* object that contains the instance, the ballot and the *Value*. This *Decision* is persisted over the *StableStorage* interface (Section A.4). *Decision*s can be updated, but only the ballot. Once a *Value* is learned, it can't be changed anymore.

A URingPaxos acceptor uses a hash map to keep track of what ballot it has promised to accept in which instance. Further, it uses another hash map to cache the byte arrays from the *Value*s and the *StableStorage* interface for all *Decision*s. Instead of sending back a 2b message to the proposer, a URingPaxos acceptor increases the vote count in the message. The last acceptor in the ring checks if the vote count is larger than or equal to a predefined quorum. If so, it will generate a decision message which is forwarded along the ring up to the predecessor of the last acceptor.

Another optimization in URingPaxos is that every byte array of a *Value* only gets transmitted once over the network. Depending on which *Role* a *Node* in the ring has and at which position it is located, a phase 2 or decision message may not contain the content (byte array). The *Value* object with the ID, however, is always present and can be used to store and look-up the content from a map.

A.1.4 Learners

Learners in the URingPaxos implementation are interested in the decision messages. Like the acceptors, also the learners have to cache the content of the *Value*s, since the decision messages not always contain them.

While a simplistic learner delivers the values at the receipt, we usually want the learner to deliver the values in the order of the increasing instances instead. But this could block a learner in the case of an outstanding or missing instance. For that reason, a learner must also implement a proposer part to "ask" for such missing instances. In this implementation, a learner proposes a *null* value for an outstanding instance by starting phase 1 with a very high ballot number. This approach, described by [69], will decide the instance to *null* if it was undecided or return the previously learned value.

Like the proposers, the learners can also run in a "service mode". A learner service will not write the decisions to the logging mechanism but rather store them in a *BlockingLinkedList* for further use. One application of this mechanism is a *MultiRingLeaner*.

A *MultiRingLeaner* is started if the *Node* is a learner in multiple rings. It is a wrapper around several independent learners and delivers *m* messages from every ring in a deterministic round-robin procedure. If one ring has no value to deliver, the multi-ring learner blocks on the *take()* method until some data are

available in this ring. To guarantee the progress even in the absence of traffic in one ring, the Multi-Ring Paxos algorithm introduces the concept of skip messages.

Skip messages are issued by the ring coordinator. As already mentioned, they are proposed and learned in normal Paxos instances. The values that these instances include, however, have the static ID of "Skip!". Whenever a multi-ring learner gets such a skip message from a queue, it will not deliver this instance, but interpret the content of the value to figure out how many values it must skip. As described in Figure 2.3, these skip messages are not delivered but used to deliver values from all rings in a constant speed.

A.2 Ring Management

The heart of URingPaxos is the ring management. It is responsible for creating the network connections, looking up configurations and providing a dynamic view of the current ring. The main entity is a *Node*. A *Node* is the class that implements *main()*. It is started with several command-line arguments to define a *List* of *RingDescription*s.

A.2.1 Abstract Role

For every *RingDescription*, the *Node* initializes a *RingManager*, which holds an Apache Zookeeper connection. Further, it starts the threads for all Paxos roles described in the previous section, and registers them in the ring, using the *RingManager*. Every role has a concrete implementation which extends the abstract class *Role*:

```
public abstract class Role implements Runnable {}
   public void deliver(RingManager fromRing,Message m);
}
```

The *RingManager* is passed to the constructor of every *Role*. This allows them to look up the responsible *NetworkManager* and register themselves for message delivery. The network will invoke the *deliver()* callback for the messages targeted to the specific role.

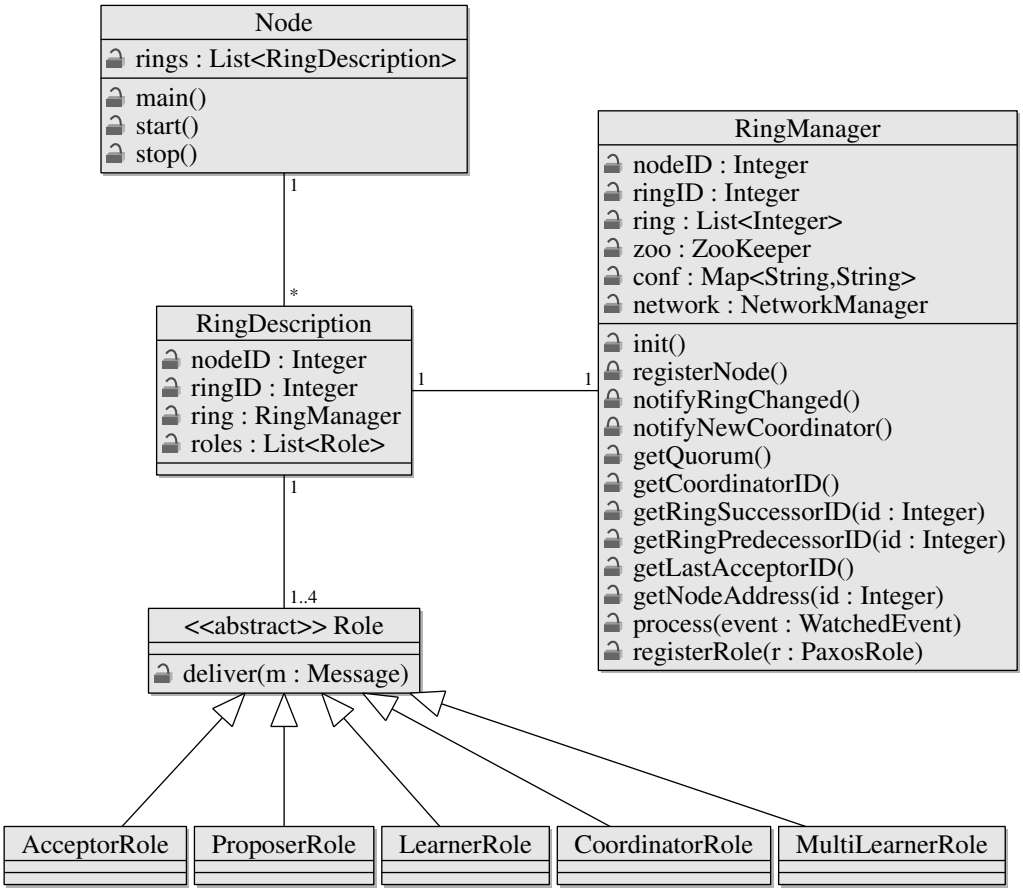

Figure A.1. Class diagram of the ring management

A.2.2 RingManager

The *RingManager* is the central component of the system. It spawns the *NetworkManager* to automatically open and close the required network connections to the ring successor, and is always informed about all changes on the ring. Moreover, it is available to all *Role*s and it is responsible for starting and stopping the coordinator process. Figure A.1 shows the class diagram with these dependencies.

During the initialization, the *RingManger* registers the node ID in the given ring on Zookeeper. The *RingManager* is the only component in the system that interacts with Zookeeper. It will also publish its IP address and its randomly chosen TCP port as additional data with the ID. Further, it looks up the configurations map, also stored in the Zookeeper directory. After the initialization, the *RingManager* will have learned the ring topology and it will automatically open a network connection to the ring successor. The ring is monitored by a Zookeeper *Watcher*.

Whenever the successor changes, the old network connection is closed and the new one will be opened. If a node in the ring crashes unexpectedly, Zookeeper will detect this and inform its manager after a timeout of some seconds.

Coordinator election is done in the same *Watcher*. The coordinator is the acceptor that has the lowest ID. The required thread with the corresponding role is also started by the *RingManager*.

Every node in the ring can look up the IP address and port of every other node with Zookeeper. The nodes get the addresses by parsing the local network interface configurations. For the Amazon EC2 deployment the IP is taken from an environment variable.

A.2.3 Failures and recovery

Nodes that fail by crashing are detected by the Zookeeper instance and removed from the ring. The changes are propagated to all other nodes. If required, the network connections are rebuilt. If the failing node was the coordinator, a new coordinator will be elected and started.

The same is true for process recovery. Whenever a new node joins the ring, its ID defines the position in the ring. The new node will automatically be included by the existing nodes.

While the *RingManager* is only responsible for the ring management, the roles must implement the required recovery methods. Acceptors are able to recover correctly if they use stable storage (Section A.4). Learners can also be configured to recover. If so, they will start learning from the first instance or the last acceptor log trim point.

A.3 Network communication

Networking is a very important component in the implementation since the goal of Ring Paxos and Multi-Ring Paxos is to achieve maximum throughput and low latency. Such requirements are hard to achieve by using existing frameworks (e.g., RPC stacks). They usually over-abstract the important features to make them simple. If we want to get the maximum out of the underlying hardware, careful tweaking of all possible parameters is required.

A.3.1 Transport

One of the goals of this thesis is the implementation of a unicast Multi-Ring Paxos (URingPaxos). With this version, which doesn't use IP multicast, it is possible to experiment on WAN links. For the transport layer we want to have a reliable message oriented protocol such as the Stream Control Transmission Protocol (SCTP). This relatively new protocol combines the message-oriented principles from UDP with the reliability and congestion control of TCP. Since this protocol seems to be perfect for this purpose, the first implementation was built on top of it. The throughput test on the cluster has shown that SCTP is significantly slower than TCP. For this reason the implementation switched entirely to TCP. The reason why SCTP is slow is not clear, one reason could be that it has not been as much optimized as TCP. TCP however is being optimized for more than twenty years.

TCP is stream oriented but Paxos is entirely message based. The use of TCP implies implementing our own message framing. In the current version, a length-prefix framing is used. This prepends the length in bytes of the following message (frame). The receiver will read the first two bytes interpreted as length, following the message itself. While this approach is very simple and works well in practice, it has the drawback that once a receive buffer is out of synchronization, it will not find the frame borders anymore. For that reason an additional magic-number preamble is sent to re-synchronize the frame handling.

The current transport layer lets itself configure the TCP nodelay option (disable Nagle's algorithm) and the TCP send and receive windows.

While the first implementation of TCP used standard (blocking) Java IO, the current implementation uses the non-blocking new Java IO (NIO). The reason to switch was not the demand for a scalable non-blocking API. But the NIO directly exposes DMA mapped *ByteBuffer*s instead of simple byte arrays, like in standard IO. This reduces the number of copies involved in transferring data, which improves speed.

A.3.2 Serialization

A critical piece to the overall protocol performance is the efficiency of serialization. Serialization is the conversion of a high-level abstraction of a message to its byte representation on the network line interface. While in C, de-serializing bytes received over the network is a simply cast to the corresponding message struct, in Java we have to re-create all required objects. It was an early design decision that the Java code will always operate on well-defined objects and never directly with byte arrays.

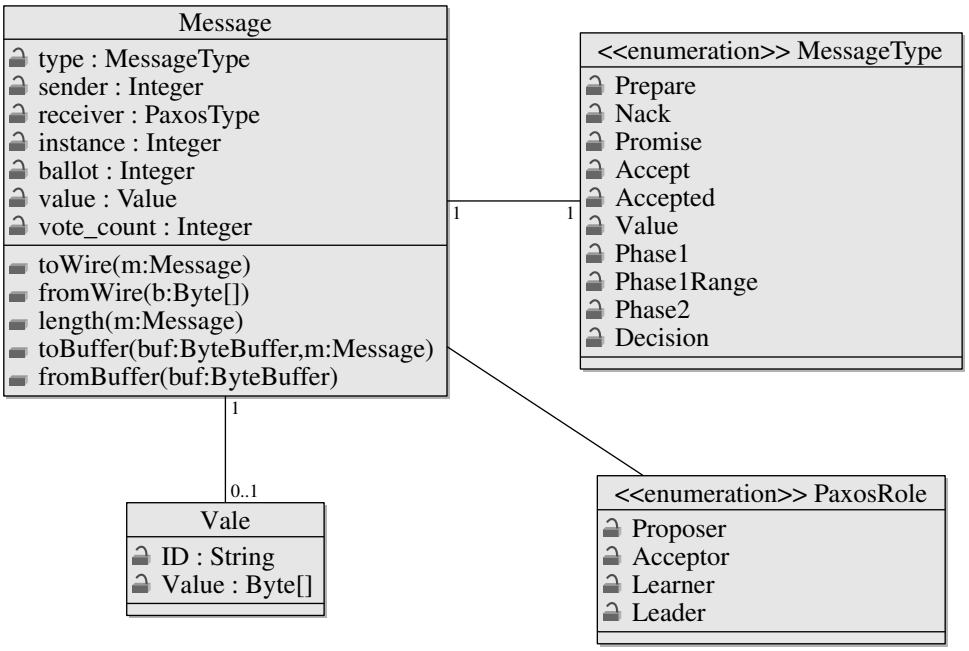

Figure A.2. Class diagram of *Message*

The classes which are responsible for messaging are defined in Figure A.2. The actual serialization part are the static methods *fromWire()* and *toWire()* in the *Message* class. The implementation of these two methods in the initial version of the prototype was done with Java object serialization (*ObjectInputStream*). This proved too slow for a high speed implementation.

The next, very modular and interoperable approach, was done using Google's protobuf[2] protocol buffers. Protobuf is a framework for message serialization. It contains an interface-definition language and a compiler. The compiler, which is available for different programming languages, will generate the required bindings.

A long test run showed that the protocol was not as fast as expected, even after several modifications everywhere in the code were conduced. Finally, we decided to remove protobuf and use our own object structure for the messages, which does not copy the objects before serialization. This re-copying of the data caused before a very high object creation rate which results in a increased cumulative garbage collection time which is shown in Figure A.3. Figure A.4 shows the protobuf implementation in red, which allocates almost 400 MByte/s objects. Since the line speed is 1 Gbit/s, an optimal allocation rate should be around 125

[2] `https://developers.google.com/protocol-buffers`

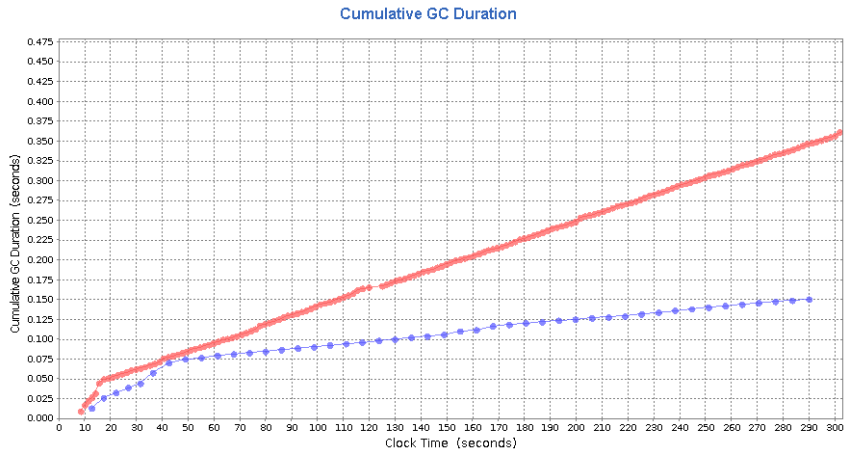

Figure A.3. Cumulative garbage collection time in seconds for protobuf (red) and direct serialization (blue)

MByte/s. This was achieved with a direct serialization (blue line). Direct serialization is implemented by copying every field of the *Message* object as byte to a *ByteBuffer*. This approach, together with the directly mapped network *ByteBuffer*s from Java NIO, resulted in an reasonable object creation rate.

A.3.3 NetworkManager

All the incoming and outgoing TCP connections are handled by the *NetworkManager*. The *NetworkManager* is provisioned by the *RingManager*. Once the server is running, the incoming messages are dispatched to the different roles which had subscribed for delivery. Local messages, for example from a proposer to a coordinator running on the same node, are also sent through the *NetworkManager* manager. The manager takes care that such messages are never sent through the network.

Further, the *receive()* method inside the *NetworkManager*, called from a *TCPListener*, will first try to forward the message in the ring and then deliver the message locally. This saves some latency for messages that are not directly targeted to this node. Before forwarding, unneeded content will be removed from *Value*s. This guaranties that the content of a *Value* is only sent once to every node.

The structure of *NetworkManager* is shown in Figure A.5. *TCPSender* and *TCPListener* both run as threads. Messages are sent out through a *TransferQueue* which combines the *send()* method in the *NetworkManager* with the *socket.write()* in the *TCPSender*. The *TCPListener* creates a *SessionHandler* for every incoming

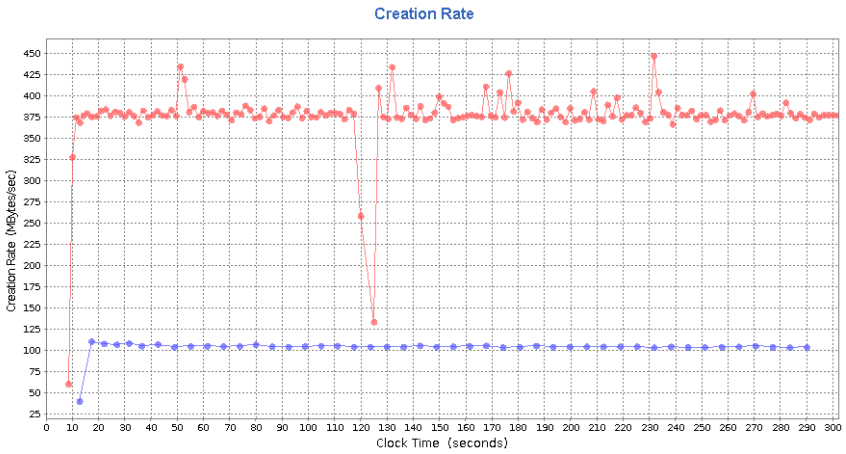

Figure A.4. Object creation rate in mbyte/s for protobuf (red) and direct serialization (blue)

connection and waits on a *Selector* for interruptions from the hardware. While in a NIO implementation this part would be typically done in a thread pool, this implementation is single-threaded. A thread pool is not required since we will always have only one open incoming connection.

A.4 Stable storage

Depending on the actual implementation of the *StableStorage* interface, an acceptor may or may not recover after a crash. Everything that implements this interface can be configured in the Zookeeper cluster as back-end. The class is loaded with *Class.forName(name).newInstance()* and must follow the definition below:

```
public interface StableStorage {
   public void putBallot(Long instance, int ballot);
   public int getBallot(Long instance);
   public boolean containsBallot(Long instance);
   public void putDecision(Long instance, Decision decision);
   public Decision getDecision(Long instance);
   public boolean containsDecision(Long instance);
   public boolean trim(Long instance);
   public Long getLastTrimInstance();
   public void close();
```

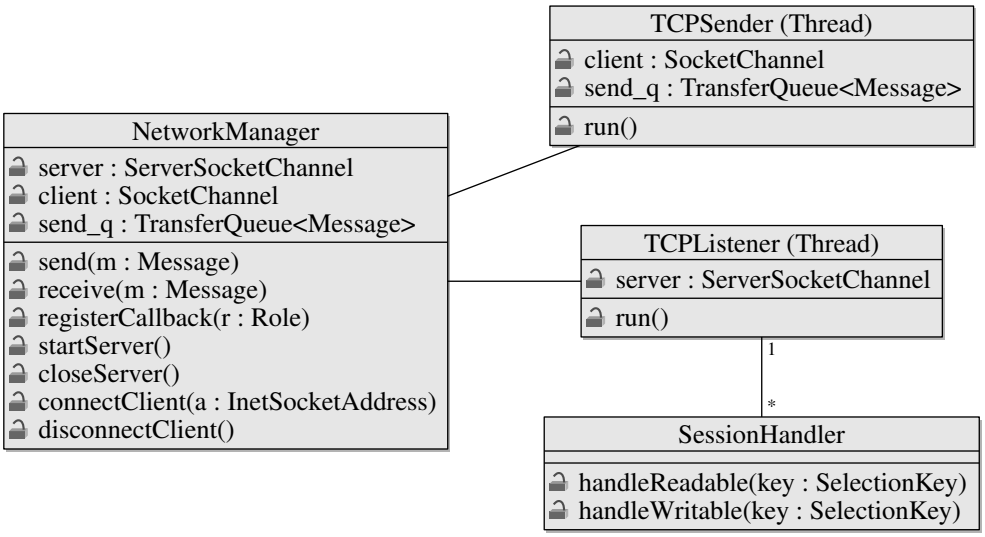

Figure A.5. Class diagram of the network management

}

The current implementation includes the following storage back-ends:

NoMemory: This implementation is for testing purpose only. Without storage, acceptors are not able to answer a single missing value. It is provided to measure the raw throughput without impact of garbage collection or disk writes.

InMemory: The *InMemory* implementation uses a *LinkedHashMap* to keep up to 15k *Decisions* in memory. Unfortunately, the overall throughput is very poor (550 Mbit/s). Even with the new garbage collector G1 the throughput can only reach 800 Mbit/s. This is more than 200 Mbit/s slower than the *NoMemory* implementation.

CyclicArray: *CyclicArray* is a array implementation in C. It uses JNI to provide a storage for up to 15k *Decisions*, not on the Java heap. This implementation has the same speed as *NoMemory*. Since the objects are stored outside the heap, no garbage collection is required while overwriting old entries.

BufferArray: *BufferArray* holds an array of 15k prealocated ByteBuffer. It is not as fast a the CyclicArray, but achives similar performance with less configuration (compiled JNI) overhead.

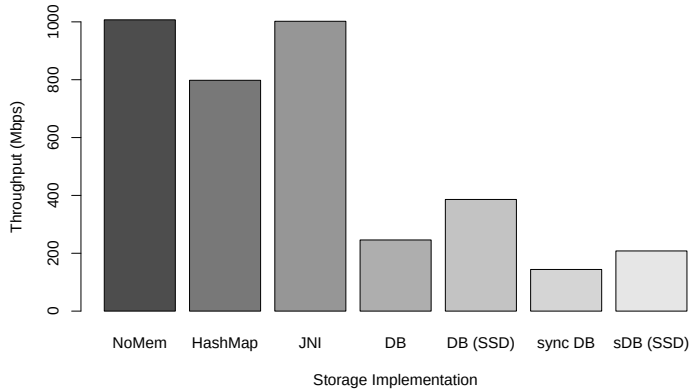

Figure A.6. Performance comparison of different *StableStorage* implementations.

BerkeleyStorage: *BerkeleyStorage* is a key-value store used to provide stable storage on spinning hard disks or solid state disks (SSD). The database uses deferred writes, which means that we can tolerate the crash of a Paxos process but not of the whole machine.

SyncBerkeleyStorage: *SyncBerkeleyStorage* is a wrapper of *BerkeleyStorage* which enables synchronous disk writes. This is the safest configuration, since it can tolerate the crash of a whole machine.

A comparison of the different implementations deployed in a local-area network is provided in the Figure A.6.

Bibliography

[1] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The Totem Multiple-ring Ordering and Topology Maintenance Protocol. *ACM*, May 1998.

[2] M. K. Aguilera, W. Chen, and S. Toueg. Failure Detection and Consensus in the Crash-Recovery Model. In *Proceedings of the International Symposium on Distributed Computing (DISC'98)*, pages 231–245, September 1998.

[3] M. K. Aguilera, W. Golab, and M. A. Shah. A practical scalable distributed b-tree. *Proceedings of the VLDB Endowment*, 1(1):598–609, 2008.

[4] M. K. Aguilera, I. Keidar, D. Malkhi, and Al. Shraer. Dynamic atomic storage without consensus. *Journal of the ACM*, 58(2):7, 2011.

[5] M. K. Aguilera, J. B. Leners, and M. Walfish. Yesquel: scalable SQL storage for Web applications. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 245–262. ACM, 2015.

[6] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS OSR*, volume 41, pages 159–174. ACM, 2007.

[7] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *PODC*, 2000.

[8] S. Alam, H. Kamal, and A. Wagner. A scalable distributed skip list for range queries. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*, pages 315–318. ACM, 2014.

[9] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread Toolkit: Architecture and Performance. Technical report, Johns Hopkins University, 2004. CNDS-2004-1.

[10] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *FTCS*, 1992.

[11] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.

[12] A. Babay and Y. Amir. Fast total ordering for modern data centers. In *Distributed Computing Systems (ICDCS), 2016 IEEE 36th International Conference on*, pages 669–679. IEEE, 2016.

[13] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *SOSP*, 2013.

[14] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato. Building global and scalable systems with Atomic Multicast. In *Middleware*, 2014.

[15] S. Benz, L. Pacheco de Sousa, and F. Pedone. Stretching Multi-Ring Paxos. In *ACM SAC*, 2015.

[16] S. Benz and F. Pedone. Elastic Paxos: A Dynamic Atomic Multicast Protocol. In *ICDCS*, 2017.

[17] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*, volume 370. Addison-wesley New York, 1987.

[18] P. A. Bernstein, C. W. Reid, and S. Das. Hyder-A Transactional Record Manager for Shared Flash. In *CIDR*, volume 11, pages 9–20, 2011.

[19] A. Bessani, M. Santos, J. Felix, N. Neves, and M. Correia. On the Efficiency of Durable State Machine Replication. In *ATC*, 2013.

[20] A. Bessani, J. Sousa, and E. EP Alchieri. State machine replication for the masses with BFT-SMaRt. In *DSN '14*. IEEE, 2014.

[21] C. E. Bezerra, F. Pedone, and R. van Renesse. Scalable State-Machine Replication. In *DSN*, 2014.

[22] K. P. Birman and R. Cooper. The Isis project: Real experience with a fault tolerant programming system. In *ACM SIGOPS*, 1990.

[23] K. P. Birman and T. A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems (TOCS)*, 5(1):47–76, February 1987.

[24] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al. Tao: Facebook distributed data store for the social graph. In *ATC*, 2013.

[25] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI*, 2006.

[26] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *OSDI*, 1999.

[27] M. Castro, R. Rodrigues, and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. *ACM Transactions on Computer Systems (TOCS)*, 21(3):236–269, 2003.

[28] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[29] K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[30] B. Charron-Bost, F. Pedone, and A. Schiper, editors. *Replication: Theory and Practise*. Springer-Verlag, 2010.

[31] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[32] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. UpRight cluster services. In *SOSP*, 2009.

[33] A. Clement, E. L. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine Fault Tolerant Systems Tolerate Byzantine Faults. In *NSDI*, 2009.

[34] V. Cogo, A. Nogueira, J. Sousa, M. Pasin, H. P. Reiser, and A. Bessani. Fitch: supporting adaptive replicated services in the cloud. In *DAIS '13*. Springer, 2013.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, 2010.

[36] J.C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, JJ Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's Globally-Distributed Database. In *OSDI*, 2012.

[37] G. Cugola and A. Margara. Raced: an adaptive middleware for complex event detection. In *Proceedings of the 8th International Workshop on Adaptive and Reflective MIddleware*, ARM '09, pages 5:1–5:6. ACM, 2009.

[38] M. Kapritsos Dahlin, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*, 2012.

[39] R. de Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. *Theoretical Computer Science*, 243(1–2):35–91, 2000.

[40] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, volume 41, pages 205–220, 2007.

[41] X. Défago, A. Schiper, and P. Urbán. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *ACM Computing Surveys,*, 36(4):372–421, December 2004.

[42] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *OPODIS*, 2000.

[43] C. Du Mouza, W. Litwin, and P. Rigaux. Sd-rtree: A scalable distributed rtree. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 296–305. IEEE, 2007.

[44] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[45] R. Ekwall and A. Schiper. Solving atomic broadcast with indirect consensus. In *Dependable Systems and Networks, 2006. DSN 2006. International Conference on*, pages 156–165. IEEE, 2006.

[46] R. Escriva, B. Wong, and E. G. Sirer. HyperDex: A distributed, searchable key-value store. *Acm sigcomm computer communication review*, 42(4):25–36, 2012.

[47] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Foundations of Computation Theory*, pages 127–140. Springer, 1983.

[48] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[49] R. Friedman and R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *HPDC*, 1997.

[50] U. Fritzke, Ph. Ingels, A. Mostéfaoui, and M. Raynal. Fault-Tolerant Total Order Multicast to Asynchronous Groups. In *SRDS*, 1998.

[51] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[52] S. Gilbert and N. Lynch. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services. *SIGACT News*, (2):51–59, June 2002.

[53] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

[54] V. Gramoli, L. Bass, A. Fekete, and D. W. Sun. Rollup: Non-disruptive rolling upgrade with fast consensus-based dynamic reconfigurations. *IEEE Transactions on Parallel and Distributed Systems*, 2016.

[55] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Distributed Computing Systems, 1997., Proceedings of the 17th International Conference on*, pages 578–585. IEEE, 1997.

[56] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, 2001.

[57] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, chapter 5. Addison-Wesley, 2nd edition, 1993.

[58] J. H. Hartman and J. K. Ousterhout. The Zebra striped network file system. *ACM Transactions on Computer Systems (TOCS)*, 13(3):274–310, 1995.

[59] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990.

[60] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *ATC*, 2010.

[61] R. Jain. *The art of computer systems performance analysis : techniques for experimental design, measurement, simulation, and modeling*. John Wiley and Sons, Inc., New York, 1991.

[62] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the b-link tree. In *Parallel Processing Symposium, 1992. Proceedings., Sixth International*, pages 319–324. IEEE, 1992.

[63] F. P. Junqueira, I. Kelly, and B. Reed. Durability with bookkeeper. *ACM SIGOPS OSR*, 47(1):9–15, 2013.

[64] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2), 2008.

[65] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, et al. All about Eve: Execute-Verify Replication for Multi-Core Servers. In *OSDI*, volume 12, pages 237–250, 2012.

[66] R. Kotla and M. Dahlin. High Throughput Byzantine Fault Tolerance. In *DSN*, 2004.

[67] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[68] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9):690–691, 1979.

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

[70] L. Lamport. Paxos Made Simple. *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory)*, 32, 2001.

[71] L. Lamport, D. Malkhi, and L. Zhou. Stoppable paxos. *TechReport, Microsoft Research*, 2008.

[72] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and Primary-backup Replication. In *PODC*, PODC '09, pages 312–313, New York, NY, USA, 2009. ACM.

[73] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a State Machine. *SIGACT News*, 41(1):63–73, March 2010.

[74] J.R. Lorch, A. Adya, W.J. Bolosky, R. Chaiken, J.R. Douceur, and J. Howell. The SMART way to migrate replicated stateful services. *ACM SIGOPS OSR*, 40(4):103–115, 2006.

[75] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: Abstractions as the Foundation for Storage Infrastructure. In *OSDI*, volume 4, pages 8–8, 2004.

[76] D. Malkhi, M. Balakrishnan, J. D. Davis, V. Prabhakaran, and T. Wobber. From paxos to CORFU: a flash-speed shared log. *ACM SIGOPS OSR*, 46(1):47–51, 2012.

[77] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *OSDI*, 2008.

[78] P. J. Marandi, S. Benz, F. Pedone, and K. Birman. The performance of Paxos in the cloud. In *SRDS*, 2014.

[79] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking State-Machine Replication for Parallelism. In *ICDCS*, 2014.

[80] P. J. Marandi and F. Pedone. Optimistic Parallel State-Machine Replication. In *SRDS*, 2014.

[81] P. J. Marandi, M. Primi, and F. Pedone. Multi-Ring Paxos. In *DSN*, 2012.

[82] P. J. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *DSN*, 2010.

[83] O. M. Mendizabal, R. ST. De Moura, F. L. Dotti, and F. Pedone. Efficient and deterministic scheduling for parallel state machine replication. In *31st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2017.

[84] O. M. Mendizabal, F. L. Dotti, and F. Pedone. High performance recovery for parallel state machine replication. In *ICDCS*. IEEE, 2017.

[85] I. Moraru, D. G. Andersen, and M. Kaminsky. Egalitarian Paxos. In *SOSP*, 2012.

[86] A. Nogueira, A. Casimiro, and A. Bessani. Elastic state machine replication. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[87] M. T. Özsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 2nd edition, 1999.

[88] L. Pacheco, R. Halalai, V. Schiavoni, F. Pedone, E. Riviere, and P. Felber. Globalfs: A strongly consistent multi-site file system. In *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*, pages 147–156. IEEE, 2016.

[89] L. Pacheco, D. Sciascia, and F. Pedone. Parallel deferred update replication. In *Network Computing and Applications (NCA), 2014 IEEE 13th International Symposium on*, pages 205–212. IEEE, 2014.

[90] R. Padilha and F. Pedone. Augustus: Scalable and Robust Storage for Cloud Applications. In *Eurosys*, 2013.

[91] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting Atomic Broadcast in Replicated Databases. In *EuroPar*, 1998.

[92] F. Pedone, R. Guerraoui, and A. Schiper. The Database State Machine Approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1), 2002.

[93] P. R. Pietzuch, B. Shand, and J. Bacon. A framework for event composition in distributed systems. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, Middleware '03, pages 62–82. Springer-Verlag New York, Inc., 2003.

[94] J. Rao, E. J. Shekita, and S. Tata. Using Paxos to build a scalable, consistent, and highly available datastore. *Proceedings of the VLDB Endowment*, 4(4):243–254, 2011.

[95] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *ICCCN*, 1998.

[96] A. Schiper and S. Toueg. From set membership to group membership: A separation of concerns. *Dependable and Secure Computing, IEEE Transactions on*, 3(1):2–12, 2006.

[97] N. Schiper and F. Pedone. On the Inherent Cost of Atomic Broadcast and Multicast Algorithms in Wide Area Networks. In *ICDCN*, 2008.

[98] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS*, 2010.

[99] F. B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[100] N. P. Schultz-Møller, M. Migliavacca, and P. R. Pietzuch. Distributed complex event processing with query rewriting. In *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*, DEBS '09, pages 4:1–4:12. ACM, 2009.

[101] D. Sciascia, F. Pedone, and F. Junqueira. Scalable Deferred Update Replication. In *DSN*, 2012.

[102] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed SQL database that scales. *Proceedings of the VLDB Endowment*, 6(11):1068–1079, 2013.

[103] A. Singh, P. Fonseca, P. Kuznetsov, R. Rodrigues, P. Maniatis, et al. Zeno: Eventually Consistent Byzantine-Fault Tolerance. In *NSDI*, 2009.

[104] J. Sousa and A. Bessani. Separating the wheat from the chaff: An empirical design for geo-replicated state machines. In *SRDS '15*. IEEE, 2015.

[105] B. Sowell, W. Golab, and M. A. Shah. Minuet: a scalable distributed multiversion B-tree. *Proceedings of the VLDB Endowment*, 5(9):884–895, 2012.

[106] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[107] A. Tomic, D. Sciascia, and F. Pedone. MoSQL: An Elastic Storage Engine for MySQL. In *SAC*, 2013.

[108] S. Toueg. Randomized byzantine agreements. In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, pages 163–178. ACM, 1984.

[109] P. Unterbrunner, G. Alonso, and D. Kossmann. E-Cast: Elastic Multicast. Technical report, ETH Zurich, Department of Computer Science, 2011.

[110] R. van Renesse and F. B. Schneider. Chain Replication for Supporting High Throughput and Availability. In *OSDI*, pages 91–104, 2004.

[111] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn. Ceph: A scalable, high-performance distributed file system. In *OSDI*, 2006.