Scalable State Machine Replication

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
Carlos Eduardo Benevides Bezerra

under the supervision of
Fernando Pedone and Cláudio Geyer

February 2016

Dissertation Committee

Dissertation accepted on 16 February 2016

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.

Carlos Eduardo Benevides Bezerra
Lugano, 16 February 2016

To my parents and my mentors.
I wouldn’t be here without you.

Life was simple before World War II.
After that, we had systems.

Grace Hopper

Abstract

Redundancy provides fault-tolerance. A service can run on multiple servers that replicate each other, in order to provide service availability even in the case of crashes. A way to implement such a replicated service is by using techniques like state machine replication (SMR). SMR provides fault tolerance, while being linearizable, that is, clients cannot distinguish the behaviour of the replicated system to that of a single-site, unreplicated one. However, having a fully replicated, linearizable system comes at a cost, namely, scalability—by scalability we mean that adding servers will always increase the maximum system throughput, at least for some workloads. Even with a careful setup and using optimizations that avoid unnecessary redundant actions to be taken by servers, at some point the throughput of a system replicated with SMR cannot be increased by additional servers; in fact, adding replicas may even degrade performance. A way to achieve scalability is by partitioning the service state and then allowing partitions to work independently. Having a partitioned, yet linearizable and reasonably performant service is not trivial, and this is the topic of research addressed here.

To allow systems to scale, while at the same time ensuring linearizability, we propose and implement the following ideas: (i) Scalable State Machine Replication (S-SMR), (ii) Optimistic Atomic Multicast (Opt-amcast), and (iii) Fast S-SMR (Fast-SSMR). S-SMR is an execution model that allows the throughput of the system to scale linearly with the number of servers without sacrificing consistency. To provide faster responses for commands, we developed Opt-amcast, which allows messages to be delivered twice: one delivery guarantees atomic order (conservative delivery), while the other is fast, but not always guarantees atomic order (optimistic delivery). The implementation of Opt-amcast that we propose is called Ridge, a protocol that combines low latency with high throughput. Fast-SSMR is an extension of S-SMR that uses the optimistic delivery of Opt-amcast: while a command is atomically ordered, some precomputation can be done based on its fast, optimistically ordered delivery, improving response time.

Acknowledgements

It was truly humbling and inspiring to come across the people that I've met throughout this PhD course. To name a few people, Fernando Pedone, whose commitment to excellence was a great motivating factor to this work; Cláudio Geyer, who advised me from Brazil; Robbert van Renesse, who made core contributions to this thesis; Kenneth Birman, who welcomed me at Cornell; João Comba, who helped me write an award-winning paper in the very beginning of my PhD course; and Benoît Garbinato, who also made very important contributions to this work. Other people that I had the luck of working with were Daniel Cason and Parisa Marandi.

I had the pleasure of being part of a great research group in Lugano: Nicolas Schiper, Amir Malekpour, Ricardo Padilha, Daniele Sciascia, Marco Primi, Alex Tomic, Leandro Pacheco, Samuel Benz, Odorico Mendizabal, Tu Dang, Edson Camargo, Paulo Coelho, Long Le, and Enrique Fynn. I am also glad for having been in the same group as Vincent Rahli, Ittay Eyal and many others at Cornell.

But the PhD road is not a short nor an easy one, so having good friends makes things much easier. In special, Leonardo Damas, Dilermando, Cadu, Leonardo Alt. In the US, Brian Byrd, Jean Reis, Dipendra Misra, Anil Aksu, Rahmtin Rotabi, Larissa di Marzo, Heidi Katherina, Sarah Tan, Rachelle Ludwick and all the Cornell people who made that cold winter in Ithaca a bit warmer.

There are usually a few rough patches in every road, and some people who supported me when I most needed were my mother Isabel, my brother Ricardo, Flávia and Cíntia Eizerik, my good friends Jean, Dipendra, Leandro, Leonardo and Fynn, and my advisor in Switzerland, Fernando Pedone, who proved to be not only a great scientist who gives and expects the best, but also a wise friend you can rely on.

Finally, it is an honor to have Rui Oliveira, Benoît Garbinato, Mehdi Jazayeri and Nate Nystrom in my dissertation committee. Thank you for offering your insights on my research on replication and for joining me at this milestone in my career.

Truly, thank you, you all.

Preface

The result of this research appears in the following publications:

C. E. Bezerra, F. Pedone, R. van Renesse, and C. Geyer. Providing scalability and low latency in state machine replication. Technical Report USI-INF-TR-2015/06, Università della Svizzera italiana, 2015.

C. E. Bezerra, D. Cason, and F. Pedone. Ridge: high-throughput, low-latency atomic multicast. In *Proceedings of the 2015 IEEE 34th Symposium on Reliable Distributed Systems*, SRDS '15, pages 256–265. IEEE Computer Society, 2015.

C. E. Bezerra, F. Pedone, and R. van Renesse. Scalable state-machine replication. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 331–342. IEEE Computer Society, 2014.

C. E. Bezerra, F. Pedone, B. Garbinato, and C. Geyer. Optimistic atomic multicast. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 380–389. IEEE Computer Society, 2013.

The following papers were written in parallel with the writing of this thesis and are related to it, although they are outside the main scope of this work:

P. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 368–377. IEEE Computer Society, 2014.

C. E. Bezerra, J. Comba, and C. Geyer. Adaptive load-balancing for MMOG servers using KD-trees. *ACM Computers in Entertainment*, 10(1):5:1–5:16, 2012.

Contents

Contents **xi**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**
- 1.1 Context and goal of the thesis 2
- 1.2 Research contributions 3
- 1.3 Thesis outline 4

2 System Model and Definitions **5**
- 2.1 Processes and communication 5
  - 2.1.1 Reliable multicast 6
  - 2.1.2 Atomic multicast 6
- 2.2 Consensus 7
- 2.3 Linearizability 7

3 Replication **11**
- 3.1 Primary-backup 12
- 3.2 State machine replication 13
- 3.3 Chain replication 14
- 3.4 Multi-primary replication 16
- 3.5 Combining linearizability and scalability 16

4 Scalable State Machine Replication **19**
- 4.1 General idea 20
- 4.2 Detailed algorithm 22
- 4.3 Performance optimizations 24
  - 4.3.1 Caching 25

4.4 Correctness 26
4.5 Implementation 28
4.5.1 Eyrie 28
4.5.2 Volery 31
4.6 Performance evaluation 32
4.6.1 Environment setup and configuration parameters 32
4.6.2 Experiments using on-disk storage 34
4.6.3 Experiments using in-memory storage 36
4.6.4 Experiments with global commands 38
4.7 Related work 38
4.8 Conclusion 40

5 Optimistic Atomic Multicast 43
5.1 Problem definition 44
5.2 Paxos 45
5.3 Ridge 47
5.3.1 Overview of the protocol 47
5.3.2 Detailed algorithms 49
5.3.3 Tolerating failures 54
5.3.4 Optimistic deliveries 55
5.3.5 Correctness 57
5.4 Performance analysis 61
5.4.1 Performance model 61
5.4.2 Throughput 62
5.4.3 Latency 63
5.5 Experimental evaluation 63
5.5.1 Environment setup and configuration parameters 64
5.5.2 Broadcast: performance vs. number of destinations 65
5.5.3 Multicast: performance vs. number of groups 66
5.5.4 Results for optimistic deliveries 67
5.6 Related work 68
5.7 Conclusion 70

6 Fast Scalable State Machine Replication 71
6.1 Baseline Fast-SSMR 73
6.2 Detailed baseline algorithm 75
6.3 Optimistic state exchange 81
6.4 Correctness 82
6.5 Implementation 83

6.5.1 Extending Eyrie 83
6.5.2 Chirper 84
6.6 Performance evaluation 86
6.6.1 Environment setup and configuration parameters 87
6.6.2 Latency improvement 87
6.6.3 Throughput scalability 89
6.7 Related work 91
6.8 Conclusion 92

7 Conclusion **93**
7.1 Research assessment 94
7.2 Future directions 95

Bibliography **97**

Figures

2.1 Linearizable vs. non-linearizable executions. 8

3.1 Primary-backup. 13
3.2 State machine replication. 14
3.3 Chain replication. 15
3.4 Multi-primary (or deferred update) replication. 17

4.1 Atomic multicast and S-SMR. (To simplify the figure, we show a single replica per partition.) 22
4.2 Results for Zookeeper, ZKsmr and Volery with 1, 2, 4 and 8 partitions, using on-disk storage. Throughput was normalized by that of Volery with a single partition (absolute values in thousands of commands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput. 34
4.3 Cumulative distribution function (CDF) of latency for different command sizes (on-disk storage). 34
4.4 Results for Zookeeper, ZKsmr and Volery with 1, 2, 4 and 8 partitions, using in-memory storage. Throughput was normalized by that of Volery with a single partition (absolute values in thousands of commands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput. 36
4.5 Cumulative distribution function (CDF) of latency for different command sizes (in-memory storage). 36
4.6 Throughput and latency versus rate of create/delete commands (in-memory storage, 1000-bytes commands). Throughput is shown in units of a thousand commands per second (kcps). Latencies shown correspond to 75% of the maximum throughput. . 37
4.7 Cumulative distribution function (CDF) of latency for different rates of create/delete commands (in-memory storage, 1000-bytes commands). 37

5.1 Throughput (maximum) and latency (with the system at maximum power) as the number of destinations increases. In the right-side graphs, bars and dashes represent the 95th percentile and average latency values, respectively. 65
5.2 Throughput (maximum) and latency (with the system at maximum power) for each multicast protocol as the number of groups increases. The throughput of each protocol was normalized by the protocol's throughput with a single multicast group. In the right-side graphs, bars and dashes represent the 95th percentile and average latency values, respectively. 66
5.3 Latency, throughput and mistakes with the optimistic delivery enabled. 68

6.1 Invalid optimistic states propagated across partitions. 81
6.2 Latency components for different workloads, partitionings and levels of optimism. 88
6.3 Cumulative distribution function (CDF) of latency for different workloads, partitionings and levels of optimism. 88
6.4 Scalability results for Chirper, implemented with S-SMR and Fast-SSMR, with 1, 2, 4 and 8 partitions. Retwis results are also shown. For each workload, the throughput was normalized by that of Chirper with a single partition (absolute values in thousands of commands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput. 90

Tables

3.1 Replication techniques and their properties. 17

Chapter 1

Introduction

We can only see a short distance ahead, but we can see plenty there that needs to be done.

Alan Turing

In the last few years, we could witness the emergence of online services that work at scales never seen before. Not only has the scale increased, but also the demand of users for quality of the services provided. Such quality of service can be understood in terms of availability and speed: users want services that are always accessible and respond fast to requests. Naturally, as the number of users increase, one can expect the quality of service to deteriorate if the system implementing the service is not able to cope with higher scales, that is, if the system is not scalable.

Both availability and scalability can be increased by means of replication. By having multiple replicas running a service, if one of the replicas crashes, others may be still accessible to users, so the service stays available, despite hardware failure. Also, one could distribute the users among replicas, so that the workload would be divided among different machines, which would translate into better scalability. However, when implementing replication, it makes sense to keep replicas consistent; ideally, clients would not be able to tell that the system is replicated at all based on the answers to requests. For many current online services, the consistency requirements are very stringent. How to achieve such a high level of consistency (i.e., linearizability [35]), while providing fault-tolerance, scalability and fast replies, is the topic of this thesis.

1.1 Context and goal of the thesis

Replication has been vastly studied in the last few decades [22], resulting in numerous techniques that can be categorized according to the consistency level ensured and their scalability. What has been found is that there is a trade-off between consistency and performance, where performance means throughput, latency and scalability. For instance, one could deploy a service using deferred-update replication (DUR) [59], which has good scalability for requests that do not change the state of the service. Although DUR provides a consistency level lower than linearizability (namely, sequential consistency [8]), it is considered good enough for many applications.

Since this thesis focuses on strongly consistent services, we target state machine replication (SMR) [46, 64], a well-known technique to provide fault tolerance without sacrificing linearizability. SMR regulates how client commands are propagated to and executed by the replicas: every replica receives and executes every command in the same order. Moreover, command execution must be deterministic. SMR provides configurable availability, by setting the number of replicas, but limited scalability: every replica added to the system must execute all requests; hence, throughput does not increase as replicas join the system.

Distributed systems usually rely on state partitioning to scale (e.g., [19, 67]). If requests can be served simultaneously by multiple partitions, then augmenting the number of partitions results in an overall increase in system throughput. However, exploiting state partitioning in SMR is challenging: First, ensuring linearizability efficiently when state is partitioned is tricky. To see why, note that the system must be able to order multiple streams of commands simultaneously (e.g., one stream per partition) since totally ordering all commands cannot scale. But with independent streams of ordered commands, how to handle commands that address multiple partitions? Second, SMR hides from the service designer much of the complexity involved in replication; all the service designer must provide is a sequential implementation of each service command. If state is partitioned, then some commands may need data from multiple partitions. Should the service designer introduce additional logic in the implementation to handle such cases? Should the service be limited to commands that access a single partition? Throughout this thesis, we try to answer all these questions, while substantiating the following claim:

It is possible to devise an approach that allows fault-tolerant systems to scale, while providing strong consistency, that is, ensuring linearizability.

1.2 Research contributions

To prove the claim made above for this doctoral thesis, we propose a technique called **Scalable State Machine Replication (S-SMR)**. It solves the scalability (or lack thereof) problem faced by traditional state machine replication through partitioning the service state: a different group of servers is assigned to each partition and each server executes only requests that concern its own partition. Although totally ordering messages (i.e., with atomic broadcast [25]) allows building strongly consistent systems, it does not scale. To allow partitions to operate independently, thus providing scalability, client requests are only partially ordered in S-SMR. This is done by means of atomic multicast [25]. We also observe that atomic multicast alone does not guarantee linearizability and we show how to address this issue.

One problem of atomic multicast is that it has inherently higher delay than atomic broadcast [63], likely increasing response time when used for request ordering. For this reason, we introduce **Optimistic Atomic Multicast (Opt-amcast)**, which is a class of atomic multicast protocols with lower latency than previous ones, given a couple of reasonable assumptions. Opt-amcast protocols deliver each message twice: there is a conservative delivery, which always guarantees consistent order among messages, and an optimistic delivery, which happens within one single communication step and is likely to match the conservative delivery order, although ordering mistakes are possible. We propose an Opt-amcast protocol called **Ridge**, which combines high throughput with low latency, by using a throughput-optimal dissemination technique.

Finally, we use the fast optimistic delivery of Opt-amcast to reduce response time for commands. We do this by extending S-SMR with optimistic execution. The resulting model is called **Fast Scalable State Machine Replication (Fast-SSMR)**. It allows the execution of commands to be precomputed based on their optimistic delivery. Fast-SSMR uses a dual state machine approach: there is a conservative state machine and an optimistic state machine at each replica, and they execute independently most of the time. This design does not require services to be able to rollback execution: the conservative state is always consistent, although likely behind the optimistic one, which may be incorrect; if the optimistic state is incorrect, a repair is done by copying the conservative state onto the optimistic one. The complexity of the technique lies in determining at which point of the stream of optimistic commands the optimistic state machine should resume its execution after a repair is completed.

1.3 Thesis outline

The remainder of this thesis is organized as follows. In Chapter 2, we define the system model considered in this work and present some definitions that are used throughout the thesis. In Chapter 3, we briefly review traditional replication techniques, including SMR. In Chapter 4, we introduce S-SMR, explaining how it combines scalability with strong consistency. In Chapter 5, we introduce the concept of optimistic atomic multicast and detail Ridge, an implementation of optimistic atomic multicast that combines high throughput with low latency. In Chapter 6, we show how Fast-SSMR extends S-SMR with optimistic execution, reducing the response time of commands. Chapter 7 concludes this thesis and points at possible research directions to continue this work.

Chapter 2

System Model and Definitions

If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.

John von Neumann

In this chapter, we detail the system model and recall the notions of reliable multicast, atomic multicast, consensus, and linearizability, our correctness criterion.

2.1 Processes and communication

We assume a distributed system composed of a number of processes, divided into an unbounded set of client processes $\mathcal{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes $\mathcal{S} = \{s_1, ..., s_n\}$. The set of all process is $\Pi = \mathcal{C} \cup \mathcal{S}$. Set $\mathcal{S}$ is divided into k disjoint groups, $\mathcal{S}_1, ..., \mathcal{S}_k$. Each process is either correct, if it never fails, or faulty, otherwise. In any case, processes do not experience arbitrary behavior (i.e., no Byzantine failures).

It is impossible to solve consensus in an asynchronous system [28], so we consider a system that is *partially synchronous* [26]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *Global Stabilization Time (GST)* [26], and it is unknown to the processes. Before GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed; after GST, such bounds exist but are unknown to the processes.

Processes communicate by message passing, using one-to-one or one-to-many communication. One-to-one communication uses primitives $send(p, m)$

and *receive*(m), where m is a message and p is the destination process. If sender and receiver are correct, then every message sent is eventually received. One-to-many communication is done with reliable multicast and atomic multicast.

2.1.1 Reliable multicast

We define reliable multicast here with the primitives *reliable-multicast*(γ, m) and *reliable-deliver*(m), where m is a message and γ is the set of groups m is addressed to (for brevity, we write "process in γ" meaning "process in some group g, where $g \in \gamma$"). Reliable multicast guarantees the following properties:

- If a correct process reliable-multicasts m, then every correct process in γ reliable-delivers m *(validity)*.
- If a correct process reliable-delivers m, then every correct process in γ reliable-delivers m *(agreement)*.
- For any message m, every process p in γ reliable-delivers m at most once, and only if some process has reliable-multicast m to γ previously *(uniform integrity)*.

2.1.2 Atomic multicast

Similar to reliable multicast, atomic multicast also allows messages to be sent to a set γ of groups. It is defined by the primitives *atomic-multicast*(γ, m) and *atomic-deliver*(m). Let relation $\prec$ be defined such that $m \prec m'$ iff there is a destination that delivers m before m'. Atomic multicast guarantees the following:

- If a correct process atomic-multicasts m, then every correct process in γ atomic-delivers m *(validity)*.
- If a process atomic-delivers m, then every correct process in γ atomic-delivers m *(uniform agreement)*.
- For any message m, every process p in γ atomic-delivers m at most once, and only if some process has atomic-multicast m to γ previously *(uniform integrity)*.
- The order relation $\prec$ is acyclic. Also, for any two messages m and m' and any two processes p and q such that $\exists \{g, h\} \subseteq \gamma \cap \gamma' : \{p, q\} \subseteq g \cup h$, if p atomic-delivers m and q atomic-delivers m', then either p atomic-delivers m' before m or q atomic-delivers m before m' *(atomic order)*.

Atomic broadcast is a special case of atomic multicast in which there is a single process group. Atomic multicast provides uniform agreement, whereas our choice of reliable multicast ensures only non-uniform agreement. Uniform agreement states that messages delivered by faulty destination processes are also delivered by all correct destination processes. Non-uniform agreement disregards messages delivered only by faulty processes: for non-uniform agreement to hold, it suffices that every message delivered by a correct destination is delivered by all correct destinations. Although there are implementations of reliable multicast that provide uniformity, those implementations require two communication steps for messages to be delivered, while messages can be delivered within a single communication step if uniformity is not enforced [62].

2.2 Consensus

An important part of this work relies on consensus to ensure that processes agree upon which messages are delivered and in which order they are delivered. Thus, we assume that consensus can be solved. Moreover, we distinguish multiple instances of consensus executed with unique natural numbers. Consensus is defined by the primitives *propose*(k, v) and *decide*(k, v), where k is a natural number and v a value. Consensus satisfies the following properties in each instance k:

– If a process decides v, then v was previously proposed by some process *(uniform integrity)*.

– No two processes decide different values *(uniform agreement)*.

– If one (or more) correct process proposes a value then eventually some value is decided by all correct processes *(termination)*.

2.3 Linearizability

A service may be defined based on a set of *state variables* $\mathcal{V} = \{v_1, ..., v_m\}$ that encode the service's state and a set of *commands* (e.g., client requests) that change the state variables and produce an output. Every command C may (i) read state variables, (ii) perform some computation, (iii) modify state variables, and (iv) produce a response for the command. An *execution* consists of a sequence of application states alternating with command invocations and responses to such commands.

Linearizability is a consistency criterion defined based on the real-time precedence of commands in the execution and a sequential specification. If the response of command C_1 occurs in execution $\mathcal{E}$ before the invocation of command C_2, then C_1 precedes C_2 in $\mathcal{E}$ in real-time, which we denote as "$C_1 <_{RT} C_2$". The *sequential specification* of a service consists of a set of commands and a set of *legal sequences of commands*, which define the behaviour of the service when it is accessed sequentially. In a legal sequence of commands, every response to the invocation of a command immediately follows its invocation, with no other invocation or response in between them. For example, a sequence of operations for a read-write variable v is legal if every read command returns the value of the most recent write command that precedes the read, if there is one, or the initial value otherwise. Finally, an execution $\mathcal{E}$ is *linearizable* if there is a permutation π of the commands executed in $\mathcal{E}$ that (i) respects the service's sequential specification and (ii) if C_1 precedes C_2 in real-time in $\mathcal{E}$, then C_1 appears before C_2 in π [8].

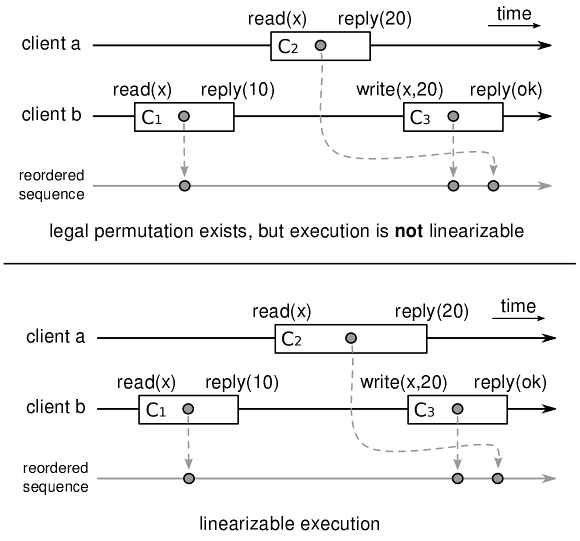

Figure 2.1. Linearizable vs. non-linearizable executions.

In Figure 2.1 (top), we show an example of an execution whose commands can be reordered forming a legal sequence (i.e., it respects the sequential specification of variable x). However, there is no permutation that is legal and respects the real-time precedence of commands at the same time: we can see that $C_1 <_{RT} C_2 <_{RT} C_3$, but a permutation that respected such real-time precedence would not be legal. In Figure 2.1 (bottom), however, we cannot determine any real-time precedence between C_2 and C_3, because C_3 was issued by client b before client a received the reply for C_2. For this reason, the execution at the bottom of the figure is linearizable.

Chapter 3

Replication

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

Replication in computer systems refers to managing a set of computers with redundant data and/or services in a way to ensure that they are consistent with each other, given a set of consistency criteria. It can be done to increase availability, throughput, or both. By availability, we mean that the system continues to work even in the presence of failures, as long as the number of failures is below a given tolerable limit. By throughput, we mean the amount of work done by the system per time unit, e.g., number of client requests handled per second.

There are many replication techniques [22], two of which have become widely well-known: *passive replication* (primary-backup) and *active replication* (state machine replication). The primary-backup replication approach [20] consists of having a primary replica that is responsible for replying to client requests, while having backup replicas that are able to take over the service if the primary one crashes. State machine replication [46, 47, 64, 71], on the other hand, consists of presenting the same set of client requests, in the same order, to all replicas; the replicas will then independently execute all requests and reach the same state after each request, and any of them may reply to the client. State machine replication thus has the following restriction: transitions in the state machine must be deterministic, based only on the previous state and on the request being handled, and all replicas must start at the same initial state. Since concurrency tends to be a source of non-determinism, state machine replicas are usually single-threaded.

Naturally, there are other replication schemes, such as chain replication [72, 73] and multi-primary replication, which is commonly used for transactional databases that implement deferred-update replication [22, 23, 65, 67].

In the next few sections, we briefly describe the replication models mentioned here. At the end of the chapter, we compare them based on their scalability and consistency level provided.

3.1 Primary-backup

In primary-backup replication (PBR) [20], only one of the replicas is the *primary* one; all the others are *backup* replicas. Clients send their requests to the primary replica, which executes the request and forwards the results to the backups. To ensure that any state change caused by the execution of the request was safely persisted, the primary sends a reply only after being sure that a number of backups have received the new state (e.g., by waiting for acknowledgements from backup replicas, as in Figure 3.1).

An obvious consequence of this model is that read-only requests do not require the primary to communicate with its backups, since there is no state change; instead, the primary replies immediately to any request that does not change the application state. Besides, it is possible to have non-deterministic execution of the requests: any non-determinism is resolved by the primary, which simply forwards the result to the backups. On the other hand, it is not possible for the clients to send any requests directly to a backup, not even read-only requests, unless linearizability is not required. As all requests (to ensure linearizability) have to be executed by the primary, adding replicas to the system does not increase its maximum throughput, although it makes the system more fault-tolerant.

If the primary fails, one of the backups takes over the role of primary and proceeds with the execution. No inconsistency can be seen by the clients, since no reply was sent before all backups had a copy of the state that preceded that reply. If a replica is removed from the system (e.g., by crashing), the system must be reconfigured so that the primary does not wait for its acknowledgement for future request executions any more.

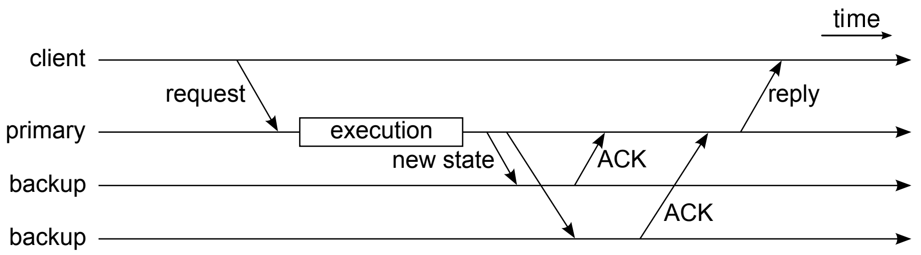

Figure 3.1. Primary-backup.

3.2 State machine replication

State-machine replication is a fundamental approach to implementing a fault-tolerant service by replicating servers and coordinating the execution of client commands against server replicas [46, 64]. The service is defined by a state machine, which consists of a set of *state variables* $\mathcal{V} = \{v_1, ..., v_m\}$ and a set of *commands* that may read and modify state variables, and produce a response for the command. SMR implements linearizability by regulating how client commands are propagated to and executed by the replicas: (i) every non-faulty replica must receive every command; and (ii) replicas must agree on the order of received and executed commands. As long as every command is implemented by a deterministic program, each replica will produce the same output upon executing the same sequence of commands. State machine replication can be implemented with atomic broadcast: if all commands are atomically broadcast to all servers, then all correct servers deliver and execute the same sequence of commands, i.e., they execute the commands in the same order. Moreover, if all servers have each a full copy of the application state, they all start at the same initial state, and command execution is deterministic, then every server reaches the same state after executing a given command and, thus, no two servers give different responses to the same command (Figure 3.2).

A great advantage of this technique is that replica failures do not interfere with the execution of commands by non-faulty replicas. As long as at least one replica is up and running, the clients should notice no difference in the response time to requests. On the other hand, requiring deterministic execution precludes possibly faster ways of executing commands. For instance, multi-threaded execution is (usually) non-deterministic. Because of that, SMR usually requires the execution of commands to be single-threaded.

There are some ways to optimize state machine replication. Read-only commands do not change the state of the replicas and can thus be executed by only

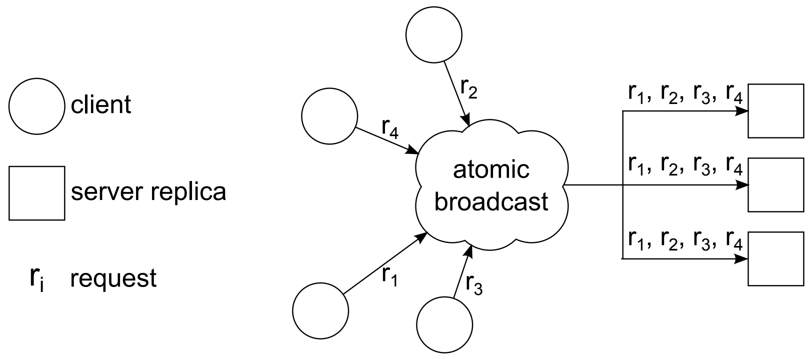

Figure 3.2. State machine replication.

one of them. Also, a single reply for each request is usually enough, so not every replica has to send a reply to the client that issued a request. These optimizations allow the system implementing such technique to increase its maximum throughput a bit more, but only up to a certain point, since all requests still need to be atomically ordered.

3.3 Chain replication

In chain replication, server replicas form a chain in order to serialize the transmission of client requests to all replicas. The replica at the head of the chain is responsible for receiving each request, executing it and forwarding it to the next replica in the chain, which effectively defines an order for all requests. Each of the remaining replicas must execute the requests in the order in which they were received from the previous replica. By doing this, this replication scheme would fit into the definition of state machine replication, where each replica implements a state machine, with the same restriction: request execution must be deterministic.

Alternatively, the head replica forwards only the result of the request execution and the other replicas simply apply the same state change. In this case, it would allow non-deterministic execution: whatever result was reached by the head replica (deterministically or not) would be applied to the state of all other replicas, ensuring consistency. This is not compatible with SMR any more, and this variant of chain replication would be an instance of the primary-backup replication scheme.

In any case, the replica responsible for sending the replies is the last one, or tail replica (as in Figure 3.3). This ensures that no reply is sent for a request before all replicas have applied the state change resulting from the request execution. An immediate optimization is to send requests that do not change the application state (e.g., read requests) to the tail replica only, which would reply immediately. This optimization does not hurt linearizability, since (i) this kind of requests would not change the state of the other replicas and (ii) all requests have to pass through the tail replica, so it would be impossible for requests to be handled in a non-linearizable way.

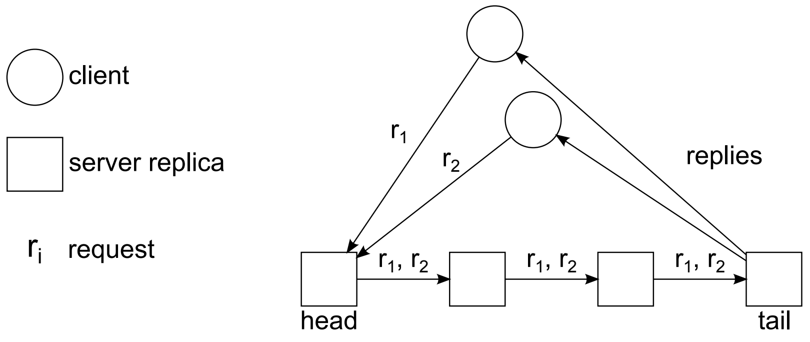

Figure 3.3. Chain replication.

The chain replication scheme proposed in [72] does not tolerate Byzantine failures. One of the reasons for that is that any of the replicas in the chain may forward incorrect information to its successors. For instance, the replica may forward incorrect requests or correct requests in an incorrect order (if the chain implements SMR), or incorrect states (if the chain implements PBR). Byzantine chain replication, proposed in [73], aims at addressing this issue. It relies on cryptographic signatures to create a *shuttle* message that is passed along and extended by each replica on its way. The shuttle contains a certification by the replica of: (i) the request (ii) the order assigned to that request and (iii) the result for the request execution by that replica. This is made to ensure that all replicas agree on what is the request currently being executed at each time and what is its result. If a mismatch is detected by any replica, the chain halts and an oracle is requested to fix the problem. In case of no replica reporting an error, at the end of the chain, the tail replica adds its own part of the shuttle and sends it back to the client, along with the reply to the request. Finally, the client may check whether the received reply agrees with the result found by all replicas.

3.4 Multi-primary replication

Both primary-backup and state machine replication schemes provide greater availability as replicas are added to the system. However, throughput hardly increases. In the case of the primary-backup scheme, all requests have to pass through the one primary replica in the system, which eventually becomes a bottleneck as its load is increased. State machine replication, on the other hand, requires all requests to be executed by all replicas (at least all non-read-only requests, even if optimizations are made), which also hits a performance limit at some point.

In multi-primary (or *deferred update* [22]) replication, this issue is addressed by allowing multiple replicas to act as primary at the same time. To do this, a replica receives the request, executes it locally and then communicates with all other replicas in the system to check if there are conflicting requests being executed at the same time. If not, then the request execution is confirmed and the state of the replicas is changed accordingly; otherwise, it is cancelled and the replica state remains unchanged [57, 59, 65, 66, 67]. In either case, the client is notified about the result (Figure 3.4).

To execute read-only requests (such as r_1 in Figure 3.4), replicas do not need to communicate with one another, since such requests do not invalidate the execution of any other request being executed at the same time. However, this allows non-linearizable executions to take place, so linearizability is not ensured by a deferred update replication scheme: a client may retrieve an old value for a data item after another client has already changed that value. Instead, the consistency property ensured by deferred update replication is *sequential consistency*; a system is sequentially consistent if there is a way to reorder the client requests in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the ordering of commands as defined by each client [8].

3.5 Combining linearizability and scalability

In this chapter, we have shown different ways of making services fault-tolerant by means of replication. Different techniques provide different properties; in particular, different levels of consistency. Some of them ensure linearizability, while others allow throughput to scale, but none of them provides both at the

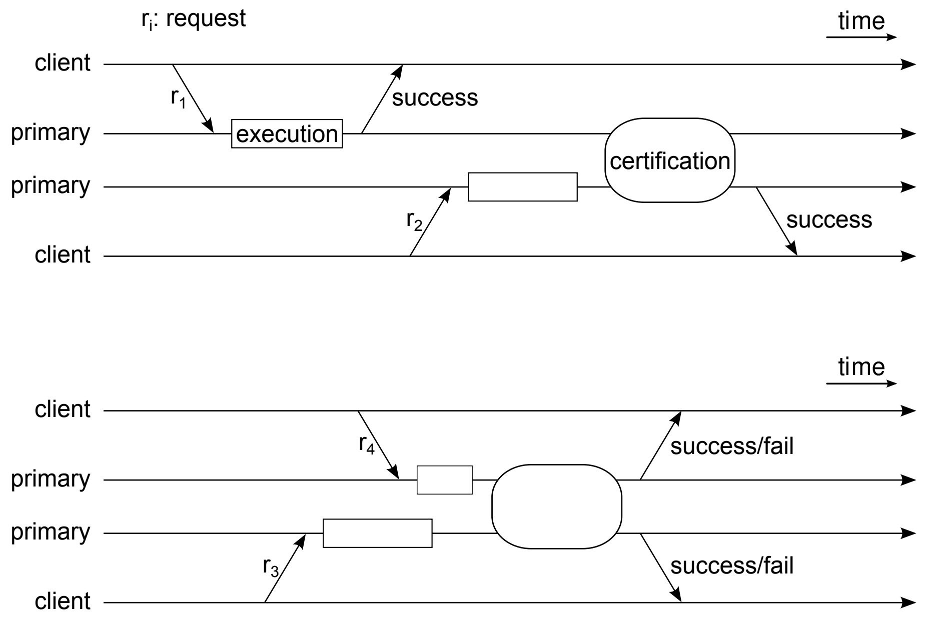

Figure 3.4. Multi-primary (or deferred update) replication.

same time.[1] Table 3.1 summarizes the kinds of replication techniques we have listed. We added a row representing the contribution of this thesis, that is, a scalable state machine replication technique (S-SMR), which combines linearizability with scalability.

Replication scheme	*Linearizable*	*Scalable*
Primary-backup	Yes	No
Multi-primary	No	Yes
State machine replication	Yes	No
Chain-replication	Yes	No
Scalable State Machine Replication	*Yes*	*Yes*

Table 3.1. Replication techniques and their properties.

In order to scale throughput, S-SMR employs state partitioning, so commands can be sent only to some servers, that is, the servers responsible for replicating the part of the state accessed by the command. Traditionally, SMR relies on totally ordering commands, but in the case of S-SMR, this may become a bottleneck. To

[1]One notable exception is Google's Spanner [24]. However, Spanner assumes clock synchrony and works better with special clock hardware, e.g., atomic clocks.

solve this problem, S-SMR does not totally order messages, but uses atomic multicast instead, which avoids enforcing a precedence relation between messages that do not share destinations. This allows the ordering layer (i.e., the atomic multicast protocol) to be itself also scalable.

Atomic multicast is scalable and is fundamental to S-SMR. However, it is inherently slower than atomic broadcast [63]. For this reason, we decided to create optimistic atomic multicast, a class of protocols that reduces message ordering time by allowing probabilistic ordering. We propose Ridge, our optimistic atomic multicast protocol, which improves previous techniques and uses a throughput-optimal message dissemination technique that does not penalize latency as much as other throughput-optimal techniques, while providing optimistic deliveries.

Chapter 4

Scalable State Machine Replication

When we had no computers, we had no programming problem either. When we had a few computers, we had a mild programming problem. Confronted with machines a million times as powerful, we are faced with a gigantic programming problem.

Edsger Dijkstra

This chapter presents Scalable State Machine Replication (S-SMR), an approach that achieves scalable throughput and strong consistency (i.e., linearizability) without constraining service commands or adding additional complexity to their implementation. S-SMR partitions the service state and relies on an atomic multicast primitive to consistently order commands within and across partitions. We show here that simply ordering commands consistently across partitions is not enough to ensure strong consistency in state machine replication when the state is partitioned. To solve this problem, S-SMR implements *execution atomicity*, a property that prevents command interleaves that violate strong consistency. To assess the performance of S-SMR, we developed Eyrie, a Java library that allows developers to implement partitioned-state services transparently, abstracting partitioning details, and Volery, a service that implements Zookeeper's API [36]. All communication between partitions is handled internally by Eyrie, including remote object transfers. In the experiments we conducted with Volery, throughput scaled with the number of partitions, in some cases linearly. In some deployments, Volery reached over 250 thousand commands per second, largely outperforming Zookeeper, which served 45 thousand commands per second under the same workload.

The following contributions are presented in this chapter: (1) It introduces S-SMR and explains its algorithm in detail. (2) It presents Eyrie, a library to simplify the design of services based on S-SMR. (3) It describes Volery to demonstrate how Eyrie can be used to implement a service that provides Zookeeper's API. (4) It presents a detailed experimental evaluation of Volery and compares its performance to Zookeeper.

The remainder of this chapter is organized as follows. Section 4.1 gives an overview of S-SMR. Section 4.2 explains the algorithm in detail. Section 4.3 proposes some performance optimizations. Section 4.4 argues about the correctness of the algorithm. Section 4.5 details the implementation of Eyrie and Volery. Section 4.6 reports on the performance of the Volery service. Section 4.7 surveys related work and Section 4.8 concludes the chapter.

4.1 General idea

S-SMR divides the application state $\mathcal{V}$ (i.e., state variables) into k partitions $\mathcal{P}_1, ..., \mathcal{P}_k$. Each variable v in $\mathcal{V}$ is assigned to at least one partition, that is, $\cup_{i=1}^{k} \mathcal{P}_i = \mathcal{V}$, and we define *part(v)* as the set of partitions that contain v. Each partition $\mathcal{P}_i$ is replicated by servers in group $\mathcal{S}_i$. For brevity, we say that server s belongs to $\mathcal{P}_i$ with the meaning that $s \in \mathcal{S}_i$, and say that client c multicasts command C to partition $\mathcal{P}_i$ meaning that c multicasts C to group $\mathcal{S}_i$. Finally, if server s is in (the server group that replicates) partition $\mathcal{P}$, we say that $\mathcal{P}$ is the *local partition* with respect to s, while every other partition is a *remote partition* to s.

To issue a command C to be executed, a client multicasts C to all partitions that hold a variable read or updated by C, denoted by *part(C)*. Consequently, the client must be able to determine the partitions that may be accessed by C. Note that this assumption does not imply that the client must know all variables accessed by C, nor even the exact set of partitions. If the client cannot determine a priori which partitions will be accessed by C, it must define a superset of these partitions, in the worst case assuming all partitions. For performance, however, clients must strive to provide a close approximation to the command's actually accessed partitions. We assume the existence of an oracle that tells the client which partitions should receive each command.

Upon delivering command C, if server s does not contain all variables read by C, s must communicate with servers in remote partitions to execute C. Essentially, s must retrieve every variable v read in C from a server that stores v (i.e., a server in a partition in *part(v)*). Moreover, s must retrieve a value of v that is consistent with the order in which C is executed, as we explain next.

In more detail, let *op* be an operation in the execution of command C. We distinguish between three operation types: *read(v)*, an operation that reads the value of a state variable v, *write*(v, val), an operation that updates v with value *val*, and an operation that performs a deterministic computation.

Server s in partition $\mathcal{P}_i$ executes *op* as follows.

i) *op* is a *read*(v) operation.
If $\mathcal{P}_i \in part(v)$, then s retrieves the value of v and sends it to every partition $\mathcal{P}_j$ that delivers C and does not hold v. If $\mathcal{P}_i \notin part(v)$, then s waits for v to be received from a server in a partition in $part(v)$.

ii) *op* is a *write*(v, val) operation.
If $\mathcal{P}_i \in part(v)$, s updates the value of v with *val*; if $\mathcal{P}_i \notin part(v)$, s executes *op*, creating a local copy of v, which will be up-to-date at least until the end of C's execution (caching is explained in Section 4.3.1).

iii) *op* is a computation operation.
In this case, s executes *op*.

As we now show, atomically ordering commands and following the procedure above is still not enough to ensure linearizability. Consider the execution depicted in Figure 4.1 (a), where state variables x and y have initial value of 10. Command C_x reads the value of x, C_y reads the value of y, and C_{xy} sets x and y to value 20. Consequently, C_x is multicast to partition $\mathcal{P}_x$, C_y is multicast to $\mathcal{P}_y$, and C_{xy} is multicast to both $\mathcal{P}_x$ and $\mathcal{P}_y$. Servers in $\mathcal{P}_y$ deliver C_y and then C_{xy}, while servers in $\mathcal{P}_x$ deliver C_{xy} and then C_x, which is consistent with atomic order. Based on those deliveries, the reply to C_x is 20, the reply to C_y is 10, and the only possible legal permutation for the commands in this execution is: C_y, C_{xy}, then C_x. However, this violates the real-time precedence of the commands, since C_x precedes C_y in real-time (C_y is invoked after the reply for C_x is received).

Intuitively, the problem with the execution in Figure 4.1 (a) is that commands C_x and C_y execute “in between” the execution of C_{xy} at partitions $\mathcal{P}_x$ and $\mathcal{P}_y$. In S-SMR, we avoid such cases by ensuring that the execution of every command is atomic. Command C is *execution atomic* if, for each server s that executes C, there exists at least one server r in every other partition in $part(C)$ such that the execution of C at s finishes only after r starts executing C. More precisely, let $start(C, p)$ and $end(C, p)$ be, respectively, the time when server p starts executing command C and the time when p finishes C's execution. Execution atomicity ensures that, for every server s in partition $\mathcal{P}$ that executes C, there is a server r

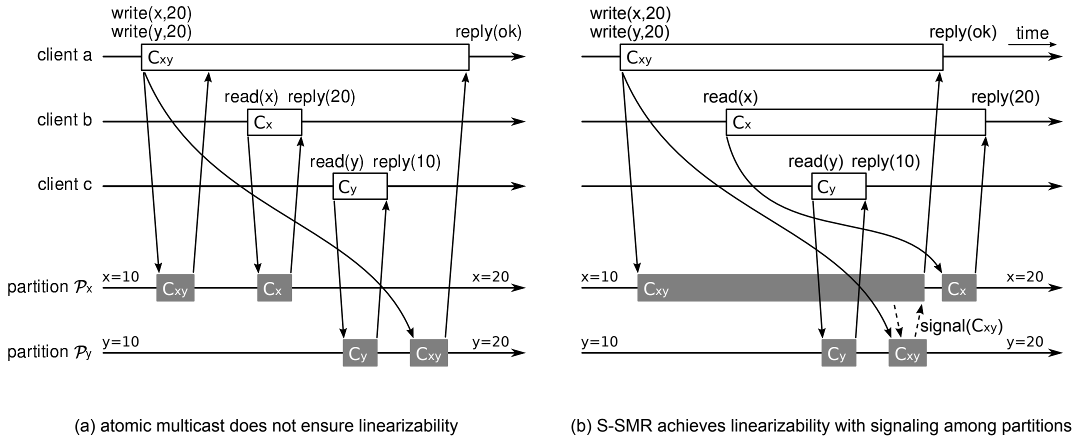
write(x,20)
write(y,20)
reply(ok)
client a
Cxy
client b
read(x) reply(20)
Cx
client c
read(y) reply(10)
Cy
partition $\mathcal{P}_x$
x=10
x=20
partition $\mathcal{P}_y$
y=10
y=20
(a) atomic multicast does not ensure linearizability
time
signal(Cxy)
(b) S-SMR achieves linearizability with signaling among partitions

Figure 4.1. Atomic multicast and S-SMR. (To simplify the figure, we show a single replica per partition.)

in every $\mathcal{P}' \in part(C)$ such that $start(C,r) < end(C,s)$. Intuitively, this condition guarantees that the execution of C at s and r overlap in time.

Replicas can ensure execution atomicity by coordinating the execution of commands. After starting the execution of command C, servers in each partition send a $signal(C)$ message to servers in the other partitions in $part(C)$. Before finishing the execution of C and sending a reply to the client that issued C, each server must receive a $signal(C)$ message from at least one server in every other partition that executes C. Because of this scheme, each partition is required to have at least $f+1$ correct servers for execution, where f is the maximum number of tolerated failures per partition; if all servers in a partition fail, service progress is not guaranteed. We separate execution from ordering [75]; the number or required correct ordering processes depends on the ordering protocol.

Figure 4.1 (b) shows an execution of S-SMR. In the example, servers in $\mathcal{P}_x$ wait for a signal from $\mathcal{P}_y$, therefore ensuring that the servers of both partitions are synchronized during the execution of C_{xy}. Note that the outcome of each command execution is the same as in case (a), but the execution of C_x, C_y and C_{xy}, as seen by clients, now overlap in time with one another. Hence, there is no real-time precedence among them and linearizability is not violated.

4.2 Detailed algorithm

The basic operation of S-SMR is shown in Algorithm 1. To submit a command C, the client queries an oracle to get set $C.dests$ (line 5), which is a superset of

Algorithm 1 Scalable State Machine Replication (S-SMR)

```
 1: Initialization:
 2:    $\forall C \in \mathcal{K} : rcvd\_signals(C) \leftarrow \emptyset$
 3:    $\forall C \in \mathcal{K} : rcvd\_variables(C) \leftarrow \emptyset$
 4: Command $C$ is submitted by a client as follows:
 5:    $C.dests \leftarrow oracle(C)$          {$oracle(C)$ returns a superset of $part(C)$}
 6:    atomic-multicast($C.dests, C$)
 7:    wait for reply from one server
 8: Command $C$ is executed by a server in partition $\mathcal{P}$ as follows:
 9:    upon atomic-deliver($C$)
10:       $others \leftarrow C.dests \setminus \{\mathcal{P}\}$
11:       reliable-multicast($others, signal(C)$)
12:       for each operation $op$ in $C$ do
13:          if $op$ is $read(v)$ then
14:             if $v \in \mathcal{P}$ then
15:                reliable-multicast($others, \langle v, C.id \rangle$)
16:             else
17:                wait until $v \in rcvd\_variables(C)$
18:                update $v$ with the value in $rcvd\_variables(C)$
19:          execute $op$
20:       wait until $rcvd\_signals(C) = others$
21:       send reply to client
22:    upon reliable-deliver($signal(C)$) from partition $\mathcal{P}'$
23:       $rcvd\_signals(C) \leftarrow rcvd\_signals(C) \cup \{\mathcal{P}'\}$
24:    upon reliable-deliver($\langle v, C.id \rangle$)
25:       $rcvd\_variables(C) \leftarrow rcvd\_variables(C) \cup \{v\}$
```

Algorithm variables:

$\mathcal{K}$: the set of all possible commands

$C.id$: unique identifier of command C

$oracle(C)$: function that returns a superset of $part(C)$

$C.dests$: set of partitions to which C is multicast

$others$: set of partitions waiting for signals and variables from $\mathcal{P}$; also, $\mathcal{P}$ waits for signals from all such partitions

$signal(C)$: a synchronization message that allows S-SMR to ensure C to be execution atomic

$rcvd\_signals(C)$: a set containing all partitions that already signaled $\mathcal{P}$ regarding the execution of C

$rcvd\_variables(C)$: a set containing all variables that must be received from other partitions in order to execute C

part(C) used by the client as destination set for C (line 6).

Upon delivering C (line 9), server s in partition $\mathcal{P}$ starts executing C and multicasts *signal*(C) to *others*, which is the set containing all other partitions involved in C (lines 10 and 11). It might happen that s receives signals concerning C from other partitions even before s started executing C. For this reason, s must buffer signals and check if there are signals buffered already when starting the execution of C. For simplicity, Algorithm 1 initializes such buffers as $\emptyset$ for all possible commands. In practice, buffers for C are created when the first message concerning C is delivered.

After multicasting signals, server s proceeds to the execution of C, which is a sequence of operations that might read or write variables in $\mathcal{V}$, or perform a deterministic computation. The main concern is with operations that read variables, as the values of those variables may have to be sent to (or received from) remote partitions. All other operations can be executed locally at s. If the operation reads variable v and v belongs to $\mathcal{P}$, s's partition, then s multicasts the value of v to the other partitions that delivered C (line 15). The command identifier $C.id$ is sent along with v to make sure that the other partitions will use the appropriate value of v during C's execution. If v belongs to some other partition $\mathcal{P}'$, s waits until an up-to-date value of v has been delivered (line 17). Every other operation is executed with no interaction with other partitions (line 19).

After executing all operations of C, s waits until a signal from every other partition has been received (line 20) and, only then, sends the reply back to the client (line 21). This ensures that C will be execution atomic.

4.3 Performance optimizations

Algorithm 1 can be optimized in many ways. We briefly describe here some of them and present caching in more detail.

- Even though all replicas in all partitions in *part*(C) execute C, a reply from a single replica (from a single partition) suffices for the client to finish the command.

- Servers can keep a cache of variables that belong to remote partitions; when multi-partition commands are executed and remote variables are received, this cache is verified and possibly updated.

- It is not necessary to exchange each variable more than once per command since any change during the execution of the command will be deterministic and thus any changes to the variable can be applied to the cached value.

– Server s does not need to wait for the execution of command C to reach a $read(v)$ operation to only then multicast v to the other partitions in $part(C)$. If s knows that v will be read by C, s can send v's value to the other partitions as soon as s starts executing C.

4.3.1 Caching

When executing multi-partition commands, servers of different partitions may have to exchange state variables. In order to reduce the amount of data exchanged and the time required to execute this kind of commands, each server can keep a cache of variables that belong to remote partitions. When a server executes a command that reads a variable from a remote partition, the server stores the read value in a cache and uses the cached value in future read operations. If a command updates a remote variable, the server updates (or creates) the cached value. The value of a variable stored in a server's cache stays up-to-date until (i) the server discards that variable's entry from the cache due to memory limitations, or (ii) some command that was not multicast to that server changes the value of the variable. In S-SMR, servers of any partition $\mathcal{P}$ know what servers of other partitions may be keeping cached values of variables from $\mathcal{P}$. They even know if such cached values are up-to-date (provided that they were not discarded to free memory). Based on this information, servers can proactively send messages to other servers to let them know if the values they cache are still valid.

For instance, say there are two partitions $\mathcal{P}$ and $\mathcal{P}_x$, and there is a variable x in $\mathcal{P}_x$. Every command that accesses x is multicast to $\mathcal{P}_x$, and each command contains the list of partitions it was multicast to. Servers in $\mathcal{P}_x$ can use this information to keep track of what other servers received commands that access x. When a command C that reads or updates x is multicast to both $\mathcal{P}$ and $\mathcal{P}_x$, servers in $\mathcal{P}$ update their cache with the value of x, which will stay valid until some other command changes x's value. Servers in $\mathcal{P}_x$ know that servers in $\mathcal{P}$ also delivered C, since $\mathcal{P} \in C.dests$, and that they are now aware of the newest value of x. Say a command C_r, which reads x, is also multicast to $\mathcal{P}_x$ and $\mathcal{P}$, and it is the first command that accesses x after C. Servers of $\mathcal{P}_x$ know that servers in $\mathcal{P}$ executed command C, and the value of x has not changed ever since. Thus, they know that the cached value of x in those servers is still valid. So, as soon as C_r is delivered, the servers of $\mathcal{P}_x$ send a message to servers in $\mathcal{P}$ notifying that the value they hold of x is up-to-date. Naturally, some servers of $\mathcal{P}$ may have discarded the cache entry for x, so they will have to send a request to $\mathcal{P}_x$ servers for x's newest value. If x was changed by a different command C_w that was executed after C, but before C_r (i.e., $C \prec C_w \prec C_r$), the servers of $\mathcal{P}_x$ will know that the value cached in the servers of $\mathcal{P}$ is stale and send the newest value. How servers use cached variables distinguishes conservative from speculative caching, which we describe next.

Conservative caching: Once server s has a cached value of x, it waits for a cache-validation message from a server in $\mathcal{P}_x$ before executing a $read(x)$ operation. The cache validation message contains of tuples $\langle var, val \rangle$, where *var* is a state variable that belongs to $\mathcal{P}_x$ and whose cache in $\mathcal{P}$ needs to be validated. If servers in $\mathcal{P}_x$ determined that the cache is stale, *val* contains the new value of *var*; otherwise, *val* contains $\perp$, telling s that its cached value is up to date. If s had a valid cache of x (therefore receiving $\perp$ as its value), but discarded x's cached copy, s sends a request for x to $\mathcal{P}_x$.

Speculative caching: It is possible to reduce execution time by speculatively assuming that cached values are up-to-date. Speculative caching requires servers to be able to rollback the execution of commands, in case the speculative assumption fails to hold. Some applications (e.g., databases) allow rolling back the execution of a command, as long as no reply has been sent to the client for the command yet.

The difference between speculative caching and conservative caching is that in the former servers that keep cached values do not wait for a cache-validation message before reading a cached entry; instead, a $read(x)$ operation returns the cached value immediately. If after reading some variable x from the cache, during the execution of command C, server s receives a message from a server in $\mathcal{P}_x$ that invalidates the cached value, s rolls back the execution to some point before the $read(x)$ operation and resumes the command execution, now with the up-to-date value of x. Either with conservative or speculate caching, a server can only send the reply for a command after every variable read from the cache has been validated.

4.4 Correctness

In this section, we argue that, if every command in execution $\mathcal{E}$ of S-SMR is delivered by atomic multicast and is execution atomic, then $\mathcal{E}$ is linearizable. Execution atomicity is ensured by the signaling mechanism, described in Section 4.1.

Proposition 1. *S-SMR ensures linearizability.*

Proof. In S-SMR, servers execute commands in the order determined by atomic multicast. As defined in Chapter 2, we denote the order given by atomic multicast by relation $\prec$. Given any two messages m_a and m_b, "$m_a \prec m_b$" means that both messages are delivered by the same processes and m_a is delivered before m_b, or there are messages $m_1, \ldots, m_n$ such that $m_a \prec m_1$, $m_1 \prec m_2$, and so on, and $m_n \prec m_b$, which can be written as $m_a \prec m_1 \prec \cdots \prec m_n \prec m_b$.

We prove here that S-SMR does not violate linearizability: if a command x precedes a command y in real time, i.e., the reply to x is received before y is issued by a client, then it is impossible in S-SMR for y to precede x in the execution. Suppose, by means of contradiction, that x finishes (i.e., its reply is received) before y starts (i.e., a client issues y), but $y \prec x$ in the execution. There are two possibilities to be considered: (i) x and y are delivered by the same process p, or (ii) no process delivers both x and y.

In case (i), at least one process p delivers both x and y. As x finishes before y starts, then p delivers x, then y. From the properties of atomic multicast, and since each partition is mapped to a multicast group, no process delivers y, then x. Therefore, we reach a contradiction in this case.

In case (ii), if there were no other commands in $\mathcal{E}$, then the execution of x and y could be done in any order, which would contradict the supposition that $y \prec x$. Therefore, there are commands $z_1, ..., z_n$ with atomic order $y \prec z_1 \prec \cdots \prec z_n \prec x$, where some process p_0 (of partition $\mathcal{P}_0$) delivers y, then z_1; some process $p_1 \in \mathcal{P}_1$ delivers z_1, then z_2, and so on: process $p_i \in \mathcal{P}_i$ delivers z_i, then z_{i+1}, where $1 \le i < n$. Finally, process $p_n \in \mathcal{P}_n$ delivers z_n, then x.

Let $z_0 = y$ and let *atomic*(i) be the following predicate: "For every process $p_i \in \mathcal{P}_i$, p_i finishes executing z_i only after some $p_0 \in \mathcal{P}_0$ started executing z_0." We now claim that *atomic*(i) is true for every i, where $0 \le i \le n$. We prove our claim by induction:

- *Basis ($i = 0$)*: *atomic*(0) is obviously true, as p_0 can only finish executing z_0 after starting to execute z_0.

- *Induction step*: If *atomic*(i) is true, then *atomic*$(i+1)$ is also true.

  Proof: Command z_{i+1} is multicast to both $\mathcal{P}_i$ and $\mathcal{P}_{i+1}$. Since z_{i+1} is execution atomic, before any $p_{i+1} \in \mathcal{P}_{i+1}$ finishes executing z_{i+1}, some $p_i \in \mathcal{P}_i$ starts executing z_{i+1}. Since $z_i \prec z_{i+1}$, every $p_i \in \mathcal{P}_i$ starts executing z_{i+1} only after finishing the execution of z_i. As *atomic*(i) is true, this will only happen after some $p_0 \in \mathcal{P}_0$ started executing z_0.

As $z_n \prec x$, for every $p_n \in \mathcal{P}_n$, p_n executes command x only after the execution of z_n at p_n finishes. From the above claim, this happens only after some $p_0 \in \mathcal{P}_0$ starts executing y. This means that y (z_0) was issued by a client before any client received a response for x, which contradicts the assumption that x precedes y in real-time. □

4.5 Implementation

In this section, we describe Eyrie, a library that implements S-SMR, and Volery, a service that provides Zookeeper's API. Both Eyrie and Volery were implemented in Java.

4.5.1 Eyrie

One of the main goals of Eyrie[1] is to make the implementation of services based on S-SMR as easy as possible. To use Eyrie, the developer (i.e., service designer) must extend two classes, `PRObject` and `StateMachine`. Class `PartitioningOracle` has a default implementation, but the developer is encouraged to override its methods.

The `PRObject` class

Eyrie supports partial replication (i.e., some objects may be replicated in some partitions, not all). Therefore, when executing a command, a replica might not have local access to some of the objects involved in the execution of the command. The developer informs to Eyrie which object classes are replicated by extending the `PRObject` class. Such class represents any kind of data that is part of the service state. Each instance of `PRObject` may be stored locally or remotely, but the application code is agnostic to that. All calls to methods of such objects are intercepted by Eyrie, transparently to the developer.

Eyrie uses AspectJ[2] to intercept method calls for all subclasses of `PRObject`. Internally, the aspect related to such method invocations communicates with the `StateMachine` instance in order to (i) determine if the object is stored locally or remotely and (ii) ensure that the object is up-to-date when each command is executed. It is worth noting that methods of the subclasses of `PRObject` must be intercepted by the aspect, be they field accesses or not: say there is an object

[1]https://bitbucket.org/kdubezerra/eyrie
[2]http://eclipse.org/aspectj

`user`, of class `User`, which extends `PRObject`. When calling a method such as `user.getField()` to access a field of `user`, that method should return the most up-to-date value of the field. This may require the caller to fetch such recent value from a remote server. However, this is done only to subclasses of `PRObject`: methods of class `PRObject` itself are not intercepted.

Each replica has a local copy of all `PRObject` objects. When a remote object is received, replicas in the local partition $\mathcal{P}_L$ must update their local copy of the object with the received (up-to-date) value. For this purpose, the developer must provide implementations for the methods `getDiff(Partition p)` and `updateFromDiff(Object diff)`. The former is used by the remote partition $\mathcal{P}_R$, which owns the object, to calculate a delta between the old value currently held by $\mathcal{P}_L$ and the newest value, held by $\mathcal{P}_R$. Such implementations may be as simple as returning the whole object. However, it also allows the developer to implement caching mechanisms. Since `getDiff` takes a partition as parameter, the developer may keep track of what was the last value received by $\mathcal{P}_L$, and then return a (possibly small) diff, instead of the whole object. The diff is then applied to the object with the method `updateFromDiff`, which is also provided by the application developer.

To avoid unnecessary communication, the developer may optionally mark some methods of their `PRObject` subclasses as local, by annotating them with `@LocalMethodCall`. Calls to such methods are not intercepted by the library, sparing communication when the developer sees fit. Although the command that contains a call to such a method still has to be delivered and executed by all partitions that hold objects accessed by the command, that particular local method does not require an up-to-date object. For example, say a command C accesses objects O_1 and O_2, respectively in partitions $\mathcal{P}_1$ and $\mathcal{P}_2$. C completely overwrites objects O_1 and O_2, by calling `O1.clear()` and `O2.clear()`. Although C has to be delivered by both partitions to ensure linearizability, a write method that completely overwrites an object, regardless of its previous state, does not need an up-to-date version of the object before executing. Because of this, method `clear()` can be safely annotated as local, avoiding unnecessary communication between $\mathcal{P}_1$ and $\mathcal{P}_2$.

The `StateMachine` class

Linearizability is ensured by the `StateMachine` class: it executes commands one by one, in the order defined by atomic multicast, and implements the exchange of signals as described in Section 4.1. This class is abstract and must be extended by the application server class. To execute commands, the developer must provide an implementation for the method `executeCommand(Command c)`. The code for such a method is agnostic to the existence of partitions. In other words, it can be exactly the same as the code used to execute commands with classical state machine replication (i.e., full replication). Eyrie is responsible for handling all communication between partitions transparently. To start the server, method `runStateMachine()` is called.

The `PartitioningOracle` class

Clients multicast each command directly to the partitions affected by the command, i.e., those that contain objects accessed by the command. Although Eyrie encapsulates most details regarding partitioning, the developer must provide an oracle that tells, for each command, which partitions are affected by the command. The set of partitions returned by the oracle needs to contain all partitions involved, but does not need to be minimal. In fact, the default implementation of the oracle simply returns all partitions for every command, which although correct, is not efficient. For best performance, the partition set returned by the oracle should be as small as possible, which requires the developer to extend `PartitioningOracle` and override its methods.

Method `getDestinations(Command c)` is used by the clients to ask the oracle which partitions should receive each command. It returns a list of `Partition` objects and can be overridden with an implementation provided by the application developer. Such an implementation must return a list containing all partitions involved in the execution of `c`. Another important method of the `PartitioningOracle` class is the `getLocalObjects(Command c)` method, which allows servers to send objects as soon as the execution of the command that accesses these objects starts. This method returns a list of local objects (i.e., objects in the server's partition) that will be accessed by `c`. It may not be possible to determine in advance which objects are accessed by every command, but the list of accessed objects does not need to be complete. Any kind of early knowledge about which objects need to be updated in other partitions helps decrease execution time. The default implementation of this method returns an empty list, which means that objects are exchanged among partitions as their methods are invoked during execution.

Other classes

In the following, we briefly describe a few other classes provided by Eyrie.

The `Partition` class has two relevant methods, `getId()` and `getPartitionList()`, which return respectively the partition's unique identifier and the list of all partitions in the system. When extending `PartitioningOracle`, the developer can use this information to map commands to partitions.

To issue a command (i.e., a request), a client must create a `Command` object containing the command's parameters. The `Command` object is then multicast to the partitions determined by the partitioning oracle at the client. The `Command` class offers methods `addItems(Objects... objs)`, `getNext()`, `hasNext()` and so on. How the server will process such parameters is application-dependent and determined by how the method `executeCommand` of the `StateMachine` class is implemented.

Eyrie uses atomic multicast to disseminate commands from clients and handle communication between partitions. This is done by configuring a `LocalReplica` object, which is created by parsing a configuration file provided by the developer, in both clients and servers. Eyrie is built on top of a multicast adaptor library,[3] being able to easily switch between different implementations of atomic multicast.

4.5.2 Volery

We implemented the Volery service on top of Eyrie, providing an API similar to that of Zookeeper [36]. Zookeeper implements a hierarchical key-value store, where each value is stored in a *znode*, and each znode can have other znodes as children. The abstraction implemented by Zookeeper resembles a file system, where each path is a unique string (i.e., a key) that identifies a znode in the hierarchy. We implemented the following Volery client API:

– `create(String path, byte[] data)`: creates a znode with the given `path`, holding `data` as content, if there was no znode with that path previously and there is a znode with the parent path.

– `delete(String path)`: deletes the znode that has the given `path`, if there is one and it has no children.

– `exists(String path)`: returns True if there exists a znode with the given `path`, or False, otherwise.

– `getChildren(String path)`: returns the list of znodes that have `path` as their parent.

[3]https://bitbucket.org/kdubezerra/libmcad

– `getData(String path)`: returns the data held by the znode identified by `path`.

– `setData(String path, byte[] data)`: sets the contents of the znode identified by `path` to `data`.

Zookeeper ensures a mix of linearizability (for write commands) and session consistency (for read commands). Every reply to a read command (e.g., `getData`) issued by a client is consistent with all write commands (e.g., `create` or `setData`) issued previously by the same client. With this consistency model and with workloads mainly composed of read-only commands, Zookeeper is able to scale throughput with the number of replicas. Volery ensures linearizability for every command. In order to scale, Volery makes use of partitioning, done with Eyrie.

Distributing Volery's znodes among partitions was done based on each znode's path: Volery's partitioning oracle uses a hash function $h(path)$ that returns the id of the partition responsible for holding the znode at *path*. Each command `getData`, `setData`, `exists` and `getChildren` is multicast to a single partition, thus being called a *local command*. Commands `create` and `delete` are multicast to all partitions and are called *global commands*; they are multicast to all partitions to guarantee that every (correct) replica has a full copy of the znodes hierarchy, even though only the partition that owns each given znode is guaranteed to have its contents up-to-date.

4.6 Performance evaluation

In this section, we assess the performance of S-SMR, in terms of throughput scalability and latency. For this purpose, we conducted experiments with Volery. We evaluated S-SMR's throughput scalability by deploying the system with different numbers of partitions, message sizes, both with on-disk storage and with in-memory storage. Our goal was also to find the limits of S-SMR, showing the performance of the technique as the percentage of global commands in the workload increases. We compare Volery results with Zookeeper and ZKsmr, which is an implementation of Zookeeper with traditional SMR.

4.6.1 Environment setup and configuration parameters

We ran all our experiments on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at

2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve Switch 2910al-48G gigabit network switch, and the Dell nodes were connected to an HP ProCurve 2900-48G gigabit network switch. Those switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 6.3 with kernel 2.6.32 and had the Oracle Java SE Runtime Environment 7. Before each experiment, we synchronize the clocks of the nodes using NTP. This is done to obtain accurate values in the measurements of the latency breakdown involving events in different servers.

In all our experiments with Volery and Zookeeper, clients submit commands asynchronously, that is, each client can keep submitting commands even if replies to previous commands have not been received yet, up to a certain number of outstanding commands. Trying to issue new commands when this limit is reached makes the client block until some reply is received. Replies are processed by callback handlers registered by clients when submitting commands asynchronously. We allowed every client to have up to 25 outstanding commands at any time. By submitting commands asynchronously, the load on the service can be increased without instantiating new client processes. Local commands consisted of calls to `setData`, while global commands were invocations to `create` and `delete`. "Message size" and "command size", in the next sections, refer to the size of the byte array passed to such commands.

We compared Volery with the original Zookeeper and with ZKsmr, which is an implementation of the Zookeeper API using traditional state machine replication. For the Zookeeper experiments, we used an ensemble of 3 servers. For the other approaches, we used Multi-Ring Paxos for atomic multicast, having 3 acceptors per ring: ZKsmr had 3 replicas that used one Paxos ring to handle all communication, while Volery had 3 replicas per partition, with one Paxos ring per partition, plus one extra ring for commands that accessed multiple partitions. Each server replica was in a different cluster node. Also, since Zookeeper runs the service and the broadcast protocol (i.e., Zab [38]) in the same machines, each ZKsmr/Volery replica was colocated with a Paxos acceptor in the same node of the cluster. We had workloads with three different message sizes: 100, 1000 and 10000 bytes. Volery was run with 1, 2, 4 and 8 partitions. We conducted all experiments using disk for storage, then using memory (for Zookeeper, we used a ramdisk at each server). For on-disk experiments, we configured Multi-Ring Paxos with Δ of 40 ms [55], batching timeout of 50 ms and batch size threshold of 250 kilobytes; for in-memory experiments, these parameters were 5 ms, 50 ms and 30 kilobytes, respectively.

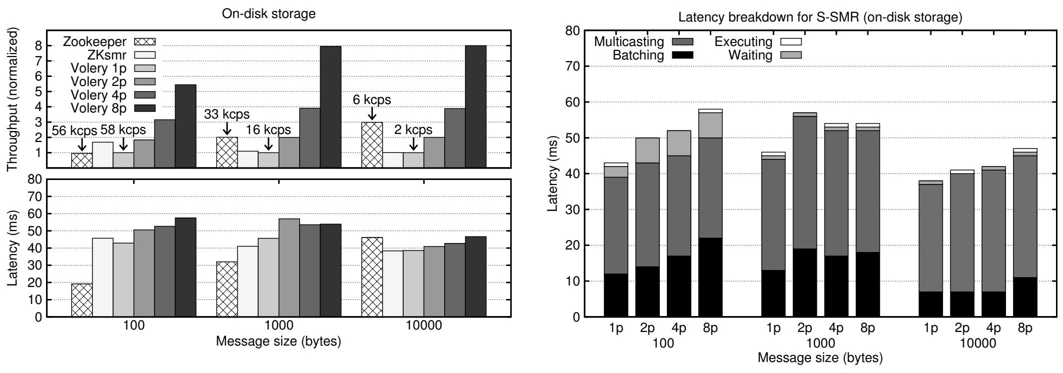

Figure 4.2. Results for Zookeeper, ZKsmr and Volery with 1, 2, 4 and 8 partitions, using on-disk storage. Throughput was normalized by that of Volery with a single partition (absolute values in thousands of commands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput.

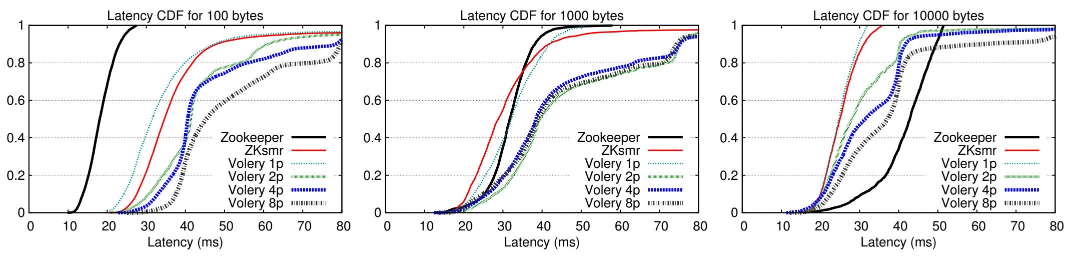

Figure 4.3. Cumulative distribution function (CDF) of latency for different command sizes (on-disk storage).

4.6.2 Experiments using on-disk storage

In Figure 4.2, we show results for local commands only. Each Paxos acceptor wrote its vote synchronously to disk before accepting each proposal. Zookeeper also persisted data to disk. In Figure 4.2 (top left), we can see the maximum throughput for each replication scheme and message size, normalized by the throughput of Volery with a single partition. Each bar represents the maximum throughput found in that configuration after many runs of the same experiment with different loads. In all cases, the maximum throughput of Volery scaled with the number of partitions and, for message sizes of 1000 and 10000 bytes, it scaled linearly (ideal case). For small messages (100 bytes), Zookeeper has similar performance to Volery with a single partition. As messages increase in size, Zookeeper's throughput improves with respect to Volery: with 1000-byte messages, Zookeeper's throughput is similar to Volery's throughput with two partitions. For large messages (10000 bytes), Zookeeper is outperformed by Volery with four partitions. Comparing S-SMR with traditional SMR, we can see that for small messages (100 bytes), ZKsmr performed better than Volery with one partition. This is due to the additional complexity added by Eyrie in order to ensure linearizability when data is partitioned. Such difference in throughput is less significant with bigger commands (1000 and 10000 bytes).

We can also see in Figure 4.2 (bottom left) the latency values for the different implementations tested. Latency values correspond to 75% of the maximum throughput. Zookeeper has the lowest latency for 100- and 1000-byte command sizes. For 10000-byte commands, Volery had similar or lower latency than Zookeeper. Such lower latency of Volery with 10000-byte commands is due to a shorter time spent with batching: as message sizes increase, the size threshold of the batch (250 kilobytes for on-disk storage) is reached faster, resulting in lower latency.

Figure 4.2 (right) shows the latency breakdown of commands executed by Volery. For the experiments with Volery, we used Multi-Ring Paxos to implement atomic multicast: to multicast a message, a process sends the message to a ring coordinator, which then proposes the message for consensus. *Batching* is the time elapsed from the moment the client sends command C to a ring coordinator until the moment when a batch of messages containing C is proposed. *Multicasting* is the time from when the command is proposed until when the batch that contains C is delivered by a server replica. *Waiting* represents the time between the delivery of C and the moment when C starts executing. *Executing* measures the delay between the start of the execution of command C until the client receives C's response. We can see that more than half of the latency time is due to multicasting, which includes saving Multi-Ring Paxos instances synchronously to disk. There is also a significant amount of time spent with batching, done to reduce the number of disk operations and allow higher throughput: each Paxos proposal is saved to disk synchronously, so increasing the number of commands per proposal (i.e., per batch) reduces the number of times the disk is accessed. This improves throughput, but increases latency.

In Figure 4.3, we show the cumulative distribution functions (CDFs) of latency for all experiments where disk was used for storage. The results show that the latency distributions for ZKsmr and Volery with a single partition are similar, while latency had more variation for 2, 4 and 8 partitions. An important difference between deployments with a single and with multiple partitions is related to how Multi-Ring Paxos is used. In ZKsmr and in Volery with a single partition, there is only one Paxos ring, which orders all commands from all clients and delivers them to all replicas. When there are multiple partitions, each

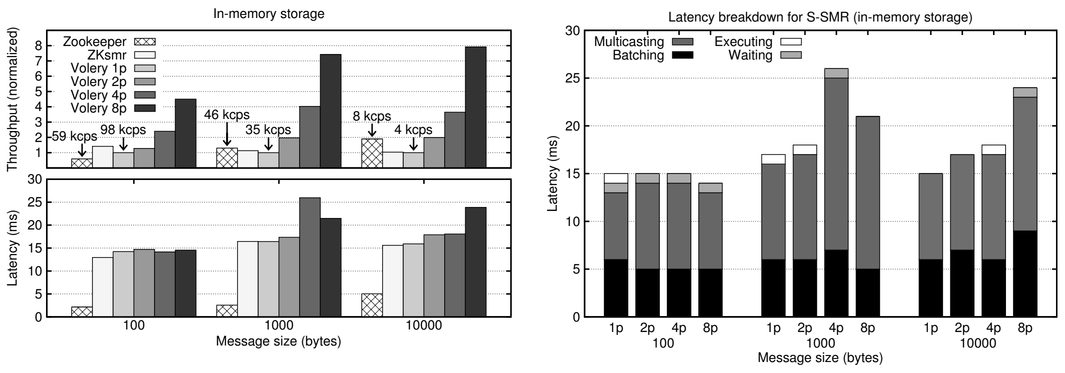

Figure 4.4. Results for Zookeeper, ZKsmr and Volery with 1, 2, 4 and 8 partitions, using in-memory storage. Throughput was normalized by that of Volery with a single partition (absolute values in thousands of commands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput.

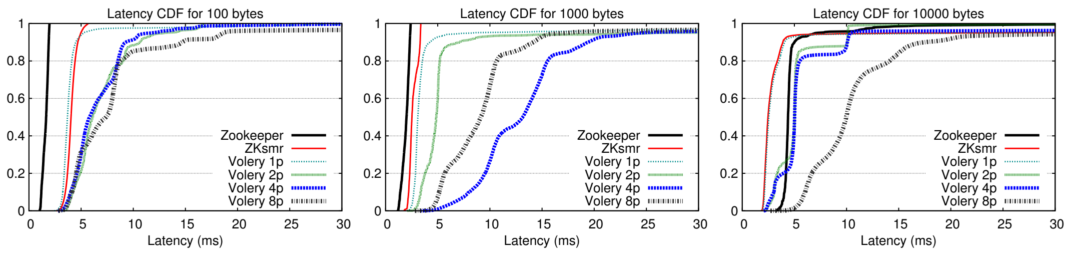

Figure 4.5. Cumulative distribution function (CDF) of latency for different command sizes (in-memory storage).

replica delivers messages from two rings: one ring that orders messages related to the replica's partition only, and another ring that orders messages addressed to more than one partition—each replica deterministically merges deliveries from multiple rings. As the time necessary to perform such deterministic merge is influenced by the level of synchrony of the rings, latency is expected to fluctuate more when merging is involved.

4.6.3 Experiments using in-memory storage

In Figure 4.4, we show the results for local commands when storing data in memory only. Volery's throughput scales with the number of partitions (Figure 4.4 (top left)), specially for large messages, in which case throughput grows linearly with the number of partitions (i.e., ideal case). We can also see that latency values for Volery and ZKsmr are less than half of what they are for on-

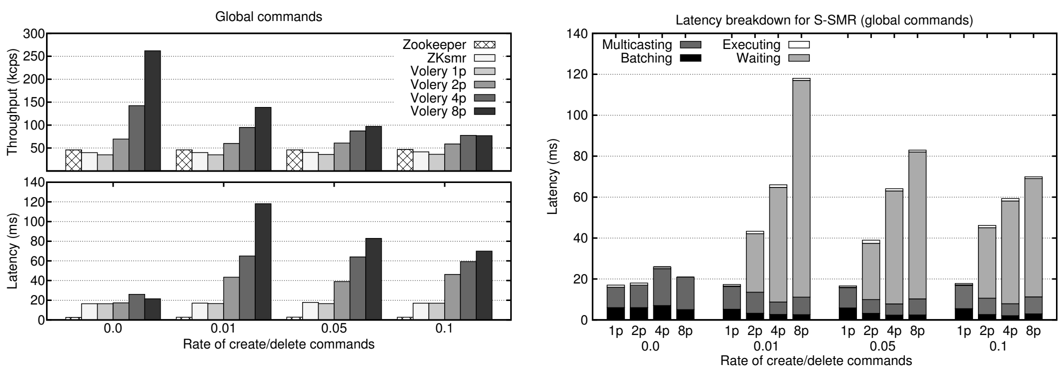

Figure 4.6. Throughput and latency versus rate of create/delete commands (in-memory storage, 1000-bytes commands). Throughput is shown in units of a thousand commands per second (kcps). Latencies shown correspond to 75% of the maximum throughput.

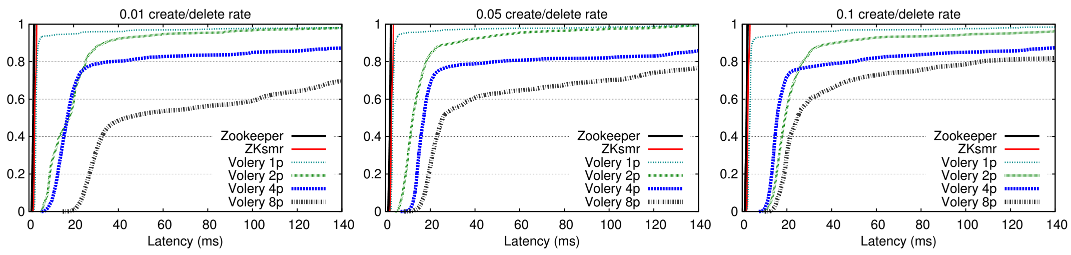

Figure 4.7. Cumulative distribution function (CDF) of latency for different rates of create/delete commands (in-memory storage, 1000-bytes commands).

disk storage (Figure 4.4 (bottom left)), while Zookeeper's latency decreased by an order of magnitude.

Figure 4.4 (right) shows the latency breakdown. Even though no data is saved to disk, multicasting is still responsible for most of the latency, followed by batching. In these experiments, although there are no disk writes, batching is still used because it reduces the number of Paxos proposals and the number of messages sent through the network, which allows higher throughput. Figure 4.5 shows the latency CDFs for the in-memory experiments, where we can see that Volery with multiple partitions (i.e., deployments where Multi-Ring Paxos uses multiple rings) tends to have more variation in latency.

4.6.4 Experiments with global commands

In this section, we analyze how Volery performs when the workload includes commands that are multicast to all partitions (global commands), which is the least favorable scenario for S-SMR. Having commands multicast to all partitions effectively limits throughput scalability: if all commands go to all partitions, adding more partitions will not increase throughput.

We ran experiments with different rates of global commands (i.e., create and delete operations): 0%, 1%, 5% and 10% of all commands. We chose such rates for two reasons: (i) it is obvious that high rates of global commands will prevent the system from scaling, plus (ii) it is common for large scale services to have a high rate of read requests (which are local commands in Volery). An example of such a service is Facebook's TAO [19], which handles requests to a social graph; it allows, for instance, pages to be generated based on the user's connections in the social network. In Facebook's TAO, 99.8% of all requests are read-only [19].

We can see in Figure 4.6 (top left) that Volery scales throughput with the number of partitions for all configurations but the exceptional case of 10% of global commands when augmenting the number of partitions from 4 to 8. Moreover, Volery with two partitions outperforms Zookeeper in all experiments. The major drawback of Volery under global commands is that to ensure linearizability, partitions must exchange signals: as `create` and `delete` commands are multicast to all partitions, no server can send a reply to a client before receiving a signal from all other partitions when executing such a command. This explains the significant increase in latency shown in Figure 4.6 (bottom left), as global commands are added to the workload: as the number of partitions increases, so does the average latency. As we can see in Figure 4.6 (right), this extra latency comes from the servers waiting for signals from other partitions.

Figure 4.7 shows the latency CDFs for the workloads with global commands. For experiments with more than one partition, the rate of messages with high latency is much higher than the rate of global commands. This happens due to a "convoy effect": local commands may be delivered after global commands, having to wait for the latter to finish.

4.7 Related work

State machine replication is a well-known approach to replication and has been extensively studied (e.g., [40, 43, 46, 61, 64]). State machine replication requires replicas to execute commands deterministically, which implies sequential execution. Even though increasing the performance of state machine replication is non-trivial, different techniques have been proposed for achieving scalable systems, such as optimizing the propagation and ordering of commands (i.e., the underlying atomic broadcast algorithm). In [39], the authors propose to have clients send their requests to multiple computer clusters, where each such cluster executes the ordering protocol only for the requests it received, and then forwards this partial order to every server replica. The server replicas, then, must deterministically merge all different partial orders received from the ordering clusters. In [16], Paxos [48] is used to order commands, but it is implemented in a way such that the task of ordering messages is evenly distributed among replicas, as opposed to having a leader process that performs more work than the others and may eventually become a bottleneck.

State machine replication seems at first to prevent multi-threaded execution since it may lead to non-determinism. However, some works have proposed multi-threaded implementations of state machine replication, circumventing the non-determinism caused by concurrency in some way. In [61], for instance, the authors propose organizing each replica in multiple modules that perform different tasks concurrently, such as receiving messages, batching, and dispatching commands to be executed. The execution of commands is still sequential, but the replica performs all other tasks in parallel. We also implemented this kind of parallelism in Eyrie.

Some works have proposed to parallelize the execution of commands in SMR. In [43], application semantics is used to determine which commands can be executed concurrently without reducing determinism (e.g., read-only commands can be executed in any order relative to one another). Upon delivery, commands are directed to a parallelizer thread that uses application-supplied rules to schedule multi-threaded execution. Another way of dealing with non-determinism is proposed in [40], where commands are speculatively executed concurrently. After a batch of commands is executed, replicas verify whether they reached a consistent state; if not, commands are rolled back and re-executed sequentially. Both [43] and [40] assume a Byzantine failure model and in both cases, a single thread is responsible for receiving and scheduling commands to be executed. In the Byzantine failure model, command execution typically includes signature handling, which can result in expensive commands. Under benign failures, command execution is less expensive and the thread responsible for command reception and scheduling may become a performance bottleneck.

Many database replication schemes also aim at improving the system throughput, although commonly they do not ensure strong consistency as we define it here (i.e., as linearizability). Many works (e.g., [23, 42, 67, 69]) are based on the deferred-update replication scheme, in which replicas commit read-only transactions immediately, not necessarily synchronizing with each other. This provides a significant improvement in performance, but allows non-linearizable executions to take place. The consistency criteria usually ensured by database systems are serializability [10] or snapshot isolation [51]. Those criteria can be considered weaker than linearizability, in the sense that they do not take into account real-time precedence of different commands among different clients. For some applications, this kind of consistency is good enough, allowing the system to scale better, but services that require linearizability cannot be implemented with such techniques.

Other works have tried to make linearizable systems scalable [24, 31, 54]. In [31], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In [54], a partitioned variant of SMR is proposed, supporting single-partition updates and multi-partition read operations. It relies on total order: all commands have to be ordered by a single sequencer (e.g., a Paxos group of acceptors), so that linearizability is ensured. The replication scheme proposed in [54] does not allow multi-partition update commands. Spanner [24] uses a separate Paxos group per partition. To ensure strong consistency across partitions, it assumes that clocks are synchronized within a certain bound that may change over time. The authors say that Spanner works well with GPS and atomic clocks.

Scalable State Machine Replication employs state partitioning and ensures linearizability for any possible execution, while allowing throughput to scale as partitions are added, even in the presence of multi-partition commands and unsynchronized clocks.

4.8 Conclusion

This chapter introduced S-SMR, a scalable variant of the well-known state machine replication technique. S-SMR differs from previous related works in that it allows throughput to scale with the number of partitions without weakening consistency. To evaluate S-SMR, we developed the Eyrie library and implemented Volery, a Zookeeper clone, with Eyrie. Our experiments demonstrate that in deployments with 8 partitions and under certain workloads, throughput experienced an 8-time improvement, resulting in ideal scalability. Moreover, Volery's throughput proved to be significantly higher than Zookeeper's.

One problem that we faced was that the time spent ordering client requests (i.e., the multicast component of latency) was a significant part of the response time in our experiments. To address this issue, we designed an atomic multicast protocol that optimizes for latency without sacrificing throughput. The resulting protocol is presented in the next chapter.

Chapter 5

Optimistic Atomic Multicast

Optimism is an essential ingredient of innovation.
How else can the individual welcome change over
security, adventure over staying in safe places?
Robert Noyce

In [34], it has been shown that the maximum throughput in a point-to-point network may be achieved by disposing processes in a ring overlay. The prospects of achieving throughput optimality with a ring overlay have motivated the design of several protocols such as LCR [34], Ring Paxos [53], Multi-Ring Paxos [55, 9], and Spread [7] (some of which coupled with ip-multicast communication). All these protocols achieve very high throughput, but are subject to an inherent limitation of ring overlays: the latency to deliver messages is proportional to the number of processes in the system.

In this chapter, we set out to investigate whether more latency efficient message dissemination techniques can promise the same throughput optimality of a ring overlay. We show here that it is possible to reach optimal throughput without resorting to a ring topology or to ip-multicast, which is usually unavailable in wide-area networks. Based on this result, we introduce Ridge, a Paxos-based atomic multicast protocol where each message is initially forwarded to a single destination, the *distributor*, whose responsibility is to propagate the message to all other destinations. To use all bandwidth available in the system, processes alternate in the role of distributor. As a result, Ridge's maximum throughput matches the throughput of ring-based protocols, with a latency that does not significantly depend on the system size.

To further reduce latency, Ridge performs two deliveries for each message: an *optimistic delivery*, which is fast (one communication step) but has a chance of violating order, and a *conservative delivery*, which always guarantees consistent delivery order across all destinations, but takes longer than the optimistic one. To deliver messages optimistically, a process estimates a wait time based on expected message transmission delays and skews between node clocks. If the process' estimate holds, the optimistic order (i.e., the order in which messages are delivered optimistically) will match the conservative order (i.e., the order in which messages are delivered conservatively). We define here the *optimistic atomic multicast* class of protocols, which are the atomic multicast protocols that perform both conservative and optimistic deliveries, as described above.

This chapter presents the following contributions: (a) we introduce Ridge, a high-throughput, latency-efficient optimistic atomic multicast protocol; (b) we reason about Ridge's theoretical maximum performance; (c) we detail how optimistic delivery is implemented in Ridge; (d) we provide a detailed experimental evaluation of Ridge's performance and compare it to the performance of other ordering protocols.

The remainder of this chapter is structured as follows. Section 5.1 defines optimistic atomic multicast. Section 5.2 recalls the Paxos protocol, which is used by Ridge. Sections 5.3 and 5.4 describe Ridge and analyze its performance analytically. Section 5.5 assesses Ridge's performance experimentally. Section 5.6 surveys related work and Section 5.7 concludes the chapter.

5.1 Problem definition

In this section, we define the optimistic atomic multicast class of message ordering protocols. Later in this chapter, we show how Ridge implements such a protocol.

Optimistic atomic multicast performs two deliveries for each multicast message m: an *optimistic delivery* and a *conservative delivery*. It is defined by primitives opt-amcast(γ, m), opt-deliver(m) and cons-deliver(m).

The conservative delivery has the same properties as atomic multicast, i.e., validity, uniform agreement, integrity and atomic order (defined by relation $\prec$). Let relation $\prec_{opt}$ be such that $m \prec_{opt} m'$ iff a process opt-delivers m before m'. The optimistic delivery has the same properties as reliable multicast, i.e., validity, agreement, and integrity, in addition to the following:

- If the given optimistic assumptions[1] hold, the order relation $\prec_{opt}$ does not contradict the relation $\prec$ *(optimistic order)*.

Finally, optimistic atomic multicast ensures the following:

- If a correct process p delivers m optimistically, p also delivers m conservatively, and vice-versa *(equivalence)*.

The optimistic order is probabilistic: with high probability processes deliver messages optimistically and conservatively in the same order. If the i-th optimistic delivery made at a process p matches the i-th conservative delivery at p, we say that the order of the i-th message was correct; otherwise, we say that the optimistic delivery made a *mistake*.

5.2 Paxos

Paxos [48] is a fault-tolerant consensus algorithm and the basis for Ridge. We describe next how a value is decided in a single consensus instance. Paxos distinguishes three roles: *proposers*, *acceptors*, and *learners*. Each process can play one or more of these roles simultaneously. Proposers propose values, acceptors choose values, and learners learn what value was decided. Hereafter, A denotes the set of acceptors, L the set of learners, and Q a *majority quorum* of acceptors, that is, a subset of A of size $\lceil(|A|+1)/2\rceil$.

The execution of one consensus instance proceeds in a sequence of *rounds*, each identified by a number. For each round, one process (typically one of the proposers) acts as *coordinator* of the round. To propose a value, proposers send the value to the coordinator. The coordinator maintains two variables: (a) *c-rnd* is the highest-numbered round that the coordinator has started; and (b) *c-val* is the value that the coordinator has picked for round *c-rnd*. Acceptors maintain three variables: (a) *rnd* is the highest-numbered round in which the acceptor has participated; (b) *v-rnd* is the highest-numbered round in which the acceptor has cast a vote—it follows that $rnd \leq v\text{-}rnd$ always holds; and (c) *v-val* is the value voted by the acceptor in round *v-rnd*.

Algorithm 2 provides an overview of Paxos. The algorithm has two phases. To execute Phase 1, the coordinator picks a round number *c-rnd* greater than any value it has picked so far, and sends it to the acceptors (Task 1). Upon receiving

[1]The optimistic assumptions depend on the implementation of optimistic atomic multicast. We explain Ridge's optimistic assumptions later.

Algorithm 2 Paxos

```
 1: Initialization:
 2:    $c\text{-}rnd \leftarrow 0; rnd \leftarrow 0; v\text{-}rnd \leftarrow 0$
 3:    $c\text{-}val \leftarrow null; v\text{-}val \leftarrow null$
 4: Task 1 (coordinator)
 5: when receiving value $v$ from proposer
 6:    $c\text{-}rnd \leftarrow$ unique value higher than current $c\text{-}rnd$
 7:    for all $acc \in A$ do send ($acc$, ⟨PHASE 1A, $c\text{-}rnd$⟩)
 8: Task 2 (acceptor)
 9: when receiving ⟨PHASE 1A, $c\text{-}rnd$⟩ from coordinator
10:    if $c\text{-}rnd > rnd$ then
11:       $rnd \leftarrow c\text{-}rnd$
12:       send (coordinator, ⟨PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$⟩)
13: Task 3 (coordinator)
14: when receiving ⟨PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$⟩ from $Q$, such that $rnd = c\text{-}rnd$
15:    $h \leftarrow$ highest $v\text{-}rnd$ value received
16:    $V \leftarrow$ set of ⟨$v\text{-}rnd$, $v\text{-}val$⟩ received with $v\text{-}rnd = h$
17:    if $h = 0$ then $c\text{-}val \leftarrow v$
18:    else $c\text{-}val \leftarrow$ the only $v\text{-}val$ in $V$
19:    for all $acc \in A$ do send ($acc$, ⟨PHASE 2A, $c\text{-}rnd$, $c\text{-}val$⟩)
20: Task 4 (acceptor)
21: when receiving ⟨PHASE 2A, $c\text{-}rnd$, $c\text{-}val$⟩ from coordinator
22:    if $c\text{-}rnd \geq rnd$ then
23:       $v\text{-}rnd \leftarrow c\text{-}rnd$
24:       $v\text{-}val \leftarrow c\text{-}val$
25:       send (coordinator, ⟨PHASE 2B, $v\text{-}rnd$, $v\text{-}val$⟩)
26: Task 5 (coordinator)
27: when receiving ⟨PHASE 2B, $v\text{-}rnd$, $v\text{-}val$⟩ from $Q$
28:    if for all received messages: $v\text{-}rnd = c\text{-}rnd$ then
29:       for all $learner \in L$ do send ($learner$, ⟨DECISION, $v\text{-}val$⟩)
```

such a message (Task 2), an acceptor checks whether the round number received from the coordinator is greater than any round number it has received so far; if that is the case, the acceptor "promises" not to accept any future message with a round smaller than *c-rnd*. The acceptor then replies to the coordinator with the highest-numbered round in which it has cast a vote, if any, and the value it voted for. Notice that no value is proposed to the acceptors in Phase 1.

The coordinator starts Phase 2 after receiving a reply from a quorum (Task 3). Before proposing a value in Phase 2, the coordinator checks whether some acceptor has already cast a vote in a previous round. If not, the coordinator can use the value received from the proposer. If one or more acceptors have cast votes in previous rounds, the coordinator picks the value that was voted for in the highest-numbered round.

An acceptor will vote for a value *c-val* with corresponding round *c-rnd* in Phase 2 if the acceptor has not received any Phase 1 message for a higher-numbered round (Task 4). Voting for a value means setting the acceptor's variables *v-rnd* and *v-val* to the values sent by the coordinator. If the acceptor votes for the value received, it replies to the coordinator. When the coordinator receives replies from a quorum (Task 5), it knows that a value has been decided and notifies the learners.

Algorithm 2 can be optimized in a number of ways [48]. For instance, the coordinator can execute Phase 1 before values are received from proposers. This way, decisions can be faster: when a proposer sends a value to the coordinator, the coordinator can immediately start Phase 2, as Phase 1 has already been executed. In the next section, we describe Ridge in detail, including its optimization to Paxos Phase 2.

5.3 Ridge

Ridge is an optimistic atomic multicast protocol that can deliver messages to groups of processes. In this section, we describe the general idea of Ridge (Section 5.3.1), detail its operation (Section 5.3.2), explain how it tolerates failures (Section 5.3.3), and how it performs optimistic deliveries (Section 5.3.4). At the end of this section, we discuss the correctness of Ridge (Section 5.3.5).

5.3.1 Overview of the protocol

Ridge makes use of Paxos, executing Phase 1 similarly to Algorithm 2, while optimizing Phase 2 for high throughput and low latency. Ridge utilizes a collection of *ensembles*, where an ensemble is a set of processes capable of executing a sequence of Paxos instances. Each ensemble contains $2f+1$ acceptors, where f is the maximum number of failures tolerated by the ensemble. For each run of Paxos, Ridge defines a majority quorum Q with $f+1$ acceptors. To decide on message m in an ensemble, the ensemble's coordinator (which has previously run Phase 1 of Paxos for multiple consensus instances) starts Phase 2 by sending m to an acceptor, which forwards it to another acceptor, and so on, until $f+1$ acceptors have received the message. Unless something abnormal occurs (e.g., a failure or multiple coordinators proposing messages in the same Paxos instance), the last acceptor (i.e., the $f+1$-th acceptor to receive m) will know that a majority of acceptors have received m, configuring a quorum. Then, the last acceptor sends m to a learner. Such a learner, called the *distributing learner* (or distributor) for m, then sends m to all other learners directly, completing Phase 2 of Paxos. Learners take turns as the distributing learners for different messages. By doing this, Ridge achieves high throughput, assuming that each learner distributes the same amount of data. To approximate this assumption, a load balancing procedure is used: the last acceptor keeps track of how much data each learner has distributed so far and chooses the distributing learner for the next message accordingly.

Ridge allows learners to receive messages from different ensembles; hereafter, we denote messages decided in an ensemble a *message stream*. If a learner subscribes to multiple ensembles, a deterministic procedure is used to merge the corresponding message streams, as we explain next.

To merge message streams, Ridge employs Paxos instance ids to ensure a gap-free sequence of decisions from each ensemble. It also employs timestamps to determine the position of each message in the resulting merged sequence. For this to be possible, each ensemble must ensure that the timestamp order of decided messages follow the messages' decision order (for messages of the same ensemble). More formally, let m be a message with timestamp $m.ts$ decided in consensus instance k by ensemble e, and let m' be a message with timestamp $m'.ts$ decided in instance k' also by ensemble e. We need that, if $k < k'$, then $m.ts < m'.ts$. When a message m is created, an *initial timestamp* is assigned to it by its sender. When m is decided and received by a learner, it is only delivered when the learner knows that there will be no message m' with a lower timestamp than that of m to be received from any ensemble the learner subscribes to. To ensure that the timestamp order and the instance order agree for each ensemble, each learner follows a deterministic procedure: upon delivery, if messages m and m' are decided in the same ensemble, $k < k'$, but $m.ts > m'.ts$, then the timestamp of m' is adjusted to be higher than that of m, being its *final* *timestamp*. Once the learner has determined the final timestamp of a message, it proceeds with deterministic merge. Every timestamp is assumed to be unique across the system and the final message sequence follows the timestamp order of the merged message streams.

The way Ridge merges multiple message sequences raises a liveness concern: what if a learner never receives a message from one of the ensembles it subscribes to (i.e., because no process multicasts a message to this ensemble)? To ensure liveness, Ridge uses *null messages*: if no message is multicast to an ensemble for a predefined time Δ, the coordinator of the ensemble generates a null message (i.e., a message with no payload), with the sole purpose of preventing learners from blocking while waiting for ensembles with low or no traffic.

Finally, in Ridge processes can propagate a message to a single ensemble only, which is at odds with the definition of atomic multicast, where a message can be multicast to multiple groups. To implement the abstraction of groups, Ridge maps groups to ensembles, as we describe next. Suppose a system in which messages can be multicast to any combination of groups $g_1, ..., g_n$. This abstraction can be implemented in Ridge with $n+1$ ensembles, $e_1, ..., e_n, e_{all}$. For every group g_i that contains process p, p becomes a learner of ensembles e_i and e_{all}. If m is multicast to a single group g_i, it is propagated to e_i; if m is multicast to multiple groups, it is sent to e_{all}, with non-addressee processes simply discarding m—that is, the ensemble's learners that have not subscribed to any of the groups that m was multicast to will just disregard m.

5.3.2 Detailed algorithms

We detail here the algorithms executed during normal operation, i.e., when there are no failures or failure suspicions; we discuss abnormal cases in Section 5.3.3. Algorithm 3 shows how Ridge optimizes Paxos Phase 2 for both throughput and latency (unchanged code from original Paxos is grayed out). For Algorithm 4, which details Ridge's deterministic merge procedure, we assume that each ensemble uses consensus as a black box, with primitives *propose*$_e(k, v)$ and *decide*$_e(k, v)$ to respectively propose and decide a value v for consensus instance k of ensemble e. Finally, Algorithm 5 shows a simple way of mapping Paxos ensembles to multicast groups, allowing messages to be multicast to multiple process groups.

To solve consensus in Ridge, we assume that each ensemble eventually has a single and correct coordinator [48]. To execute Phase 1 (Task 1), the coordinator defines a quorum Q for the proposed round of Paxos containing a majority of the acceptors in A, arranged in a sequence $a_1; a_2; \ldots; a_{(f+1)}$. When starting Phase 2

Algorithm 3 Ridge: executing Paxos in one ensemble

```
1: Initialization:
2:    $c\text{-}rnd \leftarrow 0; rnd \leftarrow 0; v\text{-}rnd \leftarrow 0$
3:    $c\text{-}val \leftarrow null; v\text{-}val \leftarrow null$
4: Task 1 (coordinator)
5: when receiving value $v$ from proposer
6:    $c\text{-}rnd \leftarrow$ unique value higher than current $c\text{-}rnd$
7:    // define a quorum $Q$ as an acceptor sequence for round $c\text{-}rnd$
8:    $Q \leftarrow \{a_1, a_2, \ldots, a_{(f+1)}\}$ // majority of the $2f+1$ acceptors
9:    for all $acc \in A$ do send ($acc$, ⟨PHASE 1A, $c\text{-}rnd$⟩)
10: Task 2 (acceptor)
11: when receiving ⟨PHASE 1A, $c\text{-}rnd$⟩ from coordinator
12:    if $c\text{-}rnd > rnd$ then
13:       $rnd \leftarrow c\text{-}rnd$
14:       send (coordinator, ⟨PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$⟩)
15: Task 3 (coordinator)
16: when receiving ⟨PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$⟩ from $Q$, such that $rnd = c\text{-}rnd$
17:    $h \leftarrow$ highest $v\text{-}rnd$ value received
18:    $V \leftarrow$ set of ⟨$v\text{-}rnd$, $v\text{-}val$⟩ received with $v\text{-}rnd = h$
19:    if $h = 0$ then $c\text{-}val \leftarrow v$
20:    else $c\text{-}val \leftarrow$ the only $v\text{-}val$ in $V$
21:    send (acceptor $a_1$, ⟨PHASE 2, $c\text{-}rnd$, $c\text{-}val$, $Q$, 0⟩)
22: Task 4 (acceptor $a_i$)
23: when receiving ⟨PHASE 2, $c\text{-}rnd$, $c\text{-}val$, $Q$, $count$⟩
24:    if $c\text{-}rnd \geq rnd$ then
25:       $v\text{-}rnd \leftarrow c\text{-}rnd$
26:       $v\text{-}val \leftarrow c\text{-}val$
27:       if $count + 1 < |Q|$ then
28:          send ($a_{(i+1)}$, ⟨PHASE 2, $v\text{-}rnd$, $v\text{-}val$, $Q$, $count+1$⟩)
29:       else
30:          for all $l \in L$ do send ($l$, ⟨DECISION, $id(v\text{-}val)$⟩)
31:          $l_d \leftarrow$ a learner in $L \setminus Q$
32:          send ($l_d$, ⟨DISTRIBUTE, $v\text{-}val$⟩)
33: Task 5 (learner)
34: when receiving ⟨DISTRIBUTE, $v\text{-}val$⟩
35:    for all $l \in L \setminus Q$ do send ($l$, ⟨VALUE, $v\text{-}val$⟩)
```

of Paxos (Task 3), the coordinator sends message ⟨PHASE 2, *c-rnd*, *c-val*, Q, 0⟩ to acceptor a_1. This tuple means that it is a message concerning Phase 2 of Paxos, for round *c-rnd*, proposing value *c-val*, the acceptor sequence in Q is the quorum, and no votes have been cast by the acceptors yet.

Upon receiving a message ⟨PHASE 2, *c-rnd*, *c-val*, Q, *count*⟩ (Task 4), acceptor a_i knows that *count* votes have been cast so far to decide value *c-val* in consensus. If the number of votes, plus a_i's own vote, is still not enough to reach a quorum, a_i increments the vote count and forwards the message to the next acceptor in Q, $a_{(i+1)}$. When the $|Q|$ votes necessary to decide *c-val* have been cast, the acceptor that completed the quorum will choose a distributing learner and ask for it to distribute the value decided to the other learners (Task 5). We want to divide equally the amount of data distributed by each of the learners. For simplicity, Algorithm 3 assumes that all learners are chosen uniformly by the last acceptor to be the distributors and that all messages have the same size. Ridge's actual implementation uses a load balancer at each acceptor to help choose distributing learners.

It is possible that some of the learners are also acceptors in the majority quorum Q. An acceptor from Q should not be a distributing learner, as such an acceptor already receives and forwards proposed values (in Task 4). Adding the burden of distributing decided values would likely overload the acceptor and defeat the purpose of Ridge, which is to maximize throughput. For this reason, the distributing learner is chosen from $L \setminus Q$. For learners in Q, which in normal cases have already received the value decided, to be notified about decisions, two separate messages are sent: VALUE, which contains the decided value *v-val*, and DECISION, which contains only *id(v-val)*, a unique identifier of *v-val*. The last acceptor of Q sends DECISION to all learners, while the distributing learner sends VALUE only to learners that haven't received *v-val* yet. By doing this, and assuming that the size of *id(v-val)* is negligible, Ridge's throughput efficiency is maintained even when processes are both learners and acceptors.

Ridge implements atomic multicast with the help of an intermediate layer, called *ensemble-multicast*, which allows processes do deterministically merge messages decided in multiple ensembles. If we consider the set of learners of each ensemble as an *ensemble-group*, the properties of ensemble-multicast (i.e., *ensemble-delivery*) are the ones of atomic multicast, described in Chapter 2, except that each message can be ensemble-multicast to only one single ensemble-group.

Algorithm 4 shows how ensemble-multicast operates. To multicast a message m to an ensemble-group of learners of ensemble e, the proposer p first assigns a unique timestamp $m.ts$ to m, which is based on p's system clock. Then, p sends

m to e's coordinator (lines 2–4). Upon receiving m, the coordinator proceeds with proposing m in the next consensus instance k (lines 10 and 11). When a learner becomes aware that m is the next decided message in ensemble e (line 30), it checks if m has a timestamp higher than $last\_ts_e$ (the timestamp of the last message from e); if not, $m.ts$ is adjusted to $last\_ts_e + 1$, ensuring that the timestamp and instance id orders agree in every ensemble. After checking this, the learner sets $last\_ts_e$ to $m.ts$ and puts $\langle e, m\rangle$ in a queue to be delivered when possible (lines 31–34). This queue contains a gap-free sequence of decisions from each ensemble; k_e is incremented each time a decision is received from an ensemble e, ensuring that the learner will be notified about decisions from e in the correct order (line 35).

When the *gap_free* queue contains at least one decision from every ensemble the learner subscribes to, the learner removes the message with lowest timestamp from the *gap_free* queue and ensemble-delivers the message (lines 36–40). This can be done because the learner processes consensus decisions from each ensemble in order; since final timestamps are adjusted to agree with the consensus decision order, any new decisions will contain a message with higher timestamp.

Algorithm 4 also shows how Ridge ensures liveness. At the initialization, and whenever a proposal is made, a timer is set to expire in Δ time units (lines 8, 13 and 19). This makes sure that, if a learner has a message m enqueued for delivery, it will eventually be able to deliver it, because it will receive messages with increasing timestamps from every ensemble at least every Δ (line 36).

Finally, Algorithm 5 shows how Ridge implements atomic multicast on top of ensemble-multicast. The ensemble-multicast protocol does not allow a message to be multicast to multiple ensembles. To allow each message to be multicast to multiple groups, we can have an ensemble e_γ for each combination γ of destination groups. If a message is multicast to γ, then it is implemented as an ensemble-multicast to e_γ. To avoid requiring 2^n ensembles to accommodate all possible combinations of up to n groups, we decided to have $n{+}1$ ensembles: if a message m is multicast only to g_i, then it is ensemble-multicast to e_i; if m is multicast to multiple groups, it is ensemble-multicast to e_{all} (lines 4–7). Using this simple mapping, m will be ensemble-delivered by all processes if it has been multicast to more than one group; for this reason, m receives a tag $m.dests$ that contains m's actual destination groups (line 3). Upon ensemble-delivery of m, each process p checks if m is actually addressed to p, by comparing the set of m's destination groups with the set σ of groups that p subscribed to; if that is the case, m is delivered to p.

Algorithm 4 Ensemble-multicast with deterministic merging

```
1: Task 1 (proposer)
2: To ensemble-multicast message $m$ to ensemble $e$
3:     $m.ts \leftarrow$ unique timestamp based on local clock
4:     send $m$ to the coordinator of $e$
5: Task 2 (coordinator of ensemble e)
6: Initialization:
7:     $k \leftarrow 0$
8:     set timer to expire in $\Delta$
9: when receiving $m$ from a proposer
10:     $\text{propose}_e(k, m)$
11:     $k \leftarrow k + 1$
12:     stop timer
13:     set timer to expire in $\Delta$
14: when timer expires
15:     create empty $null$ message
16:     $null.ts \leftarrow$ unique timestamp based on local clock
17:     $\text{propose}_e(k, null)$
18:     $k \leftarrow k + 1$
19:     set timer to expire in $\Delta$
20: Task 3 (acceptor)
21: execute consensus (follow Algorithm 3)
22: Task 4 (learner)
23: Initialization
24:     $\forall e : k_e \leftarrow 0$
25:     $\forall e : last\_ts_e \leftarrow 0$
26:     $all\_decisions \leftarrow \emptyset$
27:     $gap\_free \leftarrow \emptyset$
28: when $\text{decide}_e(k, m)$
29:     $all\_decisions \leftarrow all\_decisions \cup \{\langle e, k, m\rangle\}$
30: when $\exists e : \langle e, k_e, m\rangle \in all\_decisions$
31:     if $m.ts < last\_ts_e$ then
32:         $m.ts \leftarrow last\_ts_e + 1$
33:     $last\_ts_e \leftarrow m.ts$
34:     $gap\_free \leftarrow gap\_free \cup \{\langle e, m\rangle\}$
35:     $k_e \leftarrow k_e + 1$
36: when $\forall e : l$ receives from $e$, $\exists m : \langle e, m\rangle \in gap\_free$
37:     take $\langle e_d, m_d\rangle$, where $m_d.ts$ is the lowest in $gap\_free$
38:     $gap\_free \leftarrow gap\_free \setminus \{\langle e_d, m_d\rangle\}$
39:     if $m_d \neq null$ then
40:         ensemble-deliver($m_d$)
```

Algorithm 5 Ridge - multicasting messages to multiple groups

```
1: Task 1 (multicaster)
2: To multicast $m$ to set of groups $\gamma$
3:     $m.dests \leftarrow \gamma$
4:     if $|\gamma| = 1 \land \gamma = \{g_i\}$ then
5:         ensemble-multicast $m$ to $e_i$
6:     else
7:         ensemble-multicast $m$ to $e_{all}$
8: Task 2 (destination)
9: Initialization
10:     Destination $p$ subscribes to set of groups $\sigma$
11:     $delivered \leftarrow \emptyset$
12:     for each $g_i \in \sigma$ do
13:         become a learner of ensemble $e_i$
14:     if $|\sigma| > 1$ then
15:         become a learner of ensemble $e_{all}$
16: when ensemble-deliver $m$
17:     if $m.dests \cap \sigma \neq \emptyset \land m \notin delivered$ then
18:         deliver $m$
19:         $delivered \leftarrow delivered \cup \{m\}$
```

5.3.3 Tolerating failures

Each Ridge ensemble is an implementation of Paxos, so leader election, failure detection and fault-tolerance are handled in the same way as the original protocol. What changes is the way messages are routed. Because of that, Ridge is sensitive not only to the failure of the coordinator, but also to the failure of the acceptors in Q and to the failure of distributing learners.

Coordinator failure

If the coordinator is suspected of failing [21], another process will take its place, and (eventually) run Phase 1 of Paxos with a round number higher than that used by the suspected coordinator [48].

Acceptor failure

In case the coordinator suspects that an acceptor of Q has failed during the execution of a consensus instance k, the coordinator falls back to original Paxos

(Algorithm 2). It re-executes the consensus instance k, including Phase 1 and Phase 2 of original Paxos, with a higher round number.

Learner failure

If the last acceptor of Q suspects of a failure of the distributing learner, the acceptor will forward the message to a different learner. In any case, learner failures may cause other learners not to receive some consensus decisions. When suspecting that decisions were not received, learners can check with the acceptors what has been decided so far and fill any possible gaps.

5.3.4 Optimistic deliveries

Ridge allows messages to be delivered optimistically, with the optimistic order likely matching the conservative order. For messages to be delivered optimistically and conservatively in the same order, the protocol relies on a few optimistic assumptions; if they do not hold, the orders may differ, but the correctness of the conservative delivery is not affected. The optimistic assumptions are the following: (a) there are no message losses between correct processes, (b) messages from each ensemble are decided in the order of their initial timestamps (thus their final timestamps will be the same as their initial ones) and (c) when a learner delivers a message m optimistically, it has already received all messages that could possibly have a timestamp lower than that of m.

Condition (a) is eventually guaranteed in our system model (Section 2.1), and it can be approximated in practice with a reliable communication protocol (e.g., TCP). We satisfy conditions (b) and (c) by having each process wait "long enough" before proposing a message or delivering a message optimistically. In the case of (b), the coordinator waits for a certain amount of time before proposing messages, allowing messages received out of order to be proposed in the correct (initial timestamp) order; as for (c), the wait time allows the learner to deliver optimistically in the correct order.

To perform optimistic deliveries, when a process p ensemble-multicasts a message m to an ensemble e, p first sends[2] m directly to all learners of e. After receiving m directly from p, each learner delivers m optimistically after waiting a certain time, which is based on the estimated latency and clock skews of the system. Besides sending m to the learners, p sends m also to the coordinator of e, which proceeds with Phase 2 of Paxos for m. Once the $(f+1)$-th acceptor

[2]The process multicasting m uses reliable multicast [30] to send the value to the learners and to the coordinator.

is reached, it only notifies the learners about the decision, by sending *id(m)* to them. At this point, learners are expected to have already received m directly from p; if that is the case, they can deliver m conservatively. If a learner still hasn't received m when receiving *id(m)*, the learner asks the acceptors for m. Note that the optimistic delivery does not guarantee uniformity: if both p and a learner l_f are faulty, it is possible that l_f delivers m optimistically, when no other process delivers m. However, this does not affect the correctness of the conservative delivery.

One question is how exactly to define the wait time mentioned above. Say process p sends a message m to process q. The wait time w should be such that, if p creates m at time t (i.e., m has timestamp t), m is received by q before instant $t+w$. The wait time at each process q should take into account the clock skew between q and other processes in the system. The reason for this is that the timestamp of each message is given based on the clock at the process that created the message. Clock skews may differ for different pairs of processes; we denote the difference between the clocks at p and q as $\epsilon(p,q)$. As for latency, we denote the time it takes for a message from p to arrive at q as $\delta(p,q)$. Let Π be the set of all processes in the system. If we take into account that clock skew and latency differs for each pair of processes, we have that the wait time at q, denoted as w_q, is an estimate of $max_{p\in\Pi}(\delta(p,q)+\epsilon(p,q))$. Such estimate is updated as different messages are received by q.

An interesting observation is that we do not need to know either the latency or the clock skew individually. Instead, we only need the sum of those two values. Finding such a sum for any given message m is as simple as subtracting the message's timestamp from the clock value at the destination when the message is received. One problem is that there might be a process that has much higher latency to communicate with the others, increasing the wait time for all processes in the system. To mitigate this problem, a maximum value can be set for w_q, for every process q. This may increase the number of mistakes when messages from a slow process are involved, but allows the wait time to stay under control, lowering the latency of optimistic deliveries.

Finally, Ridge must ensure the equivalence property, that is, that a correct process opt-delivers a message if, and only if, it also cons-delivers the same message. A simple way to provide equivalence is by keeping track of which messages were opt-delivered and which were cons-delivered. If, when a message m is cons-delivered at a process p, p has never opt-delivered m, p opt-delivers m immediately. Likewise, if a message is opt-delivered at a process p, but it is not cons-delivered after some time t, p sends m to the coordinator of the appropriate ensemble so that it can be decided and cons-delivered. Duplicate deliveries cannot happen because each process keeps track of what has already been delivered, optimistically and conservatively.

5.3.5 Correctness

In this section, we provide a proof for the correctness of Ridge. At its core, the protocol optimizes Paxos for high throughput, and then optimistic atomic multicast is implemented on top of Paxos. For this reason, we first show that the properties expected from consensus are guaranteed. Then, we prove that Ridge guarantees the properties that we defined for optimistic atomic multicast.

As described in Chapter 2, we assume a partially synchronous system: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called GST (Global Stabilization Time). Before GST, there are no bounds on message delay or relative process speed. After GST, such bounds exist but are unknown. We assume that after GST, all remaining processes are correct.

Properties of consensus

We prove here that the optimized version of Paxos implemented by Ridge (Algorithm 3) ensures the properties of consensus, namely uniform agreement and termination; uniform integrity holds trivially from the algorithm.

Proposition 2. (UNIFORM AGREEMENT) *No two processes decide different values.*

Proof. Let v and v' be two decided values, with unique identifiers *id(v)* and *id(v')*, and proposed by coordinators c and c', respectively. We prove that *id(v)* $=$ *id(v')*.

Let r be the round, with quorum Q, in which some acceptor a sent a DECISION message with *id(v)* to all learners, and let r' be the round, with quorum Q', in which some acceptor a' sent a DECISION message with *id(v')* to all learners.

In Ridge, a sends the DECISION message with *id(v)* after: (a) c receives $f+1$ messages of the form $\langle$PHASE 1B, r, *, *$\rangle$; (b) c selects value *v-val* $= v$ with the highest round number *v-rnd* among the set M_{1B} of PHASE 1B messages received, or picking a value *v-val* if *v-rnd* $= 0$; (c) $f+1$ different acceptors (a majority of the acceptors) have cast a vote for value v in round r.

When acceptor a receives a message of the form $\langle$PHASE $2, r, v, Q, f\rangle$, it is equivalent to a receiving a message $\langle$PHASE $2B, r, v\rangle$ (as in Algorithm 2) from f other acceptors. That's because each acceptor forwards a message of form $\langle$PHASE $2, r, v, Q, count\rangle$ to the next acceptor in Q only if r is higher than or equal to the previous *v-rnd* held by that acceptor. If a also casts a vote for $\langle r, v\rangle$, this is equivalent to a sending a message $\langle$PHASE 2$, r, v, Q, f+1\rangle$ to itself, so a knows that v was decided in round r, equivalently to coordinator c receiving $f+1$ $\langle$PHASE 2B$, r, v\rangle$ messages in the original Paxos algorithm. That's why a can safely send a $\langle$DECISION$, id(v)\rangle$ message to all learners. Let M_{2B} be the set of such equivalent $f+1$ Phase 2B messages. Now consider a system running original Paxos where the coordinator c received the same sets of messages M_{1B} and M_{2B}. In this case, c would send a DECIDE message with *id(v)* as well. Since the same reasoning can be applied to coordinator c', and Paxos implements consensus, we have that $id(v) = id(v')$. □

Proposition 3. (TERMINATION) *If one (or more) correct process proposes a value then eventually some value is decided by all correct processes.*

Proof. After GST, processes eventually select a correct coordinator c. Coordinator c considers a quorum Q of acceptors, and c sends a Phase 1A message to all acceptors in the ensemble. If all acceptors in the majority quorum Q are correct and c does not suspect that any of them has failed, then all messages exchanged between coordinator and acceptors in Q are received and all correct processes eventually decide some value. If c suspects that any acceptor in Q has failed [21], the protocol falls back to the original Paxos algorithm, which guarantees property (iii) of consensus. □

Properties of optimistic atomic multicast

We prove here that Ridge ensures the following properties of the conservative delivery of optimistic atomic multicast: validity, integrity, uniform agreement, and atomic order. We also prove that Ridge ensures the equivalence and optimistic order properties. Ridge implements the optimistic delivery with reliable multicast, so the validity, integrity and agreement properties of the optimistic delivery are ensured by reliable multicast [30]. The following proof applies to Algorithm 4 and can be trivially extended to Algorithm 5.

Lemma 1. *Let t be a timestamp, and ε a set of ensembles. Each correct process p that is a learner in every ensemble $e \in \varepsilon$ eventually decides a message m with timestamp $m.ts > t$, where m was decided by e, for every $e \in \varepsilon$.*

Proof. Each ensemble $e \in \varepsilon$ eventually elects a correct coordinator c after GST and proposes messages (possibly null) at least every Δ time units. Decided messages from an ensemble are assigned monotonically increasing final timestamps, so p eventually decides a message m with timestamp $m.ts > t$, for every ensemble e in ε. □

Proposition 4. (VALIDITY) *If a correct process multicasts m, then every correct process in γ delivers m.*

Proof. To multicast m, a correct process p sends m to the coordinator c of an ensemble e, and p keeps sending m until it reaches the correct coordinator. After GST, a correct process c is eventually elected as coordinator of e. Since both p and c are correct, c delivers m and proposes it. From the properties of consensus, all correct learners of e decide m. Therefore, and from Lemma 1, l eventually delivers m. □

Proposition 5. (INTEGRITY) *For any message m, every process p in γ delivers m at most once, and only if some process has multicast m previously.*

Proof. For m to be delivered, it must have been proposed by the coordinator c of an ensemble e. The proof that m was multicast by some process is immediate from Algorithm 4: c only proposes m after receiving m from a process q that multicast m. In Algorithm 5, each destinations keeps track of which messages were delivered already in set *delivered*. When a message is delivered, it is added to the *delivered* set, so that delivering the same message twice is impossible (lines 17–19 of Algorithm 5). □

Proposition 6. (UNIFORM AGREEMENT) *If a process delivers m, then every correct process in γ delivers m.*

Proof. For a process p, which subscribes to a set of ensembles ε, to deliver a message m, p must be a learner that decided m after m was accepted by a majority of acceptors in some ensemble $e \in \varepsilon$. Since e has a majority of correct acceptors, at least one correct acceptor a knows that m was decided at an instance k. After GST, eventually a correct coordinator c is elected for e, proposing messages every Δ time units at least. From the properties of consensus, every correct learner of e will decide such messages, knowing if there is any gap in the message stream from e. In other words, there will eventually be a message decided in instance k', where $k' > k$, in ensemble e. Since at least the correct acceptor a knows about m, m can be recovered by any correct learner l that missed m. This way, l is sure to eventually know about m, being able to deliver it once a message m', where $m'.ts > m.ts$, has been decided in each ensemble in ε, which is guaranteed to eventually happen after GST (Lemma 1). □

Proposition 7. (ATOMIC ORDER) *The order relation $\prec$ is acyclic. Also, for any two messages m and m' and any two processes p and q such that $\exists\{g,h\} \subseteq \gamma \cap \gamma' : \{p,q\} \subseteq g \cup h$, if p atomic-delivers m and q atomic-delivers m', then either p atomic-delivers m' before m or q atomic-delivers m before m'.*

Proof. The message stream from each ensemble is decided by consensus. Therefore, every process delivers messages from the same ensemble in the same order, or in a prefix of such an order. Moreover, the final timestamp of each message m is deterministically defined based only on previously decided messages from the same ensemble that decided m (lines 30–35 of Algorithm 4). Because of this, each message is assigned the same final timestamp in every destination. Since delivery is done in the order of final timestamps, there can be no cycle in the delivery order, even when messages come from different ensembles. Finally, the deterministic merge procedure in Ridge happens when there is at least one message from every ensemble in the *gap_free* queue. Since there is no gap in the message stream of any ensemble, and merge is done in the order of the deterministically defined final timestamps of messages, there is no gap in the merged sequence either. □

Proposition 8. (EQUIVALENCE) *If a correct process p delivers m optimistically, p also delivers m conservatively, and vice-versa.*

Proof. To ensure equivalence, Ridge does two things: (i) if, when a correct process p cons-delivers a message m, p hasn't opt-delivered m yet, p opt-delivers m immediately, and (ii) if t time units have elapsed since p opt-delivered a message m, and m has not been cons-delivered yet, p opt-amcasts m again, on its sender's behalf. Moreover, every process p keeps track of which messages it has opt-delivered and which messages it has cons-delivered.

It is trivial to see that the equivalence property is ensured in case (i); for case (ii), it comes from the validity property of optimistic atomic multicast (Proposition 4) and from the fact that p is correct. In case m is decided twice (e.g., because p wrongly inferred that m's source process q failed and decided to multicast it again on q's behalf), every destination will consider only the the first time m is decided, thus not violating atomic order. □

Proposition 9. (OPTIMISTIC ORDER) *If the given optimistic assumptions hold, the optimistic order does not contradict the conservative order.*

Proof. We prove here that the optimistic order matches the conservative order, given the following optimistic assumptions: there are no failures nor suspicions of failures, no message is cons-delivered before it is opt-delivered, and every process p has estimated a wait time w_p that is long enough. By "long enough", we mean that every message m sent to p with timestamp $m.ts$ is received by p before the clock at p reaches $m.ts + w_p$.

While there are no failures, every ensemble coordinator c will receive and propose all messages addressed to that ensemble. Since w_c is large enough,

c will be able to propose all messages for that ensemble in the same order of their initial timestamps; also, as we assume no failures or failure suspicions, those messages will be decided in the same order in which they were proposed, so the final timestamp of each of these messages will be the same as its initial timestamp. The conservative delivery order in Ridge follows the final timestamp order. As every process p has its w_p large enough, no message will arrive late at p for its optimistic delivery, so p will be able to opt-deliver all messages in the correct initial timestamp order, which will be the same as the conservative order. □

5.4 Performance analysis

In this section, we show that Ridge achieves high throughput in a point-to-point network. We first describe the performance model used here, then show that having a distributor for each different message can lead to optimal throughput, and then explain why it achieves low latency.

5.4.1 Performance model

Our performance model is based on the one used in [33] and [34]. In that model, each process can send one message per round and receive one message per round: if a single process p sends one message to a destination q, it will take a single round for q to receive the message. Likewise, if q has n messages to receive, it will finish receiving after n rounds. Although this is a very good model for throughput, it predicts latency as a function of throughput, which is not always realistic. For instance, in a 1 Gbps network with 50 μs average latency, a process can send up to 125 megabytes per second, or approximately 125 bytes per microsecond. If we consider 125-byte messages, the round length (time necessary to send one message) would be 1 μs. This model predicts that the message would take only 1 round (1 μs) to arrive at the destination. However, in reality it would take around 50 μs (or 50 rounds) in such a network.

We extend this performance model to decouple latency from throughput as follows. At each round k, every process can send only one message (we assume that the underlying network provides only one-to-one communication) and can also receive only one message. We define δ as the latency, in number of rounds, for a message to arrive from a process p at a process q, where $\delta > 0$. In more detail, at each round k, every process p_i can execute all or a subset of the following steps:

(1) compute a message for round k, $m(i,k)$,

(2) send $m(i,k)$ to one process,

(3) receive at most one message sent at round $k-\delta$.

5.4.2 Throughput

We consider a system with n processes: $p_0, p_1, ..., p_{(n-1)}$. To simplify the explanation, let p_i be the same process as $p_{(i \bmod n)}$, so p_n represents the same process as p_0, $p_{(n+1)}$ refers to p_1, and so on. Say all processes broadcast a message at the same time. Each process p_i broadcasts m_i to $p_0, ..., p_{(n-1)}$. At the beginning, process p_i already has its own message m_i, so at round 1, each p_i sends m_i to $p_{(i+1)}$; at round 2, p_i sends m_i to $p_{(i+2)}$, and so on. At the $(n-1)$-th round, p_i will send m_i to the last destination $p_{(i+n-1)}$. So, we have that, after $n-1$ rounds, the n processes will have each finished sending their messages to all n destinations, resulting in a throughput of $n/(n-1)$ messages sent per round. This shows that each process sending its own message directly to all other processes achieves the optimal throughput found in [34].

In the case of Ridge, we assume that Phase 1 of Paxos was pre-executed, so we focus on Phase 2. We consider that all n processes in the system are learners, so $L = \{p_0, p_1, ..., p_{(n-1)}\}$. For this analysis, we also assume that no failures happen nor are suspected to happen, so the majority quorum $Q = \{a_1, a_2, ..., a_{(f+1)}\}$ never changes, where a_1 is the coordinator. We show that, if the coordinator a_1 proposes one message per round, the throughput of the system is one message decided per round, which is the maximum rate of messages any process can receive from the network. With Ridge, Paxos Phase 2 is executed by having each acceptor a_i, where $1 \leq i \leq f$, receive one message m at each round and forward it to $a_{(i+1)}$, along with $id(m)$. Acceptor $a_{(f+1)}$ then sends m to l_d, which is a learner picked from $L \setminus Q$ at random (following a uniform distribution). Process l_d sends the message $|L \setminus Q|$ times, one for each of the remaining learners that have not received the message yet.[3] Each learner in $L \setminus Q$ is chosen to be a distributor $1/|L \setminus Q|$ times, so each learner sends one message per round on average. As every process receives one message and sends one message per round, we conclude that the throughput of the system is one message decided per round.

[3]Since l_d already has the message, it only sends it $|L \setminus Q|-1$ times, but it is simpler to approximate to $|L \setminus Q|$. On the other hand, $a_{(f+1)}$ sends $id(m)$ to all $|L|$ learners of the system, letting them know that m was decided, but we consider that the size of $id(m)$ is negligible.

5.4.3 Latency

Note that, following our modified performance model, the theoretical analysis of maximum system throughput using a ring and using a different process as distributor for each message remain both as $n/(n-1)$ messages per round. The theoretical latency, though, is different. In the case of a ring, the first process after the sender p_i in the ring will receive m_i after δ rounds, while the last process in the ring will receive m_i after $\delta(n-1)$, which is the time necessary to complete the broadcast. When using distributors, each process p_i sends its message m_i to all others, one destination after the other, in $n-1$ rounds. The first destination, $p_{(i+1)}$ will receive m_i in δ rounds. Process $p_{(i+2)}$, in $\delta + 1$, and so on. The last destination, $p_{(i+n-1)}$, will receive m_i in $\delta + n - 2$ rounds, which is the time needed to finish the broadcast. (This means a single communication step if we consider only the network latency between processes, i.e., if we assume that the time necessary to send a message is negligible.)

To give an example, suppose a wide-area system with 100 processes, where each has a 1 megabyte/s connection, both for sending and receiving, in full-duplex. The average latency of the system is 100 ms and every message has 1 kilobyte length. Following our model, n is 100, the round length is 1 ms (since each process can send roughly one thousand messages per second), and δ is 100 rounds. Using distributors, we have latency $\delta + n - 2 = 100 + 100 - 2 = 198$ ms. Using a ring, we would have $\delta(n-1) = 100 \times (100 - 1) = 9900$ ms.

With Ridge, it takes $|Q|\delta$ rounds for a message to pass through all the acceptors in the quorum Q and arrive at the distributing learner. From that point on, the learners in $L \setminus Q$ (i.e., the learners that are not acceptors in Q) take turns being the distributor, so $(\delta + |L \setminus Q| - 2)$ extra rounds are necessary for all remaining processes to receive the message. Therefore, the latency of Ridge is: $(|Q| + 1)\delta + |L \setminus Q| - 2$.[4]

5.5 Experimental evaluation

We present here the results of experiments with Ridge, (Multi-)Ring Paxos [53, 55], LibPaxos[5] and Spread [7]. In Section 5.5.1, we detail the environment used and the parameters given to the different protocols. In Section 5.5.2, we show

[4]If learners are delivering messages from multiple ensembles, it may be necessary to wait for Δ (maximum time between two consecutive null messages) more rounds.

[5]https://bitbucket.org/sciascid/libpaxos

the results for broadcast (i.e., multicast with a single group). In Section 5.5.3, we show how throughput scales with the number of multicast groups, for each protocol. Finally, in Section 5.5.4, we show results for optimistic deliveries.

5.5.1 Environment setup and configuration parameters

We conducted all experiments on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve 2920-48G gigabit network switch, and the Dell nodes were connected to another, identical switch. Those switches were interconnected by a 20 Gbps link. The average round-trip latency measured with ping was 133 μs. All nodes ran CentOS Linux 6.5 with kernel 2.6.32 and had the Oracle Java SE Runtime Environment 8. Clocks were kept synchronized with NTP for better results with Ridge and Multi-Ring Paxos with multiple groups and to collect consistent measurements from different processes in the system.

In all our experiments, there are clients and servers: each client multicasts a message to a group of servers, receives a reply from one of the servers, then sends another message, and so on. In all experiments with Paxos-based protocols, each Paxos group had 3 acceptors, with in-memory storage. One of the acceptors was also a proposer acting as coordinator and the servers were pure learners (i.e., being neither acceptors nor proposers). Each client message was sent to a coordinator, which proposed the message in the next consensus instance. In the case of Multi-Ring Paxos, we used a scheme similar to that of Ridge: there was a ring r_i for each multicast group g_i and a ring r_{all} for messages for multiple groups. Each process subscribing to messages from g_i would learn decision from r_i and r_{all}. Null and skip messages were sent every $\Delta = 1$ ms. LibPaxos implements Paxos in C. In LibPaxos, the coordinator sends each proposal directly to all acceptors, which then send their Phase 2B messages directly to the the learners. Upon receiving a Phase 2B message from a majority of acceptors, the learners declare the value as decided. This is done to minimize latency, although it may be detrimental to the maximum throughput. All Paxos-based protocols used TCP for communication.

As a reference, we included experiments with Spread [7], version 4.4.0. In our Spread deployments, each server had a local Spread daemon, and all daemons belonged to the same Spread segment (within a segment, daemons are arranged in a ring and send message payloads using ip-multicast). Each server joined one multicast group and had clients connected to it. The message service type used was "safe", which is equivalent to uniform total order [25].

5.5.2 Broadcast: performance vs. number of destinations

We show here the results of experiments with a single multicast group (i.e., broadcast). The throughput we report is the maximum, which we find by increasing the system load as long as the throughput increases. To report latency, we look for the load that leads the system to its maximum *power*, which we define as the ratio between throughput and latency. As load increases, power tends to increase as well, until latency increases faster than throughput, indicating that the system is overloaded and that latency values are no longer reliable.

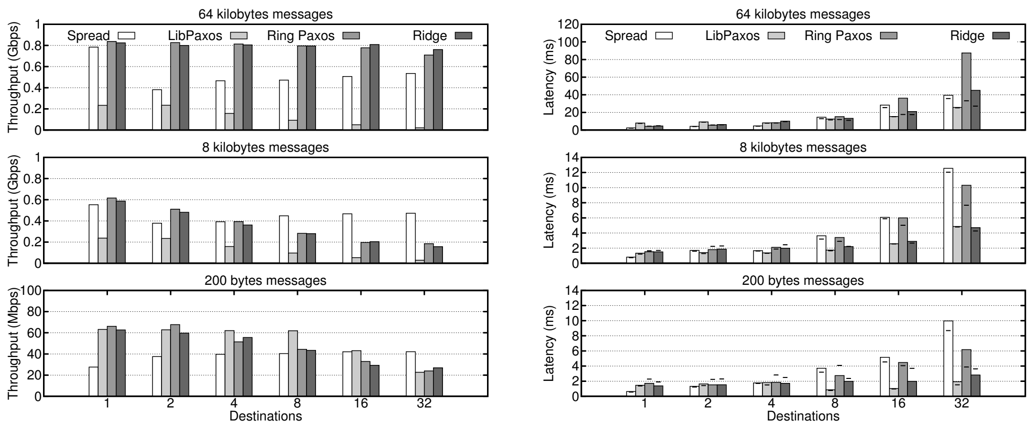

Figure 5.1. Throughput (maximum) and latency (with the system at maximum power) as the number of destinations increases. In the right-side graphs, bars and dashes represent the 95th percentile and average latency values, respectively.

Ridge has roughly the same throughput achieved by Ring Paxos. Figure 5.1 (left) shows that, for 64 kB messages, both algorithms reach 0.8 Gbps in a 1 Gbps network, with very little variation as the number of destinations increases. This comes from the fact that both algorithms use throughput-optimal dissemination techniques: a ring (Ring Paxos) and alternating distributors (Ridge). For 8 kB and 200 B messages, there is a throughput drop as the number of destinations increases. LibPaxos, which is implemented in C and is optimized for latency in detriment of throughput, has the lowest throughput of all algorithms, except for very small messages, where CPU is more likely to be the bottleneck than network. Spread uses ip-multicast for disseminating messages, having a somewhat constant throughput of 0.4–0.5 Gbps for 64 kB and 8 kB messages, and around

40 Mbps for 200 B messages. With 64 kB messages and a single destination, Spread had the same 0.8 Gbps throughput of Ridge. This is a special case for Spread, as there is a single server, i.e., a ring with a single Spread daemon.

Figure 5.1 (right) shows latency results. Bars and dashes represent the 95th percentile and average latencies, respectively. Among the protocols that use Paxos, LibPaxos had the lowest latency in most cases. However, Ridge's latency was comparable to that of LibPaxos. The latency of Spread was not as sensitive to message size as the latency of the other protocols tested, since Spread uses ip-multicast to disseminate payloads. Still, Ridge has lower latency than Spread in many cases. This happens because Spread uses a ring topology: although it uses ip-multicast to disseminate the payload of each message, such a message can only be safely delivered after relevant data has traversed the whole ring.

5.5.3 Multicast: performance vs. number of groups

In this section, we show how throughput scales with the number of groups in the system. In our experiments, each group had four servers, and each message was sent to a single group. Ideally, the aggregate throughput would scale linearly with the number of groups. We do not show multicast results for LibPaxos because it only offers a broadcast API (by means of consensus with Paxos).

Figure 5.2 (left) shows the scalability of maximum throughput for each pro-

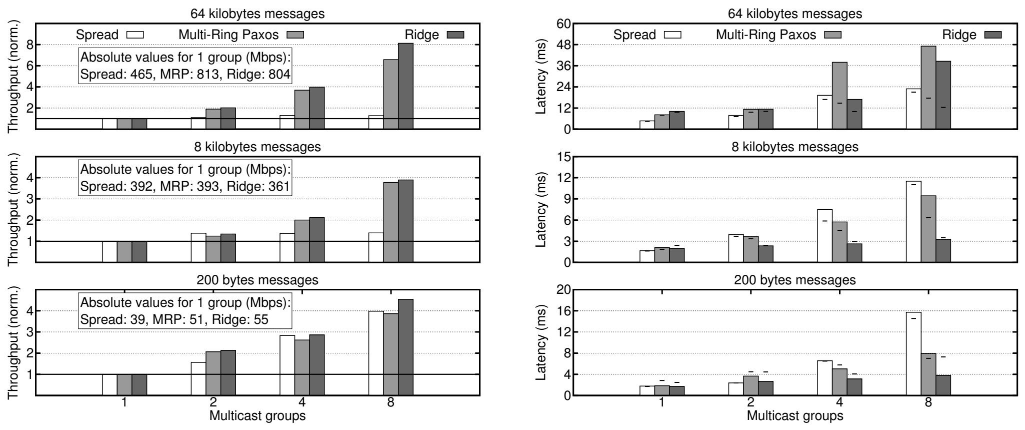

Figure 5.2. Throughput (maximum) and latency (with the system at maximum power) for each multicast protocol as the number of groups increases. The throughput of each protocol was normalized by the protocol's throughput with a single multicast group. In the right-side graphs, bars and dashes represent the 95th percentile and average latency values, respectively.

tocol, while latency is reported on Figure 5.2 (right). The maximum throughput of each protocol is normalized by its absolute value with a single group, which is shown in the graph. We can see that both Ridge and Multi-Ring Paxos have very similar throughput scalability in all cases. This is expected, since both algorithms use one separate group of Paxos acceptors for each multicast group. By doing this, these protocols are able to scale the aggregate throughput by adding independent sets of Paxos acceptors. This scheme allows Ridge and Multi-Ring Paxos to have nearly ideal scalability for 64 kB messages. Ridge's throughput scaled 8 times, proportionally to the number of groups. These algorithms had good scalability for 8 kB and 200 B messages, with throughput scaling 4 times with 8 groups. Spread uses a single ring for all servers in the system, even if they subscribe to different groups. Because of that, network was a bottleneck and Spread's throughput did not scale much beyond 0.5 Gbps with 64 kB and 8 kB messages. This result is similar to what Spread achieved for broadcast with multiple destinations (here, each group has four destinations). Spread scaled better with small messages, as network was likely not the bottleneck, and having more groups allowed Spread's throughput to increase.

5.5.4 Results for optimistic deliveries

We also benchmarked Ridge with optimistic deliveries enabled. We had a number of clients sending 64 kB messages to a multicast group, where a server delivered messages both conservatively and optimistically, comparing the sequences. A *mistake* happened whenever the i-th conservative delivery in the server did not match the i-th optimistic delivery. The results we found are displayed in Figure 5.3, where we show mistakes as a percentage of the optimistic deliveries. We also highlight the load with which the system had maximum power, i.e., the maximum ratio between throughput and latency (of the conservative delivery).

Figure 5.3 (top) shows that the optimistic delivery has significantly lower latency than the conservative delivery, and this difference tends to increase with the load on the system. This happens because each message is delivered optimistically after being received directly from its origin, while the conservative delivery of a message only happens after the message has been decided in consensus. In Figure 5.3 (bottom), we can see that the rate of mistakes tends to increase with the load, which is expected: more messages lead to higher chance of out-of-order optimistic deliveries. However, even with the system heavily loaded, the percentage of mistakes never reached 3%.

We can see that delivering every message twice did not hurt throughput. Ridge's maximum throughput of 0.8 Gbps was the same found previously (Sec-

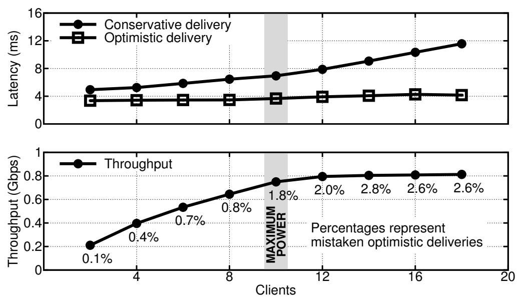

Figure 5.3. Latency, throughput and mistakes with the optimistic delivery enabled.

tion 5.5.2) without optimistic deliveries. With optimistic deliveries enabled, Ridge relies on the process multicasting a message to send (with reliable multicast [30]) the actual message payload to each Paxos learner directly. This is done to create a route that bypasses consensus, allowing optimistic deliveries to happen. When this happens, the acceptors do not send the message payload to the learners, but only its id, so that the learners do not waste downstream bandwidth to receive the same message twice. Although each individual client sends messages multiple times (to the learners directly, then to the Paxos coordinator), the throughput of the system remains unaltered.

5.6 Related work

In [34], the authors proved that ring topologies allow systems to achieve optimal throughput. Some protocols that benefit from such topologies are LCR [34], Totem [6], Spread [7] and Ring Paxos [53]. LCR arranges processes in a ring and uses vector clocks to ensure total order. One disadvantage of LCR in comparison to Ridge is that it requires *perfect failure detection*: suspecting that a correct process failed is not tolerated. Totem is a ring-based protocol based on Transis [5] that provides total order for messages. Spread, which is based on Totem, relies on daemons interconnected as a ring to order messages, while message payloads are disseminated using ip-multicast. Finally, Ring-Paxos deploys Paxos [48] processes in a ring to maximize throughput. A problem of all such ring-based protocols is that their latency is proportional to the size of the system times the network point-to-point latency. S-Paxos [16] is an atomic broadcast protocol that achieves high throughput by offloading the coordinator. Unlike Ridge, S-Paxos does not implement atomic multicast and does not optimize for latency.

Skeen's algorithm (presented in [17]) is possibly the first atomic multicast algorithm. Even though it is not fault-tolerant, it is *genuine*: processes communicate only if they actually have application messages to exchange [32]. In Skeen's algorithm, the destination processes of a message m exchange timestamps and eventually decide on m's final timestamp. Messages are then delivered based on their final timestamps. Several extensions have been proposed to Skeen's protocol, aiming to provide fault-tolerance [29, 32, 60, 63]. The basic idea of such extensions is to replace each process with a fault-tolerant group of processes that act as one single entity by means of consensus, as in state-machine replication [46].

Multi-Ring Paxos [9, 55] achieves very high throughput by increasing the number of multicast groups. Each group uses Ring Paxos to solve consensus and message streams from different groups are merged as proposed in [4]. Ridge also uses Paxos and deterministic merge, but it does not use a ring overlay. Instead, Ridge employs alternating distributors, being also capable of reaching very high throughput. Multi-Ring Paxos merges messages from multiple rings in a round-robin fashion, assuming that all rings produce the same number λ of decisions per time unit; if a ring does not have enough application messages to reach λ, *skip-messages* are created. If a ring produces more than λ decisions per time unit, performance likely deteriorates. Ridge's merge function is different, using timestamps taken from the processes' system clocks; these timestamps are also used to deliver messages optimistically. The merge functions of both Ridge and Multi-Ring Paxos are sensitive to clock synchronization: the better the clocks are synchronized, the lower the average latency is. Moreover, both the theoretical analysis and our experiments show that Ridge has throughput similar to that of Multi-Ring Paxos, with significantly lower latency for the conservative delivery, and even lower latency for the optimistic delivery.

Regarding optimistic deliveries, optimistic atomic broadcast [58] relies on messages being spontaneously delivered through the network in the same order at all destinations. If this does not happen, the algorithm runs consensus to ensure that a final, totally-ordered delivery is also done. In [70], the authors propose a technique that approximates spontaneous ordering in a wide-area setting. The idea is for each process to insert artificial delays in incoming messages, so that the resulting artificial latency between each process and a given sequencer is the same. Fast Paxos [50] allows messages to be sent directly to the acceptors (bypassing the coordinator), saving time. However, if the acceptors receive those messages in different orders, classic Paxos is run to ensure total order. In [11], we proposed an optimistic atomic multicast algorithm. Such algorithm is *quasi-genuine*, in the sense that processes only communicate if they "are able to" multicast messages to each other. The system's configuration determines which processes can communicate with one another, and the configuration can change dynamically. Such optimistic atomic multicast protocol does not optimize throughput, unlike Ridge.

5.7 Conclusion

This chapter introduced the optimistic atomic multicast class of ordering protocols, and presented Ridge, an optimistic atomic multicast protocol that combines high throughput, low latency, scalability and optimistic deliveries. Unlike previous works, Ridge achieves high throughput without resorting to ring topologies nor to ip-multicast. Instead, it makes use of alternating distributors, which is a technique capable in theory of achieving optimal throughput, while having lower latency than a ring. We conducted a number of experiments that prove that this also holds in practice, with different numbers of destinations and multicast groups. Finally, we demonstrated that enabling optimistic deliveries allows messages to be delivered faster with even lower latency than that of the conservative delivery. Despite the fact that each message is sent through different routes and delivered twice, enabling optimistic deliveries does not reduce the maximum throughput of the system.

In the next chapter, we show how optimistic atomic multicast can be used to provide fast replies to client requests. Also, we show how the optimistic delivery can be used to produce even faster replies, at the cost of server replicas possibly making mistakes.

Chapter 6

Fast Scalable State Machine Replication

Anyone can build a fast CPU.
The trick is to build a fast system.
Seymour Cray

As we have seen in Chapter 4, the time spent with request ordering (e.g., atomic multicast) is a significant component of the response time in S-SMR. Low-latency atomic multicast protocols, such as Ridge (described in Chapter 5), can lower such a latency by reducing the time spent ordering requests. However, it should be possible to provide even faster replies by relaxing the consistency level. Such a faster, possibly inconsistent reply, could be provided *in addition* to the final reply, so that clients could see the likely result of their requests in a shorter time.

This chapter introduces Fast-SSMR, a state machine replication technique that uses optimistic atomic multicast and provides an *optimistic reply*, which is potentially inconsistent, while the client waits for the final, *conservative reply*. Conservative replies can be used by clients to confirm optimistic replies. Fast-SSMR extends S-SMR, providing the same guarantees of its predecessor, that is, high availability combined with strong consistency and scalability. Conservative replies are always guaranteed to be strongly consistent, while optimistic replies rely on the underlying optimistic atomic multicast protocol not making mistakes in the optimistic delivery.

Fast-SSMR implements two state machines at each server replica: an *optimistic state machine*, which executes commands as they are optimistically delivered, and a *conservative state machine*, which waits for the conservative delivery to execute each command. Previous optimistic replicated designs (e.g., [37, 41, 44, 54]) require replicas to recover from commands executed in the wrong order, when the optimistic and the conservative orders do not match. Fast-SSMR does not have this constraint: in case of order mismatches, a correct state is copied from the conservative state machine to the optimistic one. Moreover, our technique generalizes optimistic execution to partitioned-state systems, dealing with problems not faced in fully replicated optimistic approaches, such as handling dependencies between states of different partitions and the exchange of optimistic state.

In S-SMR, state machines of different partitions may have to exchange state when executing commands that access multiple partitions. In Fast-SSMR, both conservative and optimistic state machines execute each command, so Fast-SSMR can operate in two modes: with and without optimistic state exchange. Without exchanging optimistic state, the optimistic state machine waits for the conservative state machine to execute the command and receive the remote state; then, the received conservative state is used by the optimistic state machines when executing the command. Since the received conservative state is guaranteed to be correct, this significantly simplifies the optimistic execution of multi-partition commands. However, it may increase latency, as the optimistic state machines wait for the conservative ones to execute the command. If optimistic state exchange is enabled, execution time is likely shorter, but the possibility of the received state being invalid must be taken into account. An invalid optimistic state can be received even if the local optimistic delivery order of optimistic atomic multicast was correct, making the verification of the optimistic execution significantly more complex.

Applications that involve user interaction (e.g., social networks) are well-suited for Fast-SSMR: before the conservative execution finishes, the client can receive the likely result of the computation, based on the optimistic reply. Although this may expose clients to temporary inconsistencies, we made this design choice based on the importance of response time for user experience and business revenues [15, 18, 74]. Alternatively, Fast-SSMR could be configured in a less aggressive way, providing only consistent replies. In such a case, servers would execute commands as soon as they were optimistically delivered, but the reply would be sent only if the optimistic delivery order was confirmed by the conservative one. Latency would still be lower than the original S-SMR because the reply to each command would be sent immediately after its conservative delivery. This would be possible because such a reply would have been already precomputed, based on the (earlier) optimistic delivery.

The dual state machine approach used by Fast-SSMR is service-independent and does not require an environment that supports recoverable actions (e.g., transactions) or the implementation of service-specific recovery. Running two state machines, however, requires additional processing and memory resources. We judge that this cost in resources is offset by the flexibility of the approach and the resulting simplification it brings in the design of applications. Moreover, one could argue that running two state machines as separate threads makes better use of multi-core architectures than SMR or S-SMR, which typically have a single execution thread per replica. As for memory overhead, widely deployed techniques, such as copy-on-write [68], can mitigate the effects of maintaining two state machines. Finally, to minimize the cost of copying the conservative state onto the optimistic one, Fast-SSMR copies only the divergent portion of the state.

To assess the performance of Fast-SSMR, we extended Eyrie, our Java library that implements S-SMR, and developed Chirper, which is a scalable Twitter-like social-network application. In the experiments we conducted with Chirper, throughput scaled with the number of partitions, in some cases linearly. These experiments also demonstrated that the optimistic replies of Fast-SSMR have significantly lower latency than the conservative replies. This comes at the cost of a low number of inconsistent optimistic replies.

The remainder of this chapter is organized as follows. In Section 6.1, we present baseline Fast-SSMR, a basic version of our optimistic technique. We provide a detailed algorithm in Section 6.2. In Section 6.3, we extend the baseline Fast-SSMR algorithm with a technique to speed up the execution of multi-partition commands, by allowing partitions to exchange optimistic state when executing multi-partition commands. Section 6.4 argues for the correctness of the algorithm. Section 6.5 shows how Eyrie was extended to implement Fast-SSMR and describes Chirper, our scalable social network application. Section 6.6 presents the results we have found with Chirper, also comparing Fast-SSMR to S-SMR. Section 6.7 surveys related works in the area of optimistic replication techniques. The chapter is concluded in Section 6.8.

6.1 Baseline Fast-SSMR

The baseline Fast-SSMR is inspired by previous optimistic designs to reduce the latency of replication (e.g., [37, 41, 44, 54]). Some replicated systems share a similar execution pattern to exploit optimistic message ordering: (a) commands are exposed to the service before their order is final (i.e., optimistically delivered); (b) commands are optimistically executed by the service while the ordering protocol determines the final order of the commands (i.e., conservative delivery); (c) if the optimistic and the conservative orders do not match, commands must be re-executed in the order established by the conservative delivery. Like other optimistic replicated systems, baseline Fast-SSMR reduces latency by overlapping the execution of commands with their ordering. Differently from previous techniques, Fast-SSMR generalizes optimistic execution to partitioned-state systems, while providing linearizability.

Optimistic replicated designs require the ability to recover from commands executed in the wrong order. Previous approaches have tackled this issue by relying on the recoverability of the environment in which the service is deployed (e.g., transaction support [37, 41]) or by explicitly implementing a recovery procedure, either "physical recovery" (e.g., checkpoints [44]) or "logical recovery" (e.g., application-dependent recovery [54]). We propose a different technique, consisting of two state machines that run concurrently: a *conservative state machine*, which implements S-SMR and whose execution is always correct, and an *optimistic state machine*, which relies on optimistic delivery to speed up the execution. Repairing the state of the optimistic state machine, in case of commands delivered out-of-order, boils down to copying the conservative state (or part of it) over the optimistic state. This dual state machine approach is service-independent and does not require an environment that supports recoverable actions or the implementation of service-specific recovery, although it requires additional processing and memory resources. Since current servers are typically multi-processor, each state machine can be assigned to a different processor, lessening the processing overhead. The memory overhead is mitigated by widely deployed OS techniques like copy-on-write [68]. Finally, we note that Fast-SSMR only copies the divergent portion of the state when performing a repair.

Most of the complexity of the Fast-SSMR algorithm lies in repairing the optimistic state in case of order violations. When an order violation is detected, the optimistic state machine pauses the execution of commands. To repair the optimistic state, the conservative state machine needs to "catch up" with the optimistic state machine. From the equivalence property of optimistic atomic multicast (defined in Chapter 5), eventually every command delivered optimistically by a correct process will be delivered conservatively by the same process. Therefore, once the optimistic execution pauses, eventually the conservative state machine will catch up. After the optimistic state is repaired, any optimistically delivered command that is already included in the copied state is discarded.

Besides the repair procedure, the two state machines differ in another important aspect. Instead of exchanging optimistic state when executing a *read*(v)

operation for a remote variable v, the optimistic state machine waits for the conservative state machine to execute the command and receive the conservative state of v from the remote partition; the received conservative state of v is then used by the optimistic state machine. This procedure simplifies the optimistic state machine as only valid state is exchanged across partitions—we introduce optimistic state exchange in Section 6.3. Both the optimistic and the conservative state machines send replies to clients; the conservative reply for a command allows the client to confirm the optimistic reply, likely to have been received earlier.

6.2 Detailed baseline algorithm

Algorithm 6 is executed by the conservative state machine. It is essentially the same as that of S-SMR (Algorithm 1), except that it uses optimistic atomic multicast and it keeps track of the conservative execution order. Algorithm 7 details the execution of the optimistic state machine, which shares variables of the conservative state machine's algorithm. For brevity, we say that a command is "opt-executed" and "cons-executed" meaning that it was executed by the optimistic and the conservative state machines, respectively.

The core idea behind Algorithm 6 and Algorithm 7 is to compare the sequence of commands executed by the optimistic state machine to the sequence of commands executed by the conservative state machine, one by one. Those sequences are kept in the ordered sets *opt_executed* and *cons_executed*, respectively. Commands are appended to those sets as they are executed by the different state machines, and are removed as the optimistic execution order is confirmed (or a mistake is detected and the optimistic state is repaired). If the optimistic execution order at a server was the same as the conservative one, the optimistic state is correct. This condition suffices to determine that the optimistic state at a server is correct because mistaken optimistic deliveries at a server cannot cause the optimistic state of other servers to be incorrect—baseline Fast-SSMR does not exchange optimistic state between servers when executing multi-partition commands; instead, it uses the conservative state exchanged by the conservative state machines. This way, when the first element of both *opt_executed* and *cons_executed* is the same, this means that both state machines reached the same state after executing that command, which is removed from both sets. On the other hand, if the first element of *opt_executed* differs from that of *cons_executed*, this means that there was an ordering mistake and the *repairing* flag is set to *True*, signaling that the optimistic state machine is under repair and no command will be opt-executed before the repair is finished. To repair its state, the optimistic

Algorithm 6 Conservative State Machine (Fast-SSMR)

```
1: Initialization:
2:    $cons\_executed \leftarrow$ empty ordered set
3:    $\forall C \in \mathcal{K} : rcvd\_signals(C) \leftarrow \emptyset$
4:    $\forall C \in \mathcal{K} : rcvd\_variables(C) \leftarrow \emptyset$
5: Command C is submitted by a client as follows:
6:    $C.dests \leftarrow oracle(C)$
7:    opt-amcast($C.dests, C$)
8:    wait for reply
9: Server s of partition $\mathcal{P}$ executes command C as follows:
10:    **when** cons-deliver($C$)
11:       reliable-multicast($C.dests, signal(C)$)
12:       **for** each operation $op$ in $C$ **do**
13:          **if** $op$ is $read(v)$ **then**
14:             **if** $v \in \mathcal{P}$ **then**
15:                reliable-multicast($C.dests, \langle v, C.id \rangle$)
16:             **else**
17:                **wait until** $v \in rcvd\_variables(C)$
18:                update $v$ with the value in $rcvd\_variables(C)$
19:          execute $op$
20:       **wait until** $rcvd\_signals(C) = C.dests$
21:       send reply to client
22:       append $C$ to $cons\_executed$
23:    **when** reliable-deliver($signal(C)$) from partition $\mathcal{P}'$
24:       $rcvd\_signals(C) \leftarrow rcvd\_signals(C) \cup \{\mathcal{P}'\}$
25:    **when** reliable-deliver($\langle v, C.id \rangle$)
26:       $rcvd\_variables(C) \leftarrow rcvd\_variables(C) \cup \{v\}$
```

Algorithm variables:

$\mathcal{K}$: the set of all possible commands

$C.id$: unique identifier of command C

$oracle(C)$: function that returns a superset of the partitions accessed by C

$C.dests$: set of partitions to which C is multicast

$signal(C)$: signal exchanged to ensure linearizability

$rcvd\_signals(C)$: set of all partitions that already signaled $\mathcal{P}$ regarding C

$rcvd\_variables(C)$: set of all variables received from other partitions in order to execute C

$cons\_executed$: commands conservatively executed, in order of execution

Algorithm 7 Optimistic State Machine (Fast-SSMR) (part 1)

```
 1: Initialization:
 2:     $opt\_queue \leftarrow opt\_executed \leftarrow$ empty ordered set
 3:     $repairing \leftarrow False$, $skip\_opt \leftarrow \emptyset$
 4:     $\forall C \in \mathcal{K} : rcvd\_opt\_signals(C) \leftarrow \emptyset$
 5: Server $s$ of partition $\mathcal{P}$ executes command $C$ as follows:
 6:     when opt-deliver($C$)
 7:         reliable-multicast($C.dests, opt\_signal(C)$)
 8:         append $C$ to $opt\_queue$
 9:     when $opt\_queue \neq \emptyset \wedge not\ repairing$
10:         remove first element $C$ of $opt\_queue$
11:         if $C \notin skip\_opt$ then
12:             append $C$ to $opt\_executed$
13:             execute-opt($C$)
14:     when $opt\_executed \neq \emptyset \wedge not\ repairing$
15:         $C \leftarrow$ first element of $opt\_executed$
16:         if $C$ is the first element of $cons\_executed$ then
17:             // $C$ was correctly opt-executed
18:             remove $C$ from $opt\_executed$
19:             remove $C$ from $cons\_executed$
20:         else if $cons\_executed \neq \emptyset$ then
21:             $repairing \leftarrow True$
22:     when $repairing \wedge \exists C : C \in opt\_executed \cap cons\_executed$
23:         remove $C$ from $opt\_executed$
24:         remove $C$ from $cons\_executed$
25:     when $repairing \wedge opt\_executed = \emptyset$
26:         copy conservative state onto optimistic state
27:         $skip\_opt \leftarrow skip\_opt \cup cons\_executed$
28:         $cons\_executed \leftarrow \emptyset$
29:         $repairing \leftarrow False$
30:     when reliable-deliver($opt\_signal(C)$) from partition $\mathcal{P}'$
31:         $rcvd\_opt\_signals(C) \leftarrow rcvd\_opt\_signals(C) \cup \{\mathcal{P}'\}$
```

Algorithm 7 Optimistic State Machine (Fast-SSMR) (part 2)

```
32: Server $s$ of partition $\mathcal{P}$ executes command $C$ as follows:
33:   function execute-opt($C$)
34:     for each operation $op$ in $C$ do
35:       if $op$ is $read(v)$ then
36:         if $v \notin \mathcal{P}$ then
37:           wait until $v \in rcvd\_variables(C) \vee repairing$
38:           if $repairing$ then
39:             exit function
40:           $v_{opt} \leftarrow v : v \in rcvd\_variables(C)$
41:         execute $read(v_{opt})$
42:       else if $op$ is $write(v, val)$ then
43:         execute $write(v_{opt}, val)$
44:       else
45:         execute $op$
46:     wait until $rcvd\_opt\_signals(C) = C.dests$
47:     send optimistic reply to client
```

Additional variables:

v_{opt}: the optimistic copy of variable v

repairing: tells whether the optimistic state is under repair

opt_queue: commands waiting to be opt-executed

opt_executed: commands whose optimistic execution has not been confirmed yet

$opt\_signal(C)$: signal exchanged to provide linearizability

$rcvd\_opt\_signals(C)$: partitions that signaled $\mathcal{P}$ about C

skip_opt: commands to be skipped

state machine first waits until the conservative state machine has caught up with the optimistic execution, that is, until $opt\_executed = \emptyset$ (commands are removed from that set as they are executed by the conservative state machine). This is sure to eventually happen at any correct replica thanks to the equivalence property of optimistic atomic multicast: any command that is opt-delivered is also cons-delivered, and vice-versa. Once the command is cons-delivered, it is executed by the conservative state machine.

Once the conservative state machine has caught up with the optimistic execution during a repair (i.e., $opt\_executed = \emptyset$), there are two possibilities:

(i) All commands that were cons-executed were also opt-executed, i.e., $cons\_executed = \emptyset$.

(ii) Some command that was not executed by the optimistic state machine was cons-executed, i.e., $\mathit{cons\_executed} \neq \emptyset$.

In case (i), both state machines executed the same commands, although in different orders. The optimistic state machine then simply copies the conservative state and resumes execution with the next command opt-delivered. In case (ii), not only the order was different, but also the set of commands executed. To deal with this, those commands that were not opt-executed yet, but already cons-executed, are put in a set to be skipped by the optimistic state machine. If one of these commands changes the state of the service, the state copied from the conservative state machine already contains any such changes. Reexecuting them might cause the optimistic state to become incorrect. To illustrate how Algorithm 7 works, suppose the following events happen at a replica (the command execution sequences after each event are shown):

1. Command C_3 is opt-executed.
   $\mathit{opt\_executed} = (C_3)$, $\mathit{cons\_executed} = \emptyset$

2. Command C_1 is opt-executed.
   $\mathit{opt\_executed} = (C_3, C_1)$, $\mathit{cons\_executed} = \emptyset$

3. Command C_1 is cons-executed.
   $\mathit{opt\_executed} = (C_3, C_1)$, $\mathit{cons\_executed} = (C_1)$

   At this point, the first element of *opt_executed* (C_3) is different from that of *cons_executed* (C_1). This triggers a repair and the optimistic state machine will pause, waiting until every command in *opt_executed* has also been cons-executed. This is already the case for C_1, which is removed from both sets.

4. Command C_2 is cons-executed.
   $\mathit{opt\_executed} = (C_3)$, $\mathit{cons\_executed} = (C_2)$

5. Command C_3 is cons-executed.
   $\mathit{opt\_executed} = \emptyset$, $\mathit{cons\_executed} = (C_2)$

   As soon as C_3 is executed, it is removed from both ordered sets, leaving *opt_executed* empty. This means that the conservative state machine has caught up with the optimistic one. However, C_2 was not opt-delivered yet. The conservative state is copied to the optimistic one, but C_2 was already executed (by the conservative state machine) against this state, so the optimistic state machine must skip C_2 once it is delivered. For this reason,

C_2 is put in the *skip_opt* set, so that the optimistic state machine does not execute it.

We now give an "operational" description of baseline Fast-SSMR. In Algorithm 7, when a server opt-delivers a command (line 6) it appends the command to a queue (line 8) from which commands are later removed to be executed (lines 10–13). For every *read*(v) operation during the execution of a command (line 34), the server checks whether v is local (line 36); if not, the server waits until the value of v is received from a remote conservative state machine. Once v has arrived, its value is used to update v_{opt}, the optimistic copy of v (line 40). When reading (or writing) a variable v, the optimistic state machine reads (or writes) v_{opt} instead, thus accessing only the optimistic state (lines 41–43). After all operations have been executed, the optimistic reply is sent to the client that issued the command (line 47).

The rest of the algorithm deals with checking for order violations (lines 14–21) and with repairing the optimistic execution (lines 22–29). The *repairing* flag indicates whether the optimistic state machine is in normal operation or in repair mode. During normal operation, the algorithm keeps verifying the optimistic execution order, saved in *opt_executed*, by comparing it with the conservative execution order, kept by the conservative state machine in *cons_executed*. To verify that the optimistic and the conservative orders match, the first command of those sequences are compared: if they are the same, the command is removed from both sequences (lines 16–19); if not, the repair procedure is initiated by setting the *repairing* flag to *True* (lines 20 and 21), which also pauses the optimistic execution (line 9).

The repair procedure consists of waiting until all opt-executed commands have been cons-executed. Until then, there are two possibilities for each command C: (i) C was opt-executed out-of-order, or (ii) C was cons-executed and not opt-executed yet. Case (i) is handled by removing C from both *opt_executed* and *cons_executed* (lines 22–24). In case (ii), at the end of the repair procedure (line 25), that is, when *opt_executed* is empty, C remains in *cons_executed*, since it was never opt-executed. C is then moved to *skip_opt* (line 27), so that it will not be opt-executed after the repair is complete. This is done because at the end of a repair, the optimistic state is overwritten with the conservative state (line 26), which is already based on C. Finally, the *repairing* flag is set to *False* so that the optimistic execution can resume.

One observation is that the repair algorithm presented here could be refined, by taking into account that some order violations do not cause inconsistencies. For instance, commands that access different sets of variables could be executed in any order. However, accounting for such fine-grained checks would increase the complexity of the algorithm. The algorithm we presented works for any kind of commands and is fairly easy to understand.

6.3 Optimistic state exchange

In the baseline Fast-SSMR, partitions exchange only conservative state when executing multi-partition commands. This simplifies the optimistic state machine since no provision is needed to cope with invalid state received from another partition. Optimistic state exchange speeds up the execution, at the cost of a more complex design, needed to detect and handle invalid state exchanged among partitions due to the possibility of messages delivered out of order.

Figure 6.1 illustrates how exchanging optimistic state increases the complexity of the optimistic execution. The figure shows servers s_1, s_2 and s_3, respectively of partitions $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, as they execute optimistically and conservatively commands C_1, C_2 and C_3. Server s_1 optimistically delivers and executes C_2 before C_1, but s_1 conservatively delivers and executes commands C_1 before C_2. Because of this, s_1 sends invalid state to s_2, which also executes C_2. Even though the optimistic and conservative delivery orders match at s_2, s_2 reaches an invalid state after executing C_2. Even commands that were optimistically delivered in the correct order in all replicas can be "contaminated" by remote invalid states. Consider, for example, command C_3, which involves s_2 and s_3. As we can see, the optimistic and conservative delivery orders match in both servers. The problem is that C_3 was executed by s_2 after C_2, which in turn was executed based on an invalid state received from s_1. This may cause the state read by C_3 from $\mathcal{P}_2$ (and sent to s_3) to be invalid.

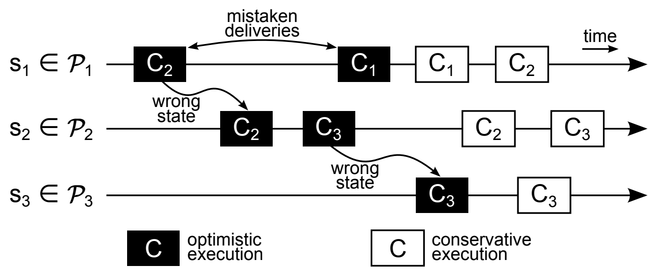

Figure 6.1. Invalid optimistic states propagated across partitions.

For a server s to determine that the optimistic execution of a command C is correct, s must (i) optimistically deliver C in the correct order and (ii) receive a confirmation from each remote partition involved in the command that the state s received from the partition is valid. Moreover, to handle contaminated states, commands that precede C must be confirmed as well. For this reason, a server s only sends a confirmation regarding the local execution of command C to other partitions after the execution of every command C' executed by s before C is confirmed. If a server detects that the optimistic execution of a command was based on invalid state, the server switches to repairing mode and proceeds as described in Section 6.2.

6.4 Correctness

In this section, we argue that the Fast-SSMR algorithm ensures linearizability and is deadlock-free.

Proposition 10. *Fast-SSMR ensures linearizability.*

Proof. The conservative state machine guarantees linearizable executions since it implements S-SMR. The optimistic state machine (Algorithm 7) ensures linearizability for optimistic replies, as long as the optimistic delivery order matches the conservative one. Both the conservative and optimistic state machines exchange signals in order to provide linearizability, as explained in detail in Chapter 4. □

Proposition 11. *Fast-SSMR is deadlock-free.*

Proof. There are two situations in which a state machine pauses and waits for some event: (i) when waiting for a signal and (ii) when waiting for the value of a variable to execute a multi-partition command.

Both optimistic and conservative deliveries have the agreement property (as described in Chapter 5). Therefore, when executing a command C, all correct replicas of all partitions in $C.dests$ deliver C. Each replica will reliable-multicast a signal to all servers in $C.dests$ as soon as C is delivered. Therefore, from the validity property, every replica will reliable-deliver at least one signal from each partition in $C.dests$. This means that no replica will block forever waiting for a signal. This argument is valid for both the optimistic and the conservative state machines in case (i).

As for case (ii), let us first assume that the optimistic order matches the conservative order, as the argument is the same for both state machines. Every multi-partition command C delivered by some server is delivered by all correct servers in $C.dests$. Each variable v accessed by C will be reliable-multicast by at least one server of the partition that contains v. Now, say command C accesses variables in multiple partitions. It is impossible to have servers $s_1, ..., s_n$, such that each server s_i (of partition $\mathcal{P}_i$) waits for a variable stored in $\mathcal{P}_{i+1}$, where $i \in \{1, ..., n-1\}$, while s_n waits for a variable stored in $\mathcal{P}_1$. This follows from the fact that C is deterministic and all servers execute the same operations in the same order: for each such operation, if the operation is *read*(v), at least one server of v's partition will send its value to the other servers executing C. Therefore, there is no deadlock in the conservative state machine. Also, there is no deadlock in the optimistic state machine when the conservative and the optimistic delivery orders match.

If the optimistic delivery order does not match the conservative delivery order, it is possible that an optimistic state machine s_{opt} (at server s) waits for a variable that will never be sent from a remote partition. For example, say s_{opt}, due to a previous optimistic ordering mistake, has a variable *index* with value i, while its correct value is j, where $j \neq i$. Then, s_{opt} delivers C, which reads *index* and, then, reads v_{index}. Since *index* is i, s_{opt} tries to access v_i, which belongs to partition $\mathcal{P}_i$, thus waiting for its value to be received from a server in $\mathcal{P}_i$. However, this value will never be sent: no conservative state machine executes C with *index* equal to i. Therefore, s_{opt} never receives v_i. To circumvent this problem, Algorithm 7 stops waiting if an ordering mistake is detected (i.e., when *repairing* is set to *True*). When a mistake is detected, the optimistic state machine stops the execution of C (i.e., it exits the execute-opt function) and the optimistic state is repaired. This prevents deadlocks from happening in case (ii), even when optimistic delivery mistakes happen. In this case, s will not produce an optimistic reply for C, but only a conservative one.

□

6.5 Implementation

In this section, we describe how Eyrie was extended to implement Fast-SSMR, then we detail Chirper, our social network application built on Eyrie. Eyrie and Chirper were implemented in Java.

6.5.1 Extending Eyrie

To implement Fast-SSMR, Eyrie creates an optimistic copy of every `PRObject` instance. When a command is optimistically delivered, it is executed against the optimistic copy of the objects, i.e., against the optimistic state. Also, there are now two threads in Eyrie that call the `executeCommand` method of the `StateMachine` class: the conservative state machine thread, and the optimistic state machine thread. This is transparent to the service designer, who simply writes one implementation of the method executeCommand, accessing objects as if there was a single copy of each of them. Internally, when Eyrie intercepts the method call to `PRObject` instances, it determines which thread (conservative or optimistic) made the call and redirects the method invocation to the proper (conservative or optimistic) copy of the object. This was implemented with the Java reflection API. Class `OptimisticStateMachine` is responsible for performing and verifying the optimistic execution, and also repairing the optimistic state when necessary. To repair the optimistic state, only objects that might have been affected by the ordering mistake are copied from the conservative state.

6.5.2 Chirper

We implemented the Chirper service on top of Eyrie, providing an API similar to that of Twitter. Twitter is an online social networking service in which users can post 140-character messages and read posted messages of other users. The API consists basically of:

- `post(long uid, String msg)`: user with id `uid` publishes message `msg`.
- `follow(long uid, long fid)`: user with id `uid` starts following user with id `fid`.
- `unfollow(long uid, long fid)`: user with id `uid` stops following user with id `fid`.
- `getTimeline(long uid)`: user with id `uid` requests messages of all people the user follows.

Chirper makes use of Fast-SSMR, so each command can have two replies: an optimistic reply (fast, but possibly inconsistent) and a conservative reply (slower, but always correct). Any execution of Chirper is linearizable with respect to conservative replies. With respect to optimistic replies, the execution may violate linearizability if the optimistic delivery order of a message does not match its conservative delivery. This means that a client of the service may observe an inconsistent state until the conservative reply arrives and rectifies the state seen by the client. An online social networking service is well-suited for Fast-SSMR, since this kind of application can tolerate brief inconsistencies in exchange for faster replies.

State partitioning in Chirper is based on user id. Chirper's oracle uses a hash function $h(uid)$ that returns the partition that contains all up-to-date information regarding user with id uid. Taking into account that a typical user probably spends more time reading messages (i.e., issuing `getTimeline`) than writing them (i.e., issuing `post`), we decided to optimize `getTimeline` to be single-partition. This means that, when a user requests her timeline, all messages should be available in the partition that stores that user's data, in the form of a *materialized timeline* (similar to a materialized view in a database). To make this possible, whenever a `post` command is executed, the message is inserted into the materialized timeline of all users that follow the one that is posting. Also, when a user starts following another user, the messages of the followed user are inserted into the follower's materialized timeline as part of the command execution; likewise, they are removed when a user stops following someone. Because of this design decision, every `getTimeline` command accesses only one partition, `follow` and `unfollow` commands access at most two partitions, and `post` commands access up to all partitions.

One detail about the `post` command is that it must be multicast to all partitions that contain a follower of the user issuing the post. The Chirper client cannot know for sure who follows the user: it keeps a cache of followers, but such a cache can become stale if a different user starts following the poster. To ensure linearizability, when executing the `post` command, the Chirper server checks if the corresponding command was multicast to the proper set of partitions. If that was the case, the command is executed. Otherwise, the server sends a *retry*(γ) message to the client and proceeds to the next command. Upon receiving the *retry* message, the client multicasts the command again, now adding all partitions in γ (a set of partitions) as destinations for the next attempt. This repeats until all partitions that contain followers of the poster deliver the command. This is guaranteed to terminate because partitions are only added to the set of destinations for retries, never removed. Therefore, in the worst case scenario, the client will retry until it multicasts the post command to all partitions of the system. Note that the optimistic state machine may wrongly infer, based on an incorrect optimistic state, that the destination set of the command is incomplete. Because of that, only the conservative state machine tells the client to retry a command: if the command is not multicast to all accessed partitions, the optimistic state machine skips the command, but does not send a *retry* message.

We compared Chirper to Retwis,[1] which is another Twitter clone. Retwis relies on Redis,[2] a key-value store well-known for its performance. Redis offers asynchronous replication, which does not ensure linearizability. The Twitter API is implemented in the Retwis client as follows. The `post` commands update the list of all messages ever posted, the list of message ids of the user who is posting, and the materialized timeline (a list of message ids) of each follower of the poster. The `follow` and `unfollow` commands update the list of people followed by the user issuing the command, and the list of followers of another user. Finally, the `getTimeline` command accesses only the materialized timeline of the user who issued the command.

Retwis is similar to Chirper in that both applications precompute each user's timeline, executing `getTimeline` commands as fast as possible, at the expense of other commands. However, even with a single server, Retwis does not ensure the same level of consistency as Chirper: when executing a `post` command, getting the followers of a poster and adding the message to each of their materialized timelines are completely separate requests to Redis, and they can be interleaved with follow and unfollow commands. This means that a user may have, in her timeline, a message posted by someone who she does not follow anymore, or vice-versa. Each post command in Chirper is a single, atomic operation, and such interleaves are impossible to happen in the conservative state. Even if such an inconsistency happens in the optimistic state, it will be due to an ordering mistake, and the optimistic state will be repaired as soon as the mistake is detected. Moreover, the client will always receive consistent conservative replies from the Chirper servers.

6.6 Performance evaluation

In this section, we assess the performance of Fast-SSMR, in terms of throughput scalability and latency. For this purpose, we conducted experiments with Chirper. We evaluated Fast-SSMR's latency improvement over S-SMR and the corresponding rate of mistaken replies received by clients. We also compared the throughput scalability of both techniques to show that Fast-SSMR does not need to sacrifice throughput in order to reduce latency. For this reason, Chirper was deployed using both S-SMR and Fast-SSMR. We compare Chirper results to Retwis, a Twitter-clone application backed by a Redis key-value store.

[1]http://retwis.antirez.com
[2]http://redis.io

6.6.1 Environment setup and configuration parameters

We ran all our experiments on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve Switch 2910al-48G gigabit network switch, and the Dell nodes were connected to an HP ProCurve 2900-48G gigabit network switch. Those switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 6.3 with kernel 2.6.32 and had the Oracle Java SE Runtime Environment 7. Before each experiment, we synchronize the clocks of the nodes using NTP. This is done to obtain accurate values in the measurements of the latency breakdown involving events in different servers.

In all our experiments, clients submit commands asynchronously, that is, each client can keep submitting commands even if replies to previous commands have not been received yet, up to a certain number of outstanding commands. Trying to issue new commands when this limit is reached makes the client block until some reply is received. Replies are processed by callback handlers registered by clients when submitting commands asynchronously. In the case of Fast-SSMR, there were two callback handlers for each request: one for the conservative reply, and one for the optimistic reply. We allowed every client to have up to 25 outstanding commands at any time. By submitting commands asynchronously, the load on the service can be increased without instantiating new client processes.

We used two kinds of workloads: Timeline (all issued requests are `getTimeline`) and Mix (7.5% `post`, 3.75% `follow`, 3.75% `unfollow`, and 85% `getTimeline`). We compared Chirper deployed with Fast-SSMR and with S-SMR.

We deployed Retwis with a single Redis server, while Chirper was run with 1, 2, 4 and 8 partitions, with 3 replicas per partition. In our experiments with Chirper, we use Ridge for multicast in experiments with both S-SMR and Fast-SSMR. We use 3 acceptors per Ridge ensemble, with in-memory storage.

6.6.2 Latency improvement

In this section we evaluate the latency improvement brought by Fast-SSMR. We report latency for five algorithms: S-SMR, Conservative, Baseline Optimistic, Fully Optimistic (i.e., with optimistic state exchange enabled), and Retwis as reference. Retwis relied on a non-fault-tolerant stand-alone Redis server. Since we are interested in measuring latency in the absence of contention, we run these experiments with a low load.

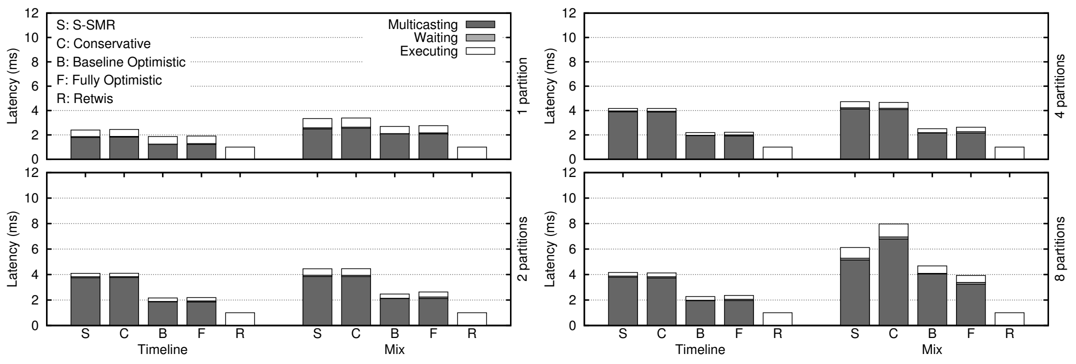

Figure 6.2. Latency components for different workloads, partitionings and levels of optimism.

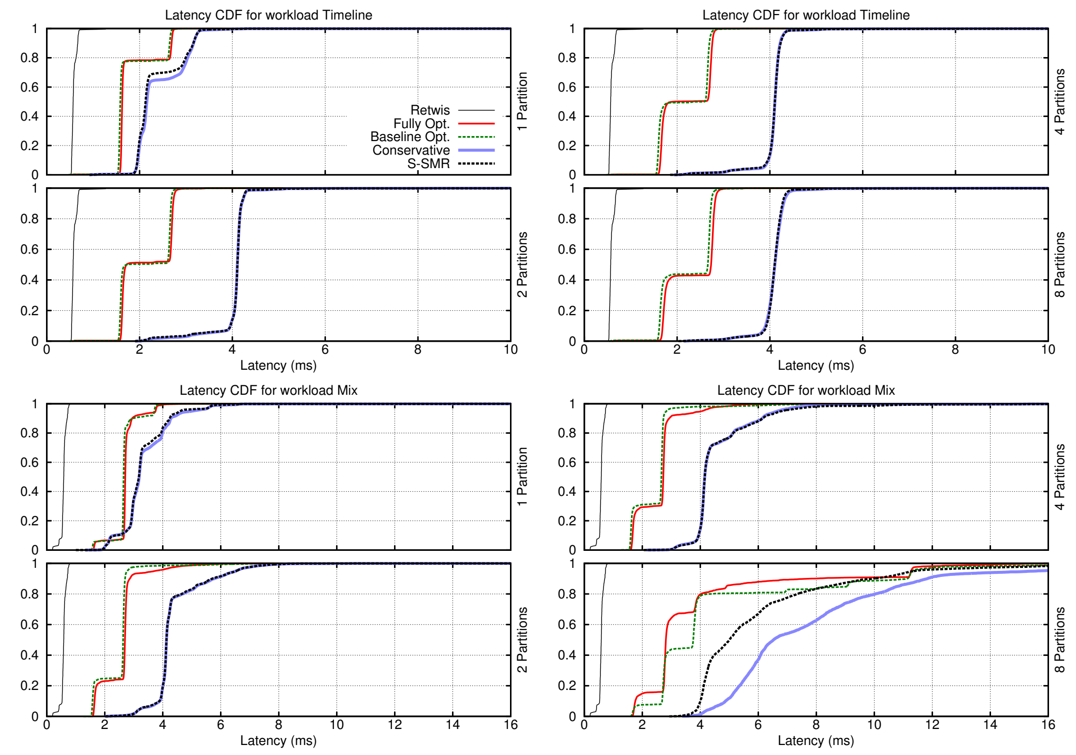

Figure 6.3. Cumulative distribution function (CDF) of latency for different workloads, partitionings and levels of optimism.

Figure 6.2 shows the latency components for each algorithm tested. For each command C, the *Multicasting* component is the time between the moment when the client multicasts C and the moment when the server s that replies to C cons-

or opt-delivers the command and puts it in an execution queue; *Waiting* is the time from when s delivers C until when s starts executing it; *Executing* is the time from when s starts to execute C to when the client receives the reply. We can see that the multicast time of the optimistic approaches (i.e., Baseline and Fully Optimistic) is significantly lower (in proportion) than that of S-SMR and the conservative one of Fast-SSMR, which happens thanks to the lower latency of the optimistic delivery. The Waiting component is nearly zero because the system is not saturated, so commands do not have to wait very long to be executed—when there are too many commands, they are put in a queue to be executed later. The execution time comprises the time it takes to execute the command itself, including any coordination across partitions. We can see that the Executing component is shorter for the Timeline workload, which happens because all requests are single-partition, requiring no cross-partition coordination. In the Mix workload, many requests are multi-partition, requiring exchange of both signals and state between servers.

The optimistic state exchange optimization (Fully Optimistic) shows some latency improvement over the Baseline approach, although this is visible only as the number of partitions increases (i.e., eight partitions, in Figure 6.2, bottom right). In Figure 6.3, we can see the cumulative distribution functions (CDFs) of latency for each workload and partitioning. The latency difference between the different techniques becomes more evident, as both optimistic latencies are significantly lower than those of S-SMR and Conservative. As for the optimistic state exchange, it is expected to reduce latency only for the Mix workload with more than one partition, and we see improvements with eight partitions. Nevertheless, the Fully Optimistic latency is roughly the same as the Baseline Optimistic latency in the worst case.

6.6.3 Throughput scalability

In this section, we evaluate the throughput scalability of Fast-SSMR, whose goal is to scale throughput while providing low-latency replies, and we compare it to the original S-SMR. For this purpose, we implemented Chirper with both replication techniques, using the baseline algorithm for Fast-SSMR.

Figure 6.4 (left) shows throughput and latency results for Chirper when deployed with S-SMR. Throughput values for each workload are normalized by the throughput of Chirper with a single partition, whose absolute value (in thousands of commands per second, or kcps) is shown in the graph. Figure 6.4 (right) presents the corresponding results for Chirper deployed with Fast-SSMR. Both optimistic and conservative latencies are shown: the solid bottom of each bar

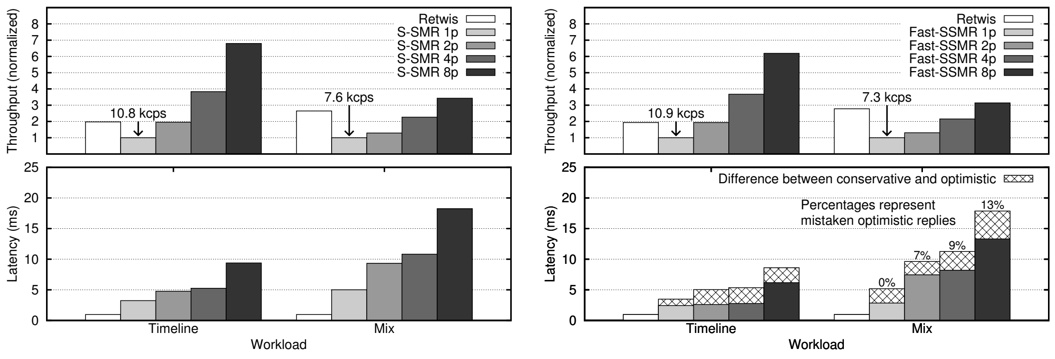

Figure 6.4. Scalability results for Chirper, implemented with S-SMR and Fast-SSMR, with 1, 2, 4 and 8 partitions. Retwis results are also shown. For each workload, the throughput was normalized by that of Chirper with a single partition (absolute values in thousands of commands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput.

represents the latency of the optimistic replies, while the hatched top represents the difference between the two latencies; the sum of the two is the conservative latency.

We can see that having two state machine threads (one conservative and one optimistic) running on each of our multi-core servers had very little impact on the maximum throughput of the system, in comparison to S-SMR. The slight decrease in throughput observed may be explained by the fact that, when the optimistic state machine performs a repair and copies the state from the conservative state machine, the latter is locked for a brief period to allow a consistent state to be copied from it. The figure also shows that the conservative latency of Fast-SSMR is roughly the same as S-SMR, which means that both approaches are able to provide consistent replies within the same time. Moreover, we can see that there is a significant latency reduction by having optimistic execution, at the cost of possibly having some mistaken optimistic replies. There are no mistaken replies for the Timeline workload because getTimeline requests are read-only. Thus, if there are no other requests, getTimeline requests can be executed in any order without affecting the final state of the service or the replies sent back to clients. For an incorrect reply to be received, the workload must contain both requests that read and requests that update the service state, as in the Mix workload. Finally, the figure shows that, with both replication techniques, Chirper scales with the number of partitions, reaching a higher throughput than Retwis already with two partitions for the Timeline workload and with eight partitions for the Mix workload. It is worth noting that Chirper ensures linearizability, unlike Retwis.

The optimistic latency of Chirper, for the Timeline workload, is significantly lower than the conservative one (between 30 and 50 percent). This happens because the Timeline workload is composed solely of single-partition requests, not requiring coordination between partitions. This is not the case for the Mix workload, which contains requests that require state exchange. When executing such a request, the (Baseline) optimistic state machine has to wait until a remote state arrives from a remote conservative state machine. This extra waiting reduces the difference between the optimistic and conservative latencies. Even then, the optimistic latency is still lower (between 20 and 45 percent) than the conservative one.

The rate of mistakes is calculated by dividing the number of mistakes (optimistic replies that differed from the conservative replies) by the total number of optimistic replies received. We can see that, in all experiments, the percentage of mistaken replies was fairly low. The percentage of mistakes can be controlled by setting more or less aggressive parameters for the optimistic atomic multicast algorithm, with a trade-off between latency improvement and number of correct replies. For instance, increasing the wait time of optimistic atomic multicast (explained in Chapter 5) would decrease the rate of mistakes at the cost of higher optimistic latency.

6.7 Related work

Fast-SSMR uses optimism to provide faster replies, but other works have used optimistic execution before. In [27, 54], optimism is implemented in the communication layer, which allows replicas to deliver messages early. In Eve [40], replicas execute commands following a partial order that should result in an identical final state for all replicas. After execution, the states are compared and, if they differ, commands are re-executed sequentially. The approach proposed in [52] uses speculative execution in the context of fully replicated state machines. It assumes that commands are independent (i.e., they do not access the same data item) to accelerate execution; when this assumption does not hold, the execution is rolled back and the command has to be multicast for execution again. An optimistic atomic broadcast protocol [58] is used in [37, 41] to certify update transactions. In [37], the system ensures snapshot isolation [51] in a partitioned database. Out-of-order optimistic deliveries may cause transactions to abort, but only if the out-of-order transactions conflict and one of them already started executing.

In Zyzzyva [44], each command is sent to a primary replica, which forwards it to a number of replicas. Those replicas immediately execute the command and send a reply to the client, along with an execution history. If the client receives enough replies with the same history, the order is guaranteed to be correct; otherwise, the client informs the replicas about the inconsistency, forcing them to execute a conservative total order protocol. MDCC [45] makes uses of Generalized Paxos [49] to reduce the number of round-trips between data centers when executing transactions. Generalized Paxos extends Fast Paxos [50], and both are optimistic consensus protocols that may achieve consensus in fewer communication rounds than Paxos. MDCC performs well with commutative operations and provides read-committed isolation [3]. The authors argue that the protocol could be extended to provide stronger consistency levels, up to serializability.

There are three main differences between these previous works that use speculative execution and what we propose here. First, Fast-SSMR accommodates any kind of (deterministic) multi-partition commands, which requires a fairly sophisticated repair procedure in case of mistakes of the optimistic delivery. Second, in our dual state machine approach, the replica never stops executing commands or sending replies to clients; only the optimistic state machine may pause and not reply to some commands in case of an ordering mistake. Third, no rollback is necessary when repairing the optimistic state: it is simply copied from the conservative one; most complexity lies in determining which commands the optimistic state machine will execute after the repair is done.

6.8 Conclusion

This chapter introduced Fast Scalable State Machine Replication (Fast-SSMR), a technique that adds optimistic execution to S-SMR. Fast-SSMR differs from previous works because (i) it does not require the application to be able to rollback executions, (ii) it allows any kind of workload (composed of deterministic commands) and any kind of partitioning of the state, and (iii) when a mistake is detected, the conservative state machine keeps handling commands and sending replies. The cost of this technique is to have two state machines running in parallel at each replica: a conservative one and an optimistic one. Our experiments show that the optimistic replies have significantly lower latency than the conservative replies, while throughput and latency of the conservative replies were roughly the same as S-SMR.

Chapter 7

Conclusion

At the end, if you fail, at least you did something interesting, rather than doing something boring and also failing.

Barbara Liskov

Online services have recently reached scales never seen before. For instance, Facebook has over a billion users [1], while Google has reported to have nine hundred million active users for its Gmail service alone [2]. There is a clear demand for systems that are designed to scale. A system that ensures strong consistency while being able to scale would be ideal. Such a system would make designing services and clients simpler.

Previous techniques provide interesting solutions that allow systems to scale largely, but they do not ensure linearizability. Granted, in many cases lower levels of consistency can be tolerated, but there are services that require the strongest level of consistency. Even if they do not, designing applications to run on a linearizable system is significantly simpler than dealing with locking, aborting, retrying and so on, which is done to cope with less consistent underlying replication techniques.

In this work, we combine scalability with linearizability by introducing state partitioning to state machine replication. This way, servers of different partitions can execute independently. State machine replication relies on atomic broadcast, which does not scale. For this reason, we chose to build our scalable state machine replication approach on top of atomic multicast instead. Although scalable, atomic multicast has higher latency than atomic broadcast [63]. To improve latency without sacrificing throughput, we developed a novel atomic multicast protocol. To further reduce response time, we provide additional, optimistic replies to client requests.

7.1 Research assessment

This thesis presents four contributions: (i) Scalable State Machine Replication (S-SMR), (ii) Optimistic Atomic Multicast (Opt-amcast), (iii) Ridge, an optimistic atomic multicast protocol, and (iv) Fast Scalable State Machine Replication (Fast-SSMR). These contributions can be divided in two categories: (i) and (iv) are replication techniques, while (ii) and (iii) deal with message ordering.

Scalable State Machine Replication. S-SMR [12] is a replication technique built around the idea of partitioning state, while ensuring that every execution, regardless of the number of partitions accessed by each command, is linearizable. The goal of S-SMR is to increase throughput linearly with the number of partitions, provided that commands are independent (and are equally distributed among partitions). In order to achieve this, S-SMR uses atomic multicast to guarantee consistent command ordering, while additional synchronization is done to prevent non-linearizable executions.[1] For commands that access multiple partitions, servers may have to exchange data items. To minimize the complexity of designing such a partitioned service, Eyrie was implemented allowing developers to abstract away state partitioning and synchronization among servers of different partitions. Our experiments show that, for workloads composed mostly of independent commands, throughput does scale with the number of partitions.

Optimistic Atomic Multicast. Opt-amcast [11] is a class of atomic multicast protocols that deliver messages with probabilistic atomic order, which is faster to ensure than proper atomic order. For every message multicast, there are two deliveries: a conservative one, with the same properties of atomic multicast, and an optimistic one, which is faster, but may violate order. If the optimistic assumptions (which depend on the specific implementation of Opt-amcast) hold, the optimistic delivery order is the same as the conservative order; if not, they may contradict each other, in which case we say that the optimistic delivery made a "mistake". Such a class of protocols allows some computation to be done based on the optimistic delivery, which can lower response time.

Ridge. To improve the latency of atomic multicast, while preserving high throughput and also implementing optimistic atomic multicast, we developed Ridge [13]. It is based on the observation that a ring overlay is not necessary

[1] In P-SMR [56], we also used atomic multicast, but to allow multi-threaded execution of commands in state machine replication, with no state partitioning.

to achieve optimal theoretical throughput in a network. Instead, one can use alternating distributors to maximize throughput. Ridge proposes to optimize Phase 2 of Paxos, by disseminating proposed values with alternating distributors. To implement atomic multicast on top of Paxos, Ridge uses a deterministic merge procedure [4], similarly to what is done with Multi-Ring Paxos [55]. Optimistic deliveries are done with a scheme similar to the one we proposed in [11]. Both with a theoretical analysis and an experimental evaluation, we demonstrated that Ridge has high throughput (as high as that of previous techniques optimized for throughput), and low latency.

Fast Scalable State Machine Replication. To make use of the optimistic deliveries of optimistic atomic multicast, we devised Fast Scalable State Machine Replication, or Fast-SSMR [14]. It allows potentially inconsistent, albeit faster replies to be seen by clients. Such fast replies are given in addition to the final replies, which are guaranteed to be always correct. Fast-SSMR consists of running two state machines at each replica: an optimistic one and a conservative one. The technique relies on Opt-amcast: when a command is optimistically delivered, it is immediately executed by the optimistic state machine, which sends an optimistic reply back to the client. The conservative state machine executes the same command only when it is conservatively delivered, likely to happen later. If the conservative and optimistic delivery orders match, the optimistic reply seen by the client is correct; otherwise, the client will be aware of the mistake when it notices the difference between the two replies, while the optimistic state machine at the server repairs its state based on the conservative state machine. We extended Eyrie to implement Fast-SSMR, and implemented a social network application on top of it. The results we found demonstrate that the latency of the conservative replies of Fast-SSMR are roughly the same as those of S-SMR, and that the optimistic replies are significantly faster than the conservative ones.

7.2 Future directions

The aim of this thesis was to explore the space of techniques for scalable, strongly consistent replication and communication primitives. However, many questions remain open, so we point here at possible research directions.

Recovery in S-SMR. For this work, we have assumed a crash-stop model, that is, we assumed that once a process fails, it does not come back online. Although this assumption makes replicating services simpler, ideally a service should be able to accommodate the recovery of a crashed server. To efficiently implement recovery in the context of S-SMR, one should deal with *checkpointing*, that is, saving to stable storage the state of a server at a given point in time. How to implement a checkpoint in the context of partitioned-state state machine replication is a question that requires further investigation.

Trimming in atomic multicast. By restoring a checkpoint, a server would not need to atomically deliver again commands that had been delivered before the checkpoint was taken. Those commands could then be safely discarded—i.e., *trimmed*—from any logs kept in the ordering layer (e.g., the Paxos acceptors for Paxos-based atomic multicast protocols). For broadcast protocols, where there is a single totally ordered sequence of delivered messages, trimming is relatively simple: given a certain message (e.g., a command that started a checkpoint in SMR), all messages that precede it can be trimmed. However, such a total order does not necessarily exist with atomic multicast, and deciding which messages to discard is not as simple. This has been discussed in [9], but trimming in genuine atomic multicast [32] remains an open question.

Dynamic partitioning. This thesis assumes that each variable that composes the service state is statically assigned to a partition. With Chirper, the partitions accessed by a `post` command issued by a user depend on who follows that user when the command is executed, and this can change dynamically. However, even in the case of Chirper, the partition that contained each variable was still statically defined. Letting variables move around partitions allows interesting features to be implemented, such as load balancing and optimizing data locality for users as access patterns change. For instance, if two users start interacting a lot, it would make sense to move their data to the same partition to reduce response time.

Optimal-throughput, optimal-latency, genuine atomic multicast. An atomic multicast protocol that had optimal latency, while providing optimal throughput, would be the ideal primitive for S-SMR to build on, assuming such a protocol exists. Moreover, when using genuine atomic multicast, processes would only communicate if they actually had messages to exchange, so such a protocol would scale as much as the messages were independent.

Bibliography

[1] Facebook reports first quarter 2015 results, 2015. URL `http://investor.fb.com/releasedetail.cfm?ReleaseID=908022`.

[2] Google I/O by the numbers: 1B Android users, 900M on Gmail, 2015. URL `http://cnet.co/1FFJREk`.

[3] A. Adya, B. Liskov, and P. O'Neil. Generalized isolation level definitions. In *Proceedings of the 16th International Conference on Data Engineering*, ICDE '00, pages 67–78. IEEE Computer Society, 2000.

[4] M. Aguilera and R. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '00, pages 209–218. ACM, 2000.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Digest of Papers., Twenty-Second International Symposium on Fault-Tolerant Computing*, FTCS-22, pages 76–84. IEEE Computer Society, 1992.

[6] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring membership protocol. *ACM Transactions On Computer Systems*, 13(4):311–342, 1995.

[7] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton. The Spread toolkit: Architecture and performance. Technical Report CNDS-2004-1, Center for Networking and Distributed Systems, 2004.

[8] H. Attiya and J. Welch. *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004.

[9] S. Benz, P. Marandi, F. Pedone, and B. Garbinato. Building global and scalable systems with atomic multicast. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 169–180. ACM, 2014.

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

[11] C. E. Bezerra, F. Pedone, B. Garbinato, and C. Geyer. Optimistic atomic multicast. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 380–389. IEEE Computer Society, 2013.

[12] C. E. Bezerra, F. Pedone, and R. van Renesse. Scalable state-machine replication. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 331–342. IEEE Computer Society, 2014.

[13] C. E. Bezerra, D. Cason, and F. Pedone. Ridge: high-throughput, low-latency atomic multicast. In *Proceedings of the 2015 IEEE 34th Symposium on Reliable Distributed Systems*, SRDS '15, pages 256–265. IEEE Computer Society, 2015.

[14] C. E. Bezerra, F. Pedone, R. van Renesse, and C. Geyer. Providing scalability and low latency in state machine replication. Technical Report USI-INF-TR-2015/06, Università della Svizzera italiana, 2015.

[15] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. *Computer Networks*, 33(1-6):1–16, 2000.

[16] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-Paxos: Offloading the leader for high throughput state machine replication. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 111–120. IEEE Computer Society, 2012.

[17] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[18] A. Bouch, A. Kuchinsky, and N. Bhatti. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, CHI '00, pages 297–304. ACM, 2000.

[19] N. Bronson et al. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC '13, pages 49–60. USENIX Association, 2013.

[20] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In *Distributed Systems (2nd Ed.)*, pages 199–216. ACM, 1993.

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

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

[23] P. Chundi, D. Rosenkrantz, and S. Ravi. Deferred updates and data placement in distributed databases. In *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, pages 469–476. IEEE Computer Society, 1996.

[24] J. Corbett et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, 2013.

[25] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

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

[27] P. Felber and A. Schiper. Optimistic active replication. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 333–341. IEEE Computer Society, 2001.

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

[29] U. Fritzke and P. Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, ICDCS '01, pages 284–291. IEEE Computer Society, 2001.

[30] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, 1991.

[31] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 15–28. ACM, 2011.

[32] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, 2001.

[33] R. Guerraoui, D. Kostic, R. Levy, and V. Quéma. A high throughput atomic storage algorithm. In *Proceedings of the 27th International Conference on Distributed Computing Systems*, ICDCS '07, page 19. IEEE Computer Society, 2007.

[34] R. Guerraoui, R. Levy, B. Pochon, and V. Quéma. Throughput optimal total order broadcast for cluster environments. *ACM Transactions on Computer Systems*, 28(2):5:1–5:32, 2010.

[35] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[36] P. Hunt, M. Konar, F. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIX ATC '10, pages 145–158. USENIX Association, 2010.

[37] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS '02, pages 477–484. IEEE Computer Society, 2002.

[38] F. Junqueira, B. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, DSN '11, pages 245–256. IEEE Computer Society, 2011.

[39] M. Kapritsos and F. Junqueira. Scalable agreement: Toward ordering as a service. In *Proceedings of the Sixth Worshop on Hot Topics in System Dependability*, HotDep '10, pages 1–8. USENIX Association, 2010.

[40] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '12, pages 237–250. USENIX Association, 2012.

[41] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 19th*

IEEE International Conference on Distributed Computing Systems, ICDCS '99, pages 424–431. IEEE Computer Society, 1999.

[42] T. Kobus, M. Kokocinski, and P. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 286–296. IEEE Computer Society, 2013.

[43] R. Kotla and M. Dahlin. High throughput Byzantine fault tolerance. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 575–584. IEEE Computer Society, 2004.

[44] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 45–58. ACM, 2007.

[45] T. Kraska, G. Pang, M. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 113–126. ACM, 2013.

[46] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[47] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2): 254–280, 1984.

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

[49] L. Lamport. Generalized consensus and paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[50] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[51] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Transactions on Database Systems*, 34(2):11:1–11:49, 2009.

[52] P. Marandi and F. Pedone. Optimistic parallel state-machine replication. In *Proceedings of the 2014 IEEE 33rd International Symposium on Reliable Distributed Systems*, SRDS '14, pages 232–241. IEEE Computer Society, 2014.

[53] P. Marandi, M. Primi, N. Schiper, and F. Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '10, pages 527–536. IEEE Computer Society, 2010.

[54] P. Marandi, M. Primi, and F. Pedone. High performance state-machine replication. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, DSN '11, pages 454–465. IEEE Computer Society, 2011.

[55] P. Marandi, M. Primi, and F. Pedone. Multi-Ring Paxos. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pages 1–12. IEEE Computer Society, 2012.

[56] P. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state-machine replication for parallelism. In *Proceedings of the 2014 IEEE 34th International Conference on Distributed Computing Systems*, ICDCS '14, pages 368–377. IEEE Computer Society, 2014.

[57] L. Pacheco, D. Sciascia, and F. Pedone. Parallel deferred update replication. In *Proceedings of the 2014 IEEE 13th International Symposium on Network Computing and Applications*, NCA '14, pages 205–212. IEEE Computer Society, 2014.

[58] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, pages 318–332. Springer-Verlag, 1998.

[59] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[60] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proceedings of the 7th International Conference on Computer Communications and Networks*, ICCCN '98, pages 840–847. IEEE Computer Society, 1998.

[61] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, pages 266–275. IEEE Computer Society, 2013.

[62] N. Schiper. *On Multicast Primitives in Large Networks and Partial Replication Protocols*. PhD thesis, Università della Svizzera italiana, 2009.

[63] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *Proceedings of the 9th International Conference on Distributed Computing and Networking*, ICDCN '08, pages 147–157. Springer-Verlag, 2008.

[64] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[65] D. Sciascia and F. Pedone. RAM-DUR: In-memory deferred update replication. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*, SRDS '12, pages 81–90. IEEE Computer Society, 2012.

[66] D. Sciascia and F. Pedone. Geo-replicated storage with scalable deferred update replication. In *Proceedings of the 2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '13, pages 1–12. IEEE Computer Society, 2013.

[67] D. Sciascia, F. Pedone, and F. Junqueira. Scalable deferred update replication. In *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, pages 1–12. IEEE Computer Society, 2012.

[68] J. Smith and G. Maguire Jr. Effects of copy-on-write memory management on the response time of unix fork operations. *Computing Systems*, 1(3): 255–278, 1988.

[69] A. Sousa, R. Oliveira, F. Moura, and F. Pedone. Partial replication in the database state machine. In *Proceedings of the IEEE International Symposium on Network Computing and Applications*, NCA '01, pages 298–309. IEEE Computer Society, 2001.

[70] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, SRDS '02, pages 190–199. IEEE Computer Society, 2002.

[71] J. Sousa and A. Bessani. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation. In *Proceedings of the 2012 Ninth European Dependable Computing Conference*, EDCC '12, pages 37–48. IEEE Computer Society, 2012.

[72] R. van Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In *Proceedings of the 6th Symposyum on Operating Systems Design & Implementation*, OSDI '04, pages 91–104. USENIX Association, 2004.

[73] R. van Renesse, C. Ho, and N. Schiper. Byzantine chain replication. In *Proceedings of the 16th International Conference on Principles of Distributed Systems*, OPODIS '12, pages 345–359. Springer-Verlag, 2012.

[74] X. Wang, A Balasubramanian, A. Krishnamurthy, and D. Wetherall. Demystifying page load performance with WProf. In *Proceedings of the 10th USENIX Symposyum on Networked Systems Design and Implementation*, NSDI '13, pages 473–485. USENIX Association, 2013.

[75] J. Yin, J. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, SOSP '03, pages 253–267. ACM, 2003.