High-Performance 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

Parisa Jalili Marandi

under the supervision of

Fernando Pedone

Parisa Jalili Marandi
Lugano,

To my parents

Only when we know a little do we
know anything; doubt grows with
knowledge.

Johann Wolfgang von Goethe

Publications

- R.Soule, S.Basu, P.J.Marandi, F.Pedone, R.Kleinberg, EG.Sirer, N.Foster. "Merlin: A Language for Provisioning Network Resources", (CoNEXT 2014).

- S.Benz, P.J.Marandi, F.Pedone, B.Garbinato. "Building global and scalable systems with Atomic Multicast", (Middleware 2014).

- P.J.Marandi, F.Pedone. "Optimistic Parallel State-Machine Replication", (SRDS 2014).

- P.J.Marandi, S.Benz, F.Pedone, K.P.Birman. "The Performance of Paxos in the Cloud", (SRDS 2014).

- P.J.Marandi, E.Bezzera, F.Pedone. "Rethinking State-Machine Replication for Parallelism", (ICDCS 2014).

- P.J.Marandi, M.Primi, F.Pedone. "Multi-Ring Paxos", (DSN 2012).

- P.J.Marandi, M.Primi, F.Pedone. "High Performance State-Machine Replication", (DSN 2011).

- P.J.Marandi, M.Primi, N.Schiper, F.Pedone. "Ring Paxos: A High-Throughput Atomic Broadcast Protocol", (DSN 2010).

Abstract

Replication, a common approach to protecting applications against failures, refers to maintaining several copies of a service on independent machines (replicas). Unlike a stand-alone service, a replicated service remains available to its clients despite the failure of some of its copies. Consistency among replicas is an immediate concern raised by replication. In effect, an important factor for providing the illusion of an uninterrupted service to clients is to preserve consistency among the multiple copies. State-machine replication is a popular replication technique that ensures consistency by ordering client requests and making all the replicas execute them deterministically and sequentially. The overhead of ordering the requests, and the sequentiality of request execution, the two essential requirements in realizing state-machine replication, are also the two major obstacles that prevent the performance of state-machine replication from scaling.

In this thesis we concentrate on the performance of state-machine replication and enhance it by overcoming the two aforementioned bottlenecks, the overhead of ordering and the overhead of sequentially executing commands. To realize a truly scalable system, one must iteratively examine and analyze all the layers and components of a system and avoid or eliminate potential performance obstructions and congestion points. In this dissertation, we iterate between optimizing the ordering of requests and the strategies of replicas at request execution, in order to stretch the performance boundaries of state-machine replication.

To eliminate the negative implications of the ordering layer on performance, we devise and implement several novel and highly efficient ordering protocols. Our proposals are based on practical observations we make after closely assessing and identifying the shortcomings of existing approaches. Communication is one of the most important components of any distributed system and thus selecting efficient communication patterns is a must in designing scalable systems.

We base our protocols on the most suitable communication patterns and extend their design with additional features that altogether realize our protocol's high efficiency. The outcome of this phase is the design and implementation of the Ring Paxos family of protocols. According to our evaluations these protocols are highly scalable and efficient. We then assess the performance ramifications of sequential execution of requests on the replicas of state-machine replication. We use some known techniques such as state-partitioning and speculative execution, and thoroughly examine their advantages when combined with our ordering protocols. We then exploit the features of multicore hardware and propose our final solution as a parallelized form of state-machine replication, built on top of Ring Paxos protocols, that is capable of accomplishing significantly high performance.

Given the popularity of state-machine replication in designing fault-tolerant systems, we hope this thesis provides useful and practical guidelines for the enhancement of the existing and the design of future fault-tolerant systems that share similar performance goals.

Acknowledgements

I wish to express my genuine gratitude to my advisor prof. Fernando Pedone. I owe the creation and completion of this thesis to his inexhaustible patience, persistent passion for educating, his continuous stream of fresh ideas, and his dedication to correcting the mistakes of his students. To him research is a glorious adventure, and his commitment to advising comes effortlessly; having witnessed his spirit has always been and will be a great motivation to me. Whenever I needed help it was impossible not to reach Fernando, as his communication links are the most reliable and fault-tolerant ones. I am extremely fortunate to have met you and to have learnt some of the most valuable lessons of my life while working with you. I could not make any phrase that justly describes the value of what you have directly and indirectly taught to me, "Thank you".

I would like to thank the rest of my thesis committee: Prof. Andre Schiper, Prof. Kenneth P. Birman, Prof. Antonio Carzaniga, Prof. Matthias Hauswirth. Having met these great people has been a unique pleasure and has contributed to the quality of this work. I would like to also thank Prof. Mehdi Jazayeri for his friendly guidance and support. I would like to thank all the faculty and staff of the University of Lugano for their commitment to enhancing the quality of education at the University. I am grateful to the Swiss National Science Foundation for supporting the projects I was assigned to. I am also grateful to Prof. Kenneth P. Birman and Prof. Robert Van Renesse for their hospitality at the Cornell University and the Zeno Karl Schindler Foundation for supporting my stay there.

I am in particular indebted to the members of the distributed systems lab at the University of Lugano: Marco Primi, Nicolas Schiper, Daniele Sciascia, Amirhossein Malekpour, Ricardo Padilha, Eduardo Bezzera, Leandro Pacheco, Samuel Benz, and Alex Tomic, without whose help and contributions this work would not be accomplished.

I would like to in particular thank Shima and Parvaz for their valuable friendship over the last few years. Many other dear friends have accompanied me during this journey to whom my gratitude is endless: Pascale, little Helena, Marjan, little Melina, Luciana, little Giovanni, Katherine, Stefania, Adina, Marjaneh, Sepideh, Sara, Elena, Elisa, Nina, Daniella, Noushin, Nayereh, Paniz, Amirhossein, Mehran, Carlo, Nicoals, Hamid, Sandeep, koorosh, Marco, Andrea, Amanj, and many many others. Thanks to the Luganese my stay at Lugano has been nothing short of amazing. In rainy days and sunny days, I have been only confronted with warm smiling faces.

I would like to dedicate this work to Fatemeh and Javad my dearest parents, for their everlasting and unconditional support, love, and companionship. Your courage, resilience, and inspiring life has always been and will be the light of my life at the dark days. I am not sure how I would have coped with the distance if it was not for your daily calls, kindness, support, and listening ears. I am fortunate to have been born to you and words fall short at expressing my gratitude, love, and respect to you, “Thank you”. I also would like to dedicate this work to the memories of my grandparents and in particular my grandfather who wished me to be a medical doctor. I am grateful to my dear sister and her family: Samira, Javad, and little Martia for their greatest friendship. I will remain indebted to you for having always welcomed me in your small but the warmest gatherings. Thank you for creating some of the most happiest moments in my life.

Contents

Figures

Tables

Chapter 1

Introduction

With the wide availability of commodity hardware, failures in data centers are common and inevitable. Machines fail individually due to hardware issues and software bugs, or collectively due to problems in interconnects and power distribution. Storage devices are equally subject to failures, and network performance is often degraded due to malfunctioning switches and damaged links. These types of failures are frequently observed in many data centers, including those from well-established service providers [1]. Given the prevalence of failures and in order to offer uninterrupted service to their clients, services deployed in data centers must look for solutions to reduce their vulnerability to failures.

Replication is a well-known approach to making computer systems fault-tolerant. To enhance fault-tolerance via replication, multiple copies of a service are created and operated on failure-independent machines. Therefore, a replicated service is always available to its clients despite the failure of some of its replicas. Since replicas operate independently, their state might diverge due to concurrent and isolated execution of requests. Thus, consistency among replicas is an immediate concern raised by replication. From the consistency perspective, replication strategies can be categorized into two classes: eager (synchronous) and lazy (asynchronous) [2]. In order to ensure that a service's state is always identical among replicas, in eager replication all the requests are synchronized among replicas before execution. In lazy replication, however, synchronization among the replicas happens in the background, possibly after executing the requests and responding to the clients. Therefore, in systems that implement lazy replication, replicas have weaker consistency guarantees. Inconsistency among replicas is undesirable if the clients can not tolerate contradictory responses; it is acceptable if the clients can bear inconsistencies in the interest of rapid responses.

A popular eager replication technique that enforces strong consistency among replicas is *state-machine replication* [3]. State-machine replication is based on the concept of deterministic state machines. A deterministic state machine models a system as a set of states among which transitions happen deterministically by the execution of well-defined operations. State-machine replication builds on this concept where all the replicas of a service independently implement a deterministic state machine: they all start at the same initial state and *sequentially* apply an *ordered* set of operations to their states, one operation at a time. Therefore, state machines on different replicas traverse the same series of states, produce the same sequence of outputs, and look consistent and identical at any given time. Consistency among the replicas of state-machine replication is the key to masking failures: the client of a failed replica can smoothly be directed to another replica without observing meaningless or inconsistent responses. In other words, consistency makes replication transparent and conveys to clients the illusion of a coherent and centralized service.

Ordering the operations and sequentially executing the ordered operations are two crucial properties of state-machine replication for realizing strongly consistent systems that are immune to failures. Throughout this dissertation, we argue that these same wanted requirements are also the reasons that restrain the performance of state-machine replication from scaling. First, to order the requests, an agreement layer is often positioned between clients and servers, to which all the requests are directed before being executed on the replicas (see Figure 1.1). This new layer of communication and computation adds extra overhead to the system's performance and increases the response time experienced by the clients. Therefore, as a result of ordering, a replicated service often loses to a single-copy service in terms of performance. Second, with the addition of replicas, as physically independent machines, there are extra resources in the system (e.g. CPU, memory, IO), but because of the sequential execution of all the requests on all the replicas, the added resources can not be exploited to the advantage of performance.

Our main goal in this dissertation is to focus on the ordering requirement and the sequentiality of execution in state-machine replication and devise efficient solutions toward the realization of high-performance available systems that are implemented by state-machine replication. The main research questions (RQ) of this dissertation are as follows:

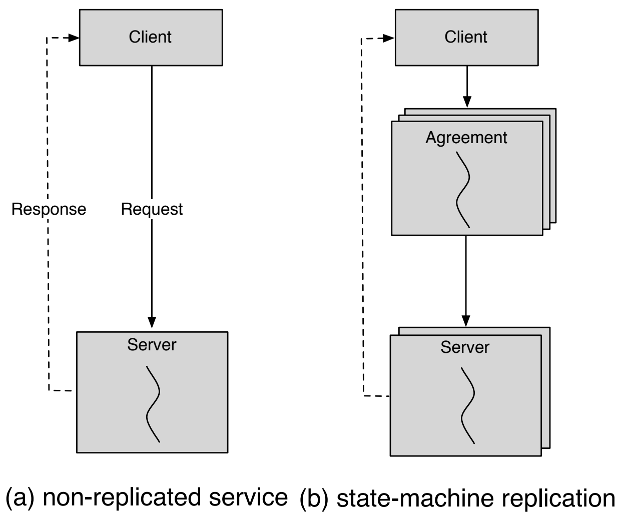


Figure 1.1. Non-replicated service and state-machine replication

- **RQ1.** How to efficiently design an agreement layer that does not restrict the performance of a replicated service?
- **RQ2.** How to overcome the sequentiality of execution in state-machine replication to enhance performance while preserving consistency?

In the next section, we briefly overview our main contributions in addressing these questions.

1.1 Research Contributions

In this section, we outline the main contributions of this dissertation and provide a short description of each one. We defer detailed discussions to the next chapters.

The design of Ring Paxos protocols *(RQ1)*: Ring Paxos is derived from Paxos [4] and is composed of two high-throughput atomic broadcast protocols. The common feature among both versions of Ring Paxos is the presence of a ring topology at the heart of the protocols. Besides relying on a logical ring overlay, Ring Paxos is based on a series of practical observations that altogether realize its high performance. Ring Paxos minimizes the overhead of ordering so that the agreement layer adds little overhead. Two variants of Ring Paxos were developed:

- **Multicast-based Ring Paxos (M-Ring Paxos):** This variant of Ring Paxos leverages network level ip-multicast to attain wire-speed throughput. Ip-multicast is an efficient primitive for disseminating messages to a set of receivers. This property, combined with a set of other practical observations, helps M-Ring Paxos achieve high throughput with reasonable latency.

- **Unicast-based Ring Paxos (U-Ring Paxos):** Ip-multicast may not be available in some environments. Therefore, we have developed U-Ring Paxos, a variant of Ring Paxos in which all communication is based on unicast. Removing ip-multicast, as one of the main communication mechanisms, raised new challenges that resulted in several modifications to the design of the protocol. Although on average it has higher latency, U-Ring Paxos's throughput is comparable to M-Ring Paxos's.

Speculative delivery with M-Ring Paxos *(RQ1)*: Ordering in Ring Paxos is efficient and has a reasonably low latency. Yet to investigate the possibility of reducing latency further, we have implemented speculative delivery with M-Ring Paxos. We have modified M-Ring Paxos such that requests are delivered to replicas before their order is determined. Speculative delivery assumes that the order in which requests are received at replicas will comply with the order decided by M-Ring Paxos. Based on the speculative assumption, replicas execute requests parallel to M-Ring Paxos ordering them. According to our findings, speculative delivery proves effective and is able to reduce latency.

State partitioning with M-Ring Paxos *(RQ2)*: In the presence of Ring Paxos as a high-throughput atomic broadcast protocol, the performance bottleneck of a replicated service may shift from the agreement layer to the service side: performance is limited by the number of requests that replicas can execute rather than the number of requests the agreement layer can order. Partitioning the state of a service is a well-known approach to enhancing the execution of requests on replicas. In a fully replicated service, all the replicas deliver all the requests and execute all of them. Thus one can not expect to gain performance as a result of adding new replicas to the system. To study the effect of partitioning, we modified M-Ring Paxos to combine it with a partitioned service. In the resulting system, M-Ring Paxos totally orders all the requests but uses a set of multicast groups rather than one, to transfer requests to relevant partitions. Our evaluations show that partitioning the state and using the modified M-Ring Paxos in the agreement layer results in significant improvements in performance.

Multi-Ring Paxos *(RQ1)*: Partitioning a service elevates the throughput of request execution on replicas as far as the agreement layer allows. In other words, assuming a service that can be partitioned into a large number of substates in which request execution is extremely efficient, the performance bottleneck once more slides to the agreement layer. We observed this phenomenon with Ring Paxos: state partitioning continues to increase the performance until Ring Paxos reaches its maximum capacity in ordering the requests. When this happens, adding more processes to the agreement layer increases its availability and fault-tolerance without having positive impact on the performance. We designed Multi-Ring Paxos as an ensemble of isolated but coordinated Ring Paxos instances to address the scalability issue of Ring Paxos. Multi-Ring Paxos implements atomic multicast by relying on a set of pre-defined parameters to coordinate independent instances of Ring Paxos. Performance of Multi-Ring Paxos scales linearly as new resources and machines are added to the system.

Parallel state-machine replication (P-SMR) *(RQ2)*: Sequential execution of requests on replicas, with both partial and full replication, is a source of performance bottleneck in state-machine replication. The development of multi-threaded services on the one hand and the sequentiality of replicas at processing requests on the other hand make the union of parallel applications with state-machine replication harder. We have built parallel state-machine replication (P-SMR) on top of Multi-Ring Paxos as an attempt to adapt state-machine replication to multicore environments, increase its performance, and bridge the gap between state-machine replication and multi-threaded services. Compared to other parallelized replication strategies, P-SMR achieves a significantly higher performance.

Evaluating Paxos in the Cloud: The last contribution of this dissertation is devoted to comparing the performance of Ring Paxos to other Paxos implementations when deployed in the cloud. We evaluated four open source Paxos libraries on Amazon EC2 under various configurations to closely analyze the performance and behavior of these libraries with and without failures. Our experiments reveal the differences among the policies open source implementations use to cope with failures. We propose some improvements and demonstrate their effectiveness with experiments.

1.2 Structure of the Dissertation

The rest of this dissertation is structured as follows. In Chapter 2, we present our system model and definitions that are frequently used in the rest of the text. In Chapter 3 we present the Ring Paxos protocols and discuss their design and properties in detail. In Chapter 4 we study speculative delivery and state partitioning and discuss their performance advantages when combined with Ring Paxos. In Chapter 5 we discuss scalability issues of Ring Paxos and introduce Multi-Ring Paxos to address them. In Chapter 6 we question the sequentiality of execution in state-machine replication and propose P-SMR, a new parallel execution model, to parallelize execution. In Chapter 7 we evaluate several open source implementations of Paxos in Amazon EC2 and compare their policies in handling failures. Finally, in Chapter 8, we conclude the dissertation by outlining our main findings and presenting directions for future research.

Chapter 2

System Model and Definitions

In this chapter, we present our assumptions about the system model and review the definitions of several concepts that are repeatedly referred to in the rest of this text.

2.1 System model

Unless mentioned otherwise, we make the following assumptions about processes, failure and synchrony models.

- **Processes and communication.** We assume a distributed system composed of a set $\Pi = \{p_1, p_2, ...\}$ of interconnected processes. If required we differentiate between processes as $C = \{c_1, c_2, ...\}$, an unbounded set of client processes, and $S = \{s_1, s_2, ..., s_n\}$, a bounded set of server processes (replicas). We also assume the nodes (machines) on which processes are located are in a local-area network. The network is mostly reliable and subject to small latencies. Communication among processes happens through message passing. Communication can be one-to-one through the primitives *send*(p, m) and *receive*(m), and one-to-many through the primitives *ip-multicast*(g, m) and *ip-deliver*(m), where m is a message, p is a process, and g is a group of processes to which m is addressed. Messages can be lost but not corrupted. Therefore, if a process p sends a message m to another process q, and q is non-faulty, q eventually receives m as long as p retries sending m. Moreover, nodes have access to stable storage if required.

- **Failure model.** We assume the crash recovery model, in which nodes can fail by crashing and recover later, but no byzantine or malicious behavior is tolerated.

- **Synchrony model.** Our protocols ensure safety under both asynchronous and synchronous execution periods. The FLP impossibility result [5] states that under asynchronous assumptions consensus cannot be both safe and live may a process crash. For our consensus algorithms, we thus assume that the system is partially synchronous, which means that it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the Global Stabilization Time (GST) [6] and 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. Moreover, in order to prove liveness, we assume that after GST all remaining processes are correct; a process that is not correct is faulty. A correct process is operational "forever" and can reliably exchange messages with other correct processes. In practice, "forever" means long enough for consensus to terminate.

2.2 Definitions

In this section, we define several concepts that constitute the building blocks of this work.

2.2.1 State-machine replication

Replication is a fundamental approach to building fault-tolerant distributed systems [3; 7]. One approach to replication is state-machine replication that is based on the concept of deterministic state machines. The idea is to model a system as a set of states among which transitions happen deterministically by the execution of commands. Consequently, in state-machine replication all the replicas of a service implement a state machine. Thus, all of the replicas are initialized with the same initial state and by applying the same set of commands in an identical order traverse the same sequence of states and always remain consistent [3].

2.2.2 Consensus

Consensus is a problem in which one or more processes cooperate to select a value which is proposed by one or more participants. Uniform consensus is defined by the primitives *propose(v)* and *decide(v)*, where v is an arbitrary value. Uniform consensus satisfies the following properties:

I. *Uniform integrity:* if a process decides v then some process proposed v.

II. *Uniform agreement:* no two processes decide different values.

III. *Termination:* if one or more correct processes propose a value then eventually some value is decided by all correct processes.

2.2.3 Atomic broadcast

Atomic broadcast, also referred to as total order broadcast, is defined in terms of two primitives: *broadcast(m)* and *deliver(m)* (where m is a message) and should satisfy the following properties:

I. *Validity:* if a correct process *broadcast*s a message m, then all correct processes eventually *deliver* m.

II. *Uniform integrity:* for any message m, every process *deliver*s m at most once and only if m was previously *broadcast* by some process.

III. *Uniform agreement:* if a process *deliver*s m then all correct processes eventually *deliver* m.

IV. *Uniform total order:* if processes p and q both *deliver* messages m and m', then p *deliver*s m before m', if and only if q *deliver*s m before m'.

Atomic broadcast can be implemented as a sequence of consensus instances [8].

2.2.4 Atomic multicast

Atomic multicast is a generalization of atomic broadcast and implements the abstraction of groups $\Gamma = \{g_1, ..., g_\gamma\}$, where for each $g \in \Gamma$, $g \subseteq \Pi$. Processes may belong to one or more groups. If process $p \in g$, we say that p *subscribes* to group g. Atomic multicast is defined by the primitives *multicast*(g, m) and *deliver*(m), and satisfies the following properties:

I. *Validity:* if a correct process *multicast*s a message m to g, then all correct processes in g will eventually *deliver* m .

II. *Uniform integrity:* for any message m, every process p in g *deliver*s m at most once and only if m was previously *multicast* by some process.

III. *Uniform agreement:* if a process *deliver*s m, then all correct processes in g *deliver* m.

IV. *Uniform partial order:* if processes p and q both *deliver* messages m and m', then p *deliver*s m before m', if and only if q *deliver*s m before m'.

If Γ is a singleton, then atomic multicast is equivalent to atomic broadcast.

2.2.5 Consistency

An object that can be concurrently accessed by many processes is called a concurrent object [9]. Interleaving accesses to the same object can sometimes lead to unexpected behaviors. The effect of this issue can be captured by defining a consistency criterion over the shared object, which specifies the level in which operations can interleave in accessing the object. In the following we explain linearizability and sequential consistency as two types of consistency that we are interested in.

- **Linearizability**. A sequence of operations is linearizable if there is a way to reorder the operations in the sequence such that (a) they respect the semantics of the objects as expressed in their sequential specifications, and (b) they respect the real-time ordering of events among all the nodes. For example, if the *response* of operation o_1 occurs in the sequence before the *invocation* of operation o_2 then in the reordering of the operations o_1 appears before o_2.

- **Sequential Consistency**. A sequence of operations is sequentially consistent if there is a way to reorder them such that (a) they respect the semantics of the objects as expressed in their sequential specifications, and (b) if the *response* of operation o_1 at node p_i occurs in the sequence before the *invocation* of operation o_2 at node p_i then in the reordering of the operations o_1 appears before o_2. Basically sequential consistency is a weaker form of linearizability.

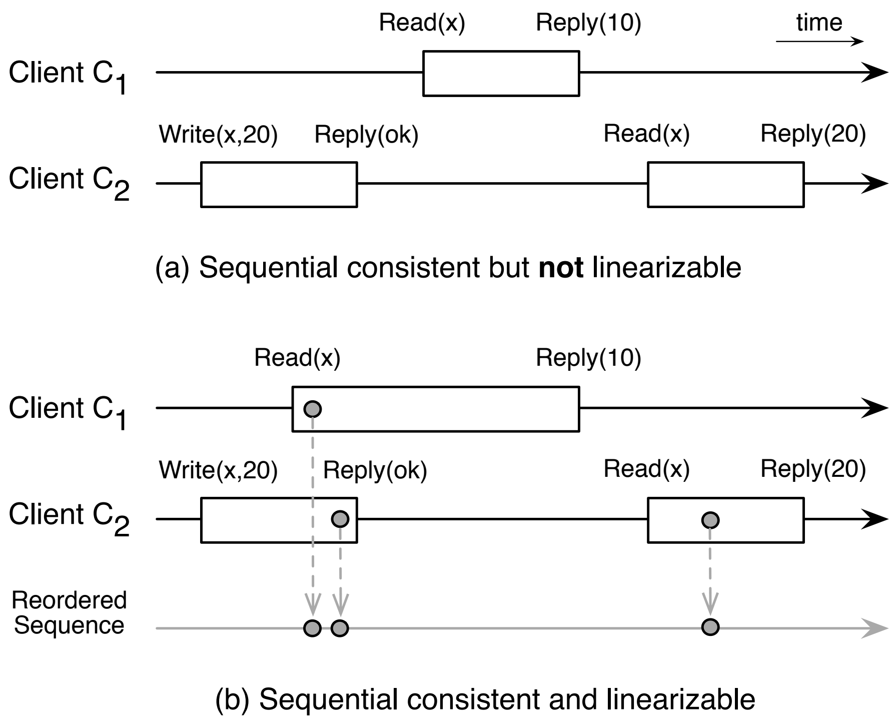


Figure 2.1. Linearizabiliy vs. sequential consistency.

Figure 2.1 illustrates sequential consistency versus linearizability for a shared object x. The sequence shown in (a) is not linearizable since any reordering of the operations that satisfies the specifications of linearizability violates the real-time ordering of the operations. Since the *response* for Write(x, 20) occurs before the *invocation* of Read(x), client $C1$ should read 20 instead of 10. However, this is not the case based on the results. On the other hand, the sequence shown in (b) is linearizable and thus sequentially consistent. This is because the *response* of Write(x) occurs after the *invocation* of Read(x). And thus these two operations can be reordered with respect to each other such that linearizability is ensured without violation of real-time ordering. Both linearizability and sequential consistency are considered strong consistency conditions.

Chapter 3

Ring Paxos

Atomic broadcast is a communication primitive often present at the core of state-machine replication. Because of the canonical role atomic broadcast protocols play in the design of the systems replicated by state-machine approach, their efficiency directly affects the performance of the replicated services. Paxos is a well-known consensus protocol upon which an atomic broadcast protocol can be built. In this chapter we devise Ring Paxos, a new high-performance atomic broadcast protocol that is optimized for local-area networks and inherits the reliabilities of Paxos. The techniques used in the design of Ring Paxos are chosen to maximize its efficiency. Our evaluations at the end of this chapter demonstrate significant performance gains obtained by Ring Paxos.

3.1 Problem statement

Atomic broadcast is a dominant primitive for implementing the agreement layer of the state-machine replication approach. Agreement is a key component of state-machine replication at preserving consistency among the replicas and is responsible for ordering client requests before they can be executed on the replicas. The agreement layer, however, imposes additional performance overhead on a replicated system compared to a stand-alone deployment, the price that must be paid to gain availability and fault tolerance. Therefore, the efficiency of the agreement layer directly affects the performance of replicated services and effort must be put into the design of efficient atomic broadcast protocols. The rich literature of atomic broadcast protocols indicates the considerable amount of effort invested in designing various atomic broadcast protocols with different properties [53].[1] However, building efficient systems out of these algorithms is only seen in a few papers [10].

We define the efficiency of an atomic broadcast protocol to be the rate between its maximum achieved throughput per receiver and the nominal transmission capacity of the system per receiver. For example, a protocol that allows a receiver to deliver up to 500 Mbps of application data in a system equipped with a gigabit network has efficiency 0.5 or 50%. A protocol is efficient if it has high efficiency ($> 90\%$), ideally independent of the number of receivers. However, due to inherent limitations of an algorithm, implementation details, and various overheads (e.g., added by the network layers), typical atomic broadcast protocols are not ideal according to this metric.

In this chapter, we focus on a class of atomic broadcast protocols that are based on Paxos [4]. Paxos is a consensus algorithm upon which an atomic broadcast protocol can be built. Paxos has several important properties that make it popular among system practitioners. For example it is (a) safe under asynchrony assumptions, (b) live under weak synchrony assumptions, and (c) resiliency-optimal (i.e., it requires only a majority of non-faulty processes to ensure progress). This chapter is devoted to the design and implementation of new and highly efficient Paxos-based atomic broadcast protocols to reduce the negative implications of the agreement layer on the performance of a replicated service.

3.1.1 Outline

The rest of this chapter is organized as follows. In Section 3.2 we review the basic form of the Paxos protocol and then present Ring Paxos in Section 3.3. We first motivate the design principles of Ring Paxos and then present in detail its two variations: M-Ring Paxos and U-Ring Paxos. In Section 3.4 we discuss related work and compare the performance of several other protocols with Ring Paxos. In Section 3.5 we experimentally evaluate Ring Paxos and discuss the main findings of our experiments. Finally in Section 3.6 we conclude the chapter.

3.2 Basic Paxos

Paxos is a well-known fault-tolerant consensus algorithm [4] that is used for implementing state-machine replication. There are three distinguished roles in

[1] We review the literature in Section 3.4.

Paxos: proposers, acceptors, and learners. A process can execute multiple roles simultaneously. Proposers propose a value, acceptors choose a value, and learners learn the decided value. Hereafter, N_a denotes the set of acceptors, N_l the set of learners, and Q_a a majority quorum of acceptors (*m-quorum*), that is, a subset of N_a of size $\lceil(|N_a|+1)/2\rceil$.

The execution of one consensus instance spans a sequence of *rounds* that are uniquely identified by a round number which is a positive integer. For each round, one process plays the role of the *coordinator* of the round. To propose, proposers send their value to the coordinator.[2] The coordinator maintains two variables: (a) *c-rnd*: the highest-numbered round that the coordinator has started, and (b) *c-val*: the value that the coordinator has picked for round *c-rnd*. The first is initialized to 0 and the second to null.

Acceptors maintain three variables: (a) *rnd* is the highest-numbered round in which the acceptor has participated, initially 0, (b) *v-rnd* is the highest-numbered round in which the acceptor has cast a vote, initially 0; thus *v-rnd* $\leq$ *rnd* always holds, and (c) *v-val* is the value voted by the acceptor in round *v-rnd* and is initially null.

Paxos, presented in Algorithm 1, has two phases. To execute Phase 1, the coordinator selects a unique round number *c-rnd* greater than any round number it has used so far, and sends it to the acceptors (Task 1, Phase 1A). Upon receiving this message (Task 2, Phase 1B), an acceptor checks whether the round proposed by the coordinator is greater than any round it has received so far; if so, the acceptor promises not to accept any future Phase 1A messages 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 has voted for in that round. Notice that the coordinator does not send any proposal in Phase 1. The coordinator starts Phase 2 after receiving a reply from a m-quorum (Task 3). Before proposing a value in Phase 2A, the coordinator checks to see if some acceptor has already cast a vote in a previous round. If an acceptor has voted for a value in a previous round, then the coordinator will propose this value (this task guarantees that only one value can be chosen in an instance of consensus); otherwise, if no acceptor has cast a vote in a previous round, then the coordinator can propose any value that is received from the proposers. In some cases it may happen that more than one acceptor have cast a vote in a previous round. In this case, the coordinator chooses the value that was voted for in the highest-numbered round. From the algorithm, two acceptors cannot

[2]Selection of the coordinator is not essential for preserving the safety of Paxos, and proposers can directly send their proposals to the acceptors. However, selecting a coordinator, enhances the liveness of the protocol.

```
Algorithm 1: Paxos (for process $p$)
Task 1 (coordinator)
upon receiving value $v$ from proposer $P(v)$
  increase c-rnd to an arbitrary unique value
  for all $q \in N_a$ do send ($q$, (PHASE 1A, c-rnd))
Task 2 (acceptor)
upon receiving (PHASE 1A, c-rnd) from coordinator
  if c-rnd > rnd then
    let rnd be c-rnd
    send (coordinator, (PHASE 1B, rnd, v-rnd, v-val))
Task 3 (coordinator)
upon receiving (PHASE 1B, rnd, v-rnd, v-val) from $Q_a$ such that c-rnd = rnd
  let $k$ be the largest v-rnd value received
  let $V$ be the set of (v-rnd,v-val) received with v-rnd=$k$
  if $k = 0$ then let c-val be $v$
  else let c-val be the only v-val in $V$
  for all $q \in N_a$ do send ($q$, (PHASE 2A, c-rnd, c-val))
Task 4 (acceptor)
upon receiving (PHASE 2A, c-rnd, c-val) from coordinator
  if c-rnd $\geq$ rnd then
    let rnd be c-rnd
    let v-rnd be c-rnd
    let v-val be c-val
    send (coordinator, (PHASE 2B, v-rnd, v-val))
Task 5 (coordinator)
upon receiving (PHASE 2B, v-rnd, v-val) from $Q_a$ such that c-rnd = v-rnd
  for all $q \in N_l$ do send ($q$, (DECISION, v-val))
```

cast votes for different values in the same round.

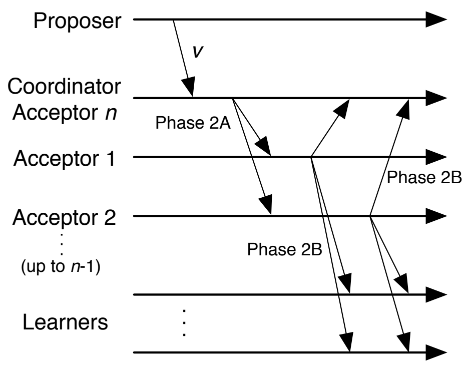


Figure 3.1. Optimized Paxos.

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 round (Task 4, Phase 2B). 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 an m-quorum (Task 5), it knows that a value has been decided and informs the learners about the decision. In order to know whether its value has been decided, a proposer is typically also a learner. If a proposer does not learn its proposed value after a certain time (e.g., because its message to the coordinator was lost), it proposes the value again. As long as a nonfaulty coordinator is eventually selected, there is a majority quorum of nonfaulty acceptors, and at least one nonfaulty proposer, every consensus instance will eventually decide on a value.

Paxos can be optimized in a number of ways [4]. For example, the coordinator can execute Phase 1 before a value is received from a proposer. In doing so, once the coordinator receives a value from a proposer, consensus can be reached in four communication steps, as opposed to six. Moreover, if acceptors send Phase 2B messages directly to the learners, the number of communication steps for a decision is further reduced to three (see Figure 3.1).

3.3 Ring Paxos

Ring Paxos is designed around two main ideas: (a) the separation of message ordering from payload propagation, and (b) the usage of efficient means for performing these two tasks. We have developed two variants of Ring Paxos as M-Ring Paxos and U-Ring Paxos. Before presenting them in detail, in the following section we discuss a set of experiments and consequent findings that have played an important role in shaping these protocols.

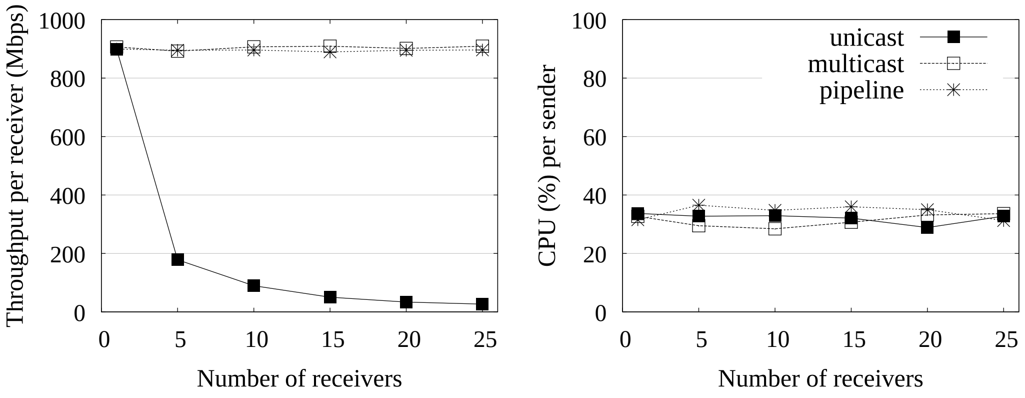


Figure 3.2. Performance comparison of unicast, multicast, and pipeline for implementing one-to-many communication pattern; the coordinator of Paxos uses this type of communication in Phase 1A and Phase 2A to communicate with a set of processes; the left graph shows the throughput per receiver and the right graph shows the CPU usage at the sender; packet size in this experiment is 8 Kbytes.

3.3.1 Motivation and design considerations

Due to the importance of communication efficiency in a message-passing system we have performed a set of experiments to evaluate and compare several communication patterns. Based on the specifications of Paxos, we are mostly interested in one-to-many and many-to-one communications. In the former, a single process (sender) transmits a message to a set of other processes (receivers), and in the latter, a set of processes (senders) transmit messages to one process (receiver). In this section we discuss our findings about these communication patterns and highlight their influence in the design of Ring Paxos protocols.[3]

(A) One-to-many communication. In the following, we compare the performance of three strategies that implement one-to-many communication: (1) network level ip-multicast (multicast), (2) a set of direct unicast links from the

[3]Details about experimental setup are available in Section 3.5

sender to the receivers (unicast), and (3) a uni-directional set of links pipelined among the processes (pipeline).

multicast vs. unicast. Network level ip-multicast enables high-throughput propagation of messages to the nodes of a cluster [11] and has two advantages over unicast. First, independent of the number of receivers only one system call at the sender is sufficient to transmit a message to all the receivers. This is possible because ip-multicast delegates the task of propagating a message to the ethernet switch. In unicast however, the number of system calls at the sender increases linearly with the number of receivers. Second, unlike the multicast sender, the outgoing bandwidth of the unicast sender should be divided among the receivers.

pipeline vs. unicast. A throughput-efficient alternative to multicast is to have nodes communicate in a pipelined pattern. In contrast to unicast where one node sends a message multiple times (equal to the number of receivers), in pipeline each node sends the message once to its successor. Pipeline enables a more balanced usage of CPU and bandwidth resources of the nodes [12].

Applicability to Paxos. It is in Phase 1A and Phase 2A of Paxos that one-to-many communication is needed. Optimizing communication in Phase 2 is more important than in Phase 1 as Phase 1 can be pre-executed for a range of instances and is not frequent. Thus, to efficiently propagate values to other processes in Phase 2A we will use multicast and pipeline in M-Ring Paxos and U-Ring Paxos respectively.

Experimental results. As it is seen in the left-most graph of Figure 3.2 as the number of receivers increases the throughput of each receiver decreases with unicast but remains constant with multicast and pipeline. The performance advantages of multicast and pipeline do not come at the expense of increased CPU utilization as the CPU of the three strategies compare similarly (right-most graph).

Packet loss. Losing packets in a message passing system invokes retransmissions and demands additional CPU and network consumption. Preventing packet loss is a challenging task, but some simple techniques can reduce its possibility. For example, the sending rate at a process can be controlled to prevent it from exceeding network capacity. This prevents message loss at the sender side. If the receiver at the other side of the communication is not fast enough to remove the received packets from its socket buffers, new messages will be dropped. To address this issue, socket buffer sizes can be configured properly. In some deployments the number of senders maybe unknown and relying only on these strategies is not sufficient to address packet loss. We will discuss this issue further in Section 3.3.6.

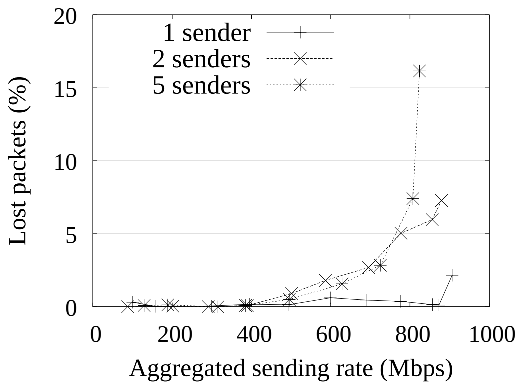


Figure 3.3. The impact of multiple ip-multicast senders on packet loss; 14 receivers are used in this experiment.

We have performed an experiment with several ip-multicast senders and measured the amount of packet loss (see Figure 3.3). As it is seen in the graph by increasing the number of senders, the rate at which packet loss starts happening reduces. The aggregate sending rate in these experiments was controlled not to exceed the capacity of the network.

(B) Many-to-one communication. In the following, we compare the performance of two strategies that implement many-to-one communication: (1) a set of direct unicast links from the senders to the receiver (unicast), and (2) a unidirectional set of links pipelined among the processes (pipeline).

pipeline vs. unicast. Pipeline has two advantages over unicast. First, it enables extensive use of batching. Each process on the line appends its own message to the one received from its predecessor and forwards it to its successor. Batching is advantageous to the receiver, as the receiver receives only one packet that contains many small messages in contrast to many small messages that are received in unicast. This reduces CPU usage at the receiver. Second, it balances the usage of the incoming and outgoing links of all the nodes. In unicast, senders only use their outgoing bandwidth.

Applicability to Paxos. It is in Phase 1B and Phase 2B of Paxos that many-to-one communication is needed. Both M-Ring Paxos and U-Ring Paxos protocols pipeline acceptors to propagate votes to the coordinator. For example, upon receiving a Phase 2A message, the first acceptor at the head of

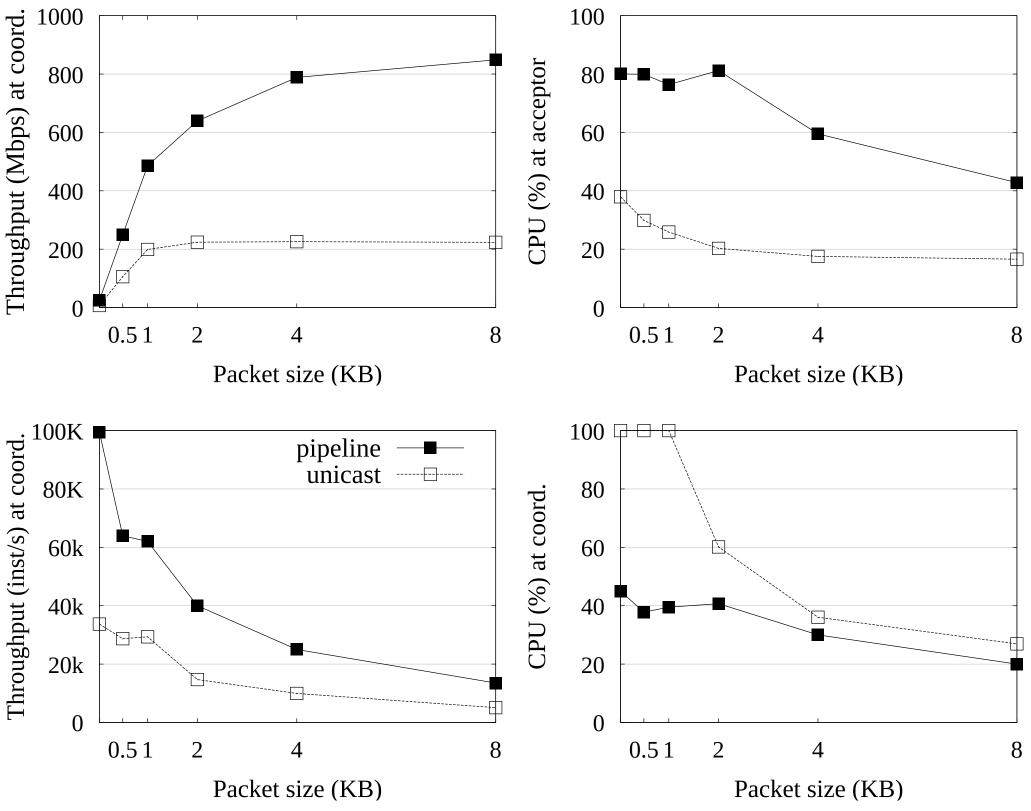


Figure 3.4. Performance comparison of pipeline and unicast for implementing many-to-one communication pattern; acceptors of Paxos protocol use this type of communication in Phase 1B and Phase 2B to propagate their votes to the coordinator; in all the experiments we have deployed four acceptors and one coordinator while varying the size of the packets; top left graph shows the receiving throughput from only one of the incoming links at the coordinator (which are in total one and four for pipeline and unicast respectively); bottom left graph shows the corresponding number of instances decided at the coordinator; the graphs on the right show the CPU usage at the acceptors (top) and the coordinator (bottom); for pipeline, CPU is shown for an acceptor at the middle of the path.

the pipeline builds a Phase 2B message and forwards it to its successor. Forwarding continues until the message reaches the coordinator.

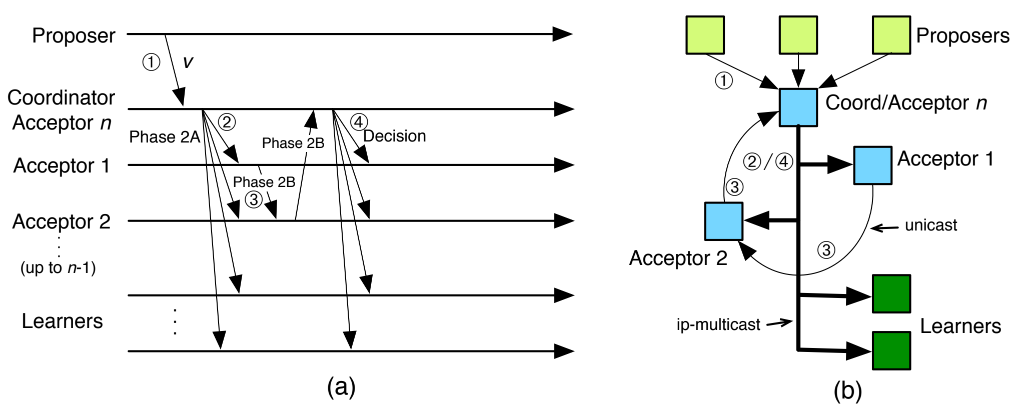


Figure 3.5. M-Ring Paxos.

Experimental results. We compare the efficiency of pipeline and unicast with respect to different message sizes with five acceptors, where one of them plays the role of the coordinator. With respect to throughput, pipeline is preferable to unicast. For small messages, this happens because pipeline is less CPU-intensive at the coordinator. For large messages, the advantage stems from a balanced use of the incoming and outgoing bandwidth of the nodes. Pipeline is more CPU-intensive at the acceptors as each acceptor (except the first one) receives and forwards a Phase 2B message, whereas the coordinator only receives Phase 2B messages.[4]

3.3.2 Multicast-based Ring Paxos (M-Ring Paxos)

In this section we present the Ring Paxos algorithm that is based on multicast communication (M-Ring Paxos hereafter). This protocol is outlined in Algorithm 2. Statements in gray are the same for Paxos and M-Ring Paxos. Similarly to Paxos, the protocol is executed in two phases and the mechanism to ensure that only one value can be decided in an instance of consensus is the same as in Paxos. Differently from Paxos, M-Ring Paxos disseminates a majority quorum of acceptors (m-quorum) in a logical and directed ring (see Figure 3.5(a)(b)). The coordinator is the last process on the ring and it also plays the role of acceptor.

[4]As we will show later, acceptors are not the most CPU-intensive nodes when the rest of the protocol is also considered.

From Section 3.3.1 we know that placing acceptors in a ring reduces the number of incoming messages at the coordinator and balances the communication among the acceptors. Before the coordinator starts Phase 1, it proposes the ring on which acceptors will be located. By replying to the coordinator, the acceptors acknowledge that they abide by the proposed ring. Since Phase 1 can be executed before values are proposed, there is little gain in optimizing the sending of Phase 1B messages. Thus, we use the ring only to propagate Phase 2B messages.

In addition to checking what value can be proposed in Phase 2 (Task 3), the coordinator also creates a unique identifier for the value to be proposed. Ring Paxos executes consensus on value ids [13; 14]; proposed values in Phase 2A messages are disseminated to the m-quorum and to the learners using ip-multicast.

Upon ip-delivering a Phase 2A message (Task 4), an acceptor checks that it can vote for the proposed value. If so, it updates its *v-rnd* and *v-val* variables as in Paxos, and its *v-vid* variable in addition to them. Variable *v-vid* contains the unique identifier of the proposed value which is initialized with null. The first acceptor in the ring sends a Phase 2B message to its successor in the ring. Learners also ip-deliver the proposed value but they do not learn it since it has not been accepted yet.

The next acceptor in the ring to receive a Phase 2B message (Task 5) checks whether it has ip-delivered the value proposed by the coordinator in a Phase 2A message. The acceptor can only vote if it has received the value and not only its unique identifier. The check is done by comparing the acceptor's *v-vid* variable to the value's identifier chosen by the coordinator. This is to ensure that when consensus is reached, a majority of acceptors knows the chosen value and this value can be retrieved by learners at any time. If the condition holds (i.e., the corresponding value is ip-delivered) then there are two possibilities: either the acceptor is not the coordinator (i.e., the last process in the ring), in which case it sends a Phase 2B message to its successor, or it is the coordinator and then it ip-multicasts a decision message including the identifier of the chosen value. Once a learner ip-delivers this message, it can learn the value received previously from the coordinator through a Phase 2A message.

Ring Paxos can use a number of optimizations, most of which have been described previously in the literature. For example, when a new coordinator is elected it executes Phase 1 for several consensus instances [4]. Another optimization is to execute Phase 2 for a batch of proposed values rather than for a single value (e.g., [10; 15]). In addition, multiple consensus instances can be started simultaneously without the previous ones being finished [4].

Placing a majority of acceptors in the ring, as opposed to placing all of them, reduces the number of communication steps to reach a decision. The remaining

```
Algorithm 2: M-Ring Paxos (for process $p$)
Task 1 (coordinator)
upon receiving value $v$ from proposer
    increase $c\text{-}rnd$ to an arbitrary unique value
    for all $q \in Q_a$ do send ($q$, (PHASE 1A, $c\text{-}rnd$))
Task 2 (acceptor)
upon receiving (PHASE 1A,$c\text{-}rnd$) from coordinator
    if $c\text{-}rnd > rnd$ then
        let $rnd$ be $c\text{-}rnd$
        send (coordinator, (PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$))
Task 3 (coordinator)
upon receiving (PHASE 1B, $rnd$, $v\text{-}rnd$, $v\text{-}val$) from $Q_a$ such that $rnd = c\text{-}rnd$
    let $k$ be the largest $v\text{-}rnd$ value received
    let $V$ be the set of ($v\text{-}rnd$,$v\text{-}val$) received with $v\text{-}rnd = k$
    if $k = 0$ then let $c\text{-}val$ be $v$
    else let $c\text{-}val$ be the only $v\text{-}val$ in $V$
    let $c\text{-}vid$ be a unique identifier for $c\text{-}val$
    ip-multicast ($Q_a \cup N_l$, (PHASE 2A, $c\text{-}rnd$, $c\text{-}val$, $c\text{-}vid$))
Task 4 (acceptor)
upon ip-delivering (PHASE 2A, $c\text{-}rnd$, $c\text{-}val$, $c\text{-}vid$)
    if $c\text{-}rnd \geq rnd$ then
        let $rnd$ be $c\text{-}rnd$
        let $v\text{-}rnd$ be $c\text{-}rnd$
        let $v\text{-}val$ be $c\text{-}val$
        let $v\text{-}vid$ be $c\text{-}vid$
        if $p = first(ring)$ then
            send ($successor(p, ring)$, (PHASE 2B, $c\text{-}rnd$, $c\text{-}vid$))
Task 5 (coordinator and acceptors)
upon receiving (PHASE 2B,$c\text{-}rnd$,$c\text{-}vid$)
    if $v\text{-}vid = c\text{-}vid$ then
        if $p \neq last(ring)$ then
            send ($successor(p, ring)$, (PHASE 2B, $c\text{-}rnd$, $c\text{-}vid$))
        else
            ip-multicast ($Q_a \cup N_l$, (DECISION, $c\text{-}vid$))

Note: $first(ring)$: process that succeeds the coordinator in
      $ring$
      $last(ring)$: the coordinator process in $ring$
      $successor(p, ring)$: process that succeeds $p$ in $ring$
```

acceptors are spares, used only when an acceptor in the ring fails.[5] Finally, although ip-multicast is used by the coordinator in Tasks 3 and 5, this can be implemented more efficiently by overlapping consecutive consensus instances, such that the message sent by Task 5 of consensus instance i is ip-multicast together with the message sent by Task 3 of consensus instance $i+1$.

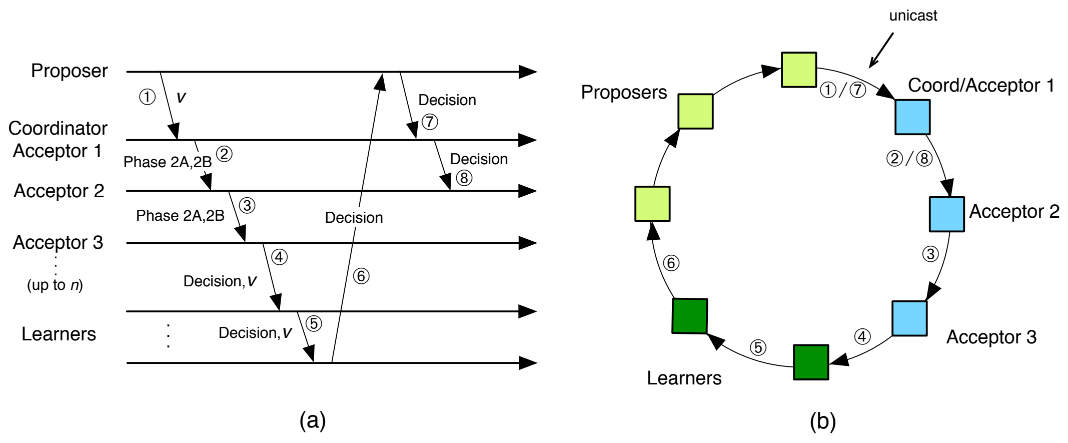


Figure 3.6. U-Ring Paxos.

3.3.3 Unicast-based Ring Paxos (U-Ring Paxos)

In some data centers network-level ip-multicasting may not be available. In this section we present a variant of Ring Paxos algorithm (U-Ring Paxos) that is entirely based on unicast communication. In Algorithm 3 statements in gray are the same for M-Ring Paxos and U-Ring Paxos. Similarly to Paxos, the protocol is executed in two phases and the mechanism to ensure that only one value can be decided in an instance of consensus is the same as in Paxos.

In addition to a m-quorum of acceptors, U-Ring Paxos also places proposers and learners in a logical directed ring (see Figure 3.6). Recalling from Section 3.3.1 pipelining all the processes in a ring is an alternative to multicast that can reach high throughput, a fact that was proved in [12]. Processes in the ring can assume multiple roles and there is no restriction on the relative position of these processes in the ring. However, for simplicity of discussion, hereafter it is assumed that acceptors are lined up one after the other in the ring (see Figure 3.6(b)). To reduce latency, the coordinator is the first acceptor in the ring.

[5]This idea is conceptually similar to Cheap Paxos [14], although Cheap Paxos uses a reduced set of acceptors in order to save hardware resources, and not to reduce latency.

Once a proposer proposes a value, it is forwarded along the ring until it reaches the coordinator, which will execute Phase 1 as in Paxos. When the coordinator receives Phase 1B messages from an m-quorum (Task 3), it will check which value can be proposed and assigns a unique identifier to the value to be proposed as in M-Ring Paxos. The coordinator then sends Phase 2A and Phase 2B messages to its successor in the ring (Task 3). Similarly to Paxos and M-Ring Paxos, the coordinator in U-Ring Paxos can execute Phase 1 before a value is proposed, reducing the latency of the protocol. Upon receiving a Phase 2A/2B message (Task 4), an acceptor checks that it can vote for the proposed value. If so, it updates its *v-rnd*, *v-val*, and *v-vid* variables. If the acceptor does not precede the last acceptor in the ring, it sends the Phase 2A/2B message to its successor. Differently from M-Ring Paxos where the coordinator checks whether a decision has been made in the instance, in U-Ring Paxos this task is delegated to the last acceptor in the ring. After deciding, the last acceptor sends the decision, possibly together with the value chosen, to its successor in the ring.

Forwarding the chosen-value ends at the predecessor of the process that has proposed the chosen value as at this point the value has been received by all the processes in the ring. The decision, i.e., the chosen-value identifier, should be forwarded along the ring until it reaches the predecessor of the last acceptor (Task 5).

3.3.4 Handling message loss

Because of unreliable communication mechanisms in M-Ring Paxos, message losses are unavoidable. In general, lost messages are handled with retransmissions. Because of message loss, three cases may happen with the learners: (a) they receive the proposed value but not the notification that it was accepted, (b) they do not receive the proposed value but receive the notification of its acceptance, or (c) they receive neither the value nor the acceptance notification. In all the cases, the learners can recover lost messages by inquiring other processes. M-Ring Paxos assigns each learner to a preferential acceptor in the ring, which the learner can contact to retrieve the lost messages. Apart from that in both M-Ring Paxos and U-Ring Paxos learners may not have sufficient time to handle the decisions, and this may cause learners to drop some decisions. In fact, both versions of Ring Paxos need flow control to mitigate this phenomenon. We discuss flow control in Section 3.3.6.

If the coordinator does not receive a response to its Phase 1A / 2A messages, it re-sends them, possibly with a bigger round number. Eventually the coordinator will receive a response or will suspect the failure of a process (this suspicion

```
1: Algorithm 3: U-Ring Paxos (for process $p$)
2: Task 1 (all)
3: upon receiving value $v$ proposed by $P(v)$ from $predecessor(p, ring)$
4:    if $p =$ coordinator then
5:       increase c-rnd to an arbitrary unique value
6:       for all $q \in Q_a$ do send ($q$, (PHASE 1A, c-rnd))
7:    else
8:       send $v$ to $successor(p, ring)$
9: Task 2 (acceptor)
10: upon receiving (PHASE 1A,c-rnd) from coordinator
11:    if $c\text{-}rnd > rnd$ then
12:       let rnd be c-rnd
13:       send (coordinator, (PHASE 1B, rnd, v-rnd, v-val))
14: Task 3 (coordinator)
15: upon receiving (PHASE 1B, rnd, v-rnd, v-val) from $Q_a$ such that $rnd = c\text{-}rnd$
16:    let $k$ be the largest v-rnd value received
17:    let $V$ be the set of (v-rnd,v-val) received with $v\text{-}rnd = k$
18:    if $k = 0$ then let c-val be $v$
19:    else let c-val be the only v-val in $V$
20:    let c-vid be a unique identifier for c-val
21:    send ($successor(p, ring)$, (PHASE 2A/2B, c-rnd, c-val, c-vid))
22: Task 4 (acceptor)
23: upon receiving (PHASE 2A/2B, c-rnd, c-val, c-vid)
24:    if $c\text{-}rnd \geq rnd$ then
25:       let rnd be c-rnd
26:       let v-rnd be c-rnd
27:       let v-val be c-val
28:       let v-vid be c-vid
29:       if $p = last\_acceptor(ring)$ then
30:          Send_Decision(c-vid, c-val)
31:       else
32:          send ($successor(p, ring)$, (PHASE 2A/2B, c-rnd, c-val, c-vid))
33: Task 5 (all)
34: upon receiving (DECISION, c-vid, c-val)
35:    if $p \neq predecessor(last\_acceptor(ring))$
36:       Send_Decision(c-vid, c-val)
37: Send_Decision(c-vid, c-val)
38: if $p \neq predecessor(P(c\text{-}val), ring)$
39:    send ($successor(p, ring)$, (DECISION, c-vid, c-val))
40: else
41:    send ($successor(p, ring)$, (DECISION, c-vid, –))

    Note: $P(v)$: proposer of value $v$
          $predecessor(p, ring)$: process that precedes $p$ in $ring$
          $successor(p, ring)$: process that succeeds $p$ in $ring$
          $last\_acceptor(ring)$: the $f$-th acceptor after the
          coordinator in $ring$
```

might be erroneous). In this situation, the coordinator lays out a new ring and excluds the suspected process.

3.3.5 Handling process crashes

A coordinator that suspects the failure of one or more acceptors may simply try to contact all the acceptors in order to gather an m-quorum. This solution would reduce throughput but allows progress despite failures. With both protocols, when an acceptor replies to a Phase 1A or to a Phase 2A message, it must not forget its state (i.e., variables *rnd*, *ring*, *v-rnd*, *v-val*, and *v-vid*) despite failures. There are two ways to ensure this. First, by assuming that a majority of acceptors never fails. Second, by requiring acceptors to keep their state on the stable storage before replying to Phases 1A and 2A messages. Finally, a failed coordinator is detected by the other processes, which select a new coordinator. Before GST (see Section 2.1) it is possible that multiple coordinators co-exist. However, similarly to Paxos, Ring Paxos guarantees safety even when multiple coordinators co-exist, although it may not guarantee liveness. After GST, eventually a single correct coordinator is selected.

3.3.6 Flow control

In Ring Paxos, flow control helps regulate the speed at which consensus instances are executed. In doing so, we not only reduce the likelihood of message loss for both unicast and multicast communications, but we also ensure that learners are given enough time to process decisions. For instance, when Ring Paxos is used to implement state-machine replication, values represent commands that read or write the application state, and may need extra processing time. To illustrate why flow control is important, consider a scenario where commands are ordered faster than they can be processed at the leaners. Without flow control, buffers that store commands at learners will eventually overflow and new messages will be dropped. This will cause learners to do extra work to retrieve the lost decisions. As a consequence, the performance of the replicated system may decrease to the point where clients time out and retransmit their commands frequently. In the following we discuss the flow control in M-Ring Paxos and U-Ring Paxos protocols:

- In M-Ring Paxos, communication is based on UDP and flow control at the coordinator ensures that messages are sent at a rate the network can handle. With M-Ring Paxos, since learners are not part of the ring, we use the following mechanism for flow control. Learners constantly monitor their buffers for decisions that remain to be processed. When the occupied buffer space reaches a certain threshold, they notify one of the acceptors. The notification tells the acceptors about the number of unprocessed requests at the learner. Acceptors forward this notification along the ring until it reaches the coordinator. The coordinator reduces the *window* of outstanding consensus instances and thus opens fewer instances in parallel. Provided that the coordinator slows down sufficiently, learners will be able to process decisions as fast as they are ordered, and eventually they will stop sending notifications to the acceptors. This technique allows learners to slow down the coordinator before decisions are dropped. In case some decisions are lost, for instance if the coordinator starts with a window that is too large, lost decisions are retrieved from the acceptors. To allow M-Ring Paxos to recover from temporarily slow learners, the coordinator gradually increases its window size if it does not receive new notifications from the learners.

- It is easier to implement flow control in U-Ring Paxos since communication between two consecutive processes in the ring is done using TCP. Accordingly, TCP buffers are made sufficiently large to take into account the processing time of Phase 1 and 2 messages. To ensure that learners have enough time to handle the decided values, U-Ring Paxos (a) lets learners process a decision before forwarding it to the next process in the ring and (b) limits the number of outstanding consensus instances.

3.3.7 Garbage collection

As we have seen in the description of the protocols, an acceptor stores several variables for each instance of Ring Paxos it participates in. The variables are: (a) the highest-round *rnd* in which the acceptor executed a Phase 1 or 2, (b) the highest-round *v-rnd* in which the acceptor cast a vote in Phase 2, as well as (c) the corresponding value *v-val* and the identifier of the value *v-vid* the acceptor voted for. Variable *rnd* is shared across consensus instances and does not need to be garbage collected. The other variables are discarded when $f+1$ learners have applied the corresponding decision to their application state. Each learner maintains its *version*, the largest instance for which it has applied the corresponding decision. Learners apply decisions in instance order so if a learner has applied decision of instance x, it also has applied all decisions of instances lower than x. In M-Ring Paxos, each learner periodically communicates its version to one of the acceptors (learners are assigned different acceptors to balance the associated load), and acceptors propagate this information along the ring. Once an acceptor receives a version from $f+1$ learners, it computes the smallest version and garbage collects variables for instances up to this version. In U-Ring Paxos, learners are part of the ring and directly forward their version to their successor.

The coordinators of M-Ring Paxos and U-Ring Paxos store two variables, (a) the highest-round started *c-rnd* and (b) the value picked *c-val* for a particular instance and round. Variable *c-rnd*, similarly to *rnd*, is shared across instances and does not need to be garbage collected. The value picked for a given instance and round can be discarded as soon as the coordinator receives the corresponding Phase 2B messages from its $f+1$ acceptors. Acceptors store the decisions sent by the coordinator in order to let learners retrieve decisions that they may not have received. These decisions can be discarded similarly as with the acceptor variables. If a learner requests a decision that has been garbage collected, the learner can be brought up to date by communicating with a learner with a sufficiently recent version, i.e., one that is larger than the instance of the decision missing at the learner. Such a learner will always exist since we garbage collect a decision only after it is reflected in the state of $f+1$ learners.[6]

3.4 Related work

Paxos is a subtle algorithm and its description leaves many non-trivial design decisions open. Several papers have argued that Paxos is not an easy algorithm to implement [16; 10; 17]. Besides providing insight into the difficulty of implementing Paxos, two of these papers present performance results of their Paxos implementations. In [15], an analytical analysis of the impact of several optimizations on the performance of the basic Paxos algorithm is presented. The authors also present extensive experimental results of their implementation in both LAN and WAN environments. Several systems use Paxos to provide various abstractions such as storage systems [18], locking services [19], and distributed databases [20]. Paxos is not the only algorithm to implement atomic broadcast. Some protocols implement atomic broadcast through the virtual synchrony model introduced by the Isis system [21]. With virtual synchrony, processes are part of a group. They may join and leave the group at any time. When processes are suspected of crashing they are evicted from the group; virtual synchrony ensures that processes observe the same sequence of group memberships or *views*

[6]For the proof of correctness for M-Ring Paxos and U-Ring Paxos protocols see Appendix.

and non-faulty members deliver the same set of messages in each view. Implementing such properties requires solving consensus.

In [53], five classes of broadcast algorithms have been identified: fixed sequencer, moving sequencer, destination agreement, communication history-based, and privilege-based. Below, we review the five classes of atomic broadcast protocols.

In fixed sequencer algorithms (e.g., [22; 23]), broadcast messages are sent to a distinguished process, called the sequencer, who is responsible for ordering these messages. The role of the sequencer is unique and only transferred to another process in case of failure of the current sequencer. In this class of algorithms, the sequencer may eventually become the system bottleneck.

Moving sequencer protocols are based on the observation that rotating the role of the sequencer distributes the load associated with ordering messages among processes. The ability to order messages is passed from process to process using a *token*. The majority of moving sequencer algorithms are optimizations of [24]. These protocols differ in the way the token circulates in the system: in some protocols the token is propagated along a ring [24; 25], in others, the token is passed to the least loaded process [26]. All the moving sequencer protocols we are aware of are based on the broadcast-broadcast communication pattern. According to this pattern, to atomically broadcast a message m, m is propagated to all processes in the system; the token holder process then replies by broadcasting a unique global sequence number for m. High-throughput can be obtained by resorting to network-level broadcast. Mencius is another moving sequencer-based protocol that implements state-machine replication and is derived from Paxos [27]. Mencius is designed for wide-area networks in which optimizing for latency is the main objective in contrast to throughput, the focus of Ring Paxos.

Protocols falling in the destination agreement class compute the message order in a distributed fashion (e.g., [8; 28]). These protocols typically exchange a quadratic number of messages for each message broadcast, and thus are not good candidates for high throughput.

In communication history-based algorithms, the message ordering is determined by the message sender, that is, the process that broadcasts the message (e.g., [29; 30]). Message ordering is usually provided using logical or physical time. Of special interest is LCR, which arranges processes along a ring and uses vector clocks for message ordering [12]. This protocol has similar throughput to our Ring Paxos protocols but requires *perfect failure detection*: erroneously suspecting a process to have crashed is not tolerated. Perfect failure detection implies strong synchrony assumptions about processing and message transmis-

Table 3.1. Comparison of several atomic broadcast algorithms (f: number of tolerated failures).

Algorithm	Class	Communication Steps	Number of processes	Synchrony assumptions
LCR [12]	comm. history	$2f$	$f+1$	strong
Totem [31]	privilege	$(4f+3)$	$2f+1$	weak
Ring+FD [13]	privilege	(f^2+2f)	$f(f+1)+1$	weak
S-Paxos [32]	—	5	$2f+1$	weak
M-Ring Paxos	—	$(f+3)$	$2f+1$	weak
U-Ring Paxos	—	$5f$	$2f+1$	weak

sion times.

The last class of atomic broadcast algorithms, denoted as privilege-based, allows a single process to broadcast messages at a time; the message order is thus defined by the broadcaster. Similarly to moving sequencer algorithms, the privilege to order messages circulates among broadcasters in the form of a token. Differently from moving sequencer algorithms, message ordering is provided by the broadcasters and not by the sequencers. In [31], the authors propose Totem, a protocol based on the virtual synchrony model. In the case of process or network failures, the ring is reconstructed and the token regenerated using the new group membership. In [13], fault-tolerance is provided by relying on a failure detector; tolerating f process failures requires a quadratic number of processes. A general drawback of privilege-based protocols is their high latency: before a process p can totally order a message m, p must receive the token, which delays m's delivery. M-Ring Paxos and U-Ring Paxos combine ideas from several broadcast protocols to provide high throughput and low latency. In this sense, they fit multiple classes, as defined above. To ensure high throughput, our protocols decouple message dissemination from ordering. The former is accomplished using ip-multicast or pipelined unicast; the latter is done using consensus on message identifiers. To use the network efficiently, processes executing consensus communicate using a ring, similarly to the majority of privilege-based protocols.

In Table 3.1, we compare algorithms that are closest to our Ring Paxos protocols in terms of throughput efficiency. Some of these protocols use a logical ring for process communication, which is an effective communication pattern when optimizing for throughput. For each algorithm, we report its class, the minimum number of communication steps required by the last process to deliver a message, the number of processes required as a function of f, and the synchrony assumption needed for correctness. For the Ring Paxos protocols, we assume that each process plays the roles of proposer, acceptor, and learner. There are $f+1$ processes in the ring of M-Ring Paxos and $2f+1$ processes in the ring of

U-Ring Paxos (i.e,. all processes are in the ring).

With M-Ring Paxos, delivery occurs as soon as messages make one revolution around the ring. Its latency is $f + 3$ message delays since each message is first sent to the coordinator, circulates around the ring of $f + 1$ processes, and is delivered after the final ip-multicast is received. With U-Ring Paxos, the worst case latency is $5f$. This happens when the process that broadcasts the message follows the coordinator in the ring. It takes $2f$ steps to reach the coordinator, and another f steps for the decision. The decision must circulate around the ring in order to reach all processes, taking another $2f$ steps.

LCR requires two revolutions and thus has a latency in between the two Ring Paxos algorithms. In Totem, each message must also rotate twice along the ring to guarantee *safe-delivery*, a property equivalent to uniform agreement: if a process (correct or not) delivers a message m then all correct processes eventually deliver m. The atomic broadcast protocol in [13] has a latency that is quadratic in f since a ring requires more than f^2 nodes.

S-Paxos [32] is another implementation of Paxos. The key idea in S-Paxos is to distribute the tasks of request reception and dissemination among all replicas. A client selects a replica arbitrarily and submits its requests to it. After receiving a request, a replica forwards it to all the other replicas. A replica receiving a forwarded request sends an acknowledgement to all other replicas. When a replica receives $f + 1$ acknowledgements, it declares the request as stable. As in basic Paxos, the leader is responsible for ordering requests; differently from Paxos, ordering is performed on request ids. S-Paxos makes a balanced use of CPU and network resources; on the negative side, many messages must be exchanged before a request can be ordered. Due to the number of messages exchanged, this protocol is CPU-intensive.

3.5 Experimental evaluation

In this section, we briefly describe the implementation of Ring Paxos protocols and evaluate our prototypes with respect to the following aspects:

- **Ring Paxos versus other protocols.** Our main goal in designing Ring Paxos protocols was to optimize for throughput. Thus in this experiment we evaluate throughput of Ring Paxos protocols and compare it to some other atomic broadcast protocols (Section 3.5.3).

- **Impact of ring size on performance.** Pipelining processes on a ring has several advantages for throughput, but it negatively affects the latency.

We perform this experiment to assess the effect of pipelining on latency (Section 3.5.4).

- **Impact of disk writes on performance.** Recording data on disk is essential to preventing the loss of critical information in Paxos. In this experiment we show the effect of synchronous disk writes on the performance of Ring Paxos (Section 3.5.5).

- **Impact of message sizes on performance.** The performance of Ring Paxos protocols is affected by the size of the requests submitted to the coordinators. The goal of this experiment is to measure the effect of various message sizes on the throughput of Ring Paxos (Section 3.5.6).

- **Impact of socket buffer sizes on performance.** Message losses happen relatively often because of high network traffic. Increasing socket buffer sizes is a simple solution to solve this issue. In this experiment we measure the performance of Ring Paxos protocols with different socket buffer sizes (Section 3.5.7).

- **Efficiency of flow control.** Adjusting socket sizes alone is not sufficient to entirely prevent message losses. In this experiment we evaluate the efficiency of our flow control mechanism in M-Ring Paxos protocol (Section 3.5.8).

3.5.1 Hardware settings

We ran the experiments in a cluster of Dell SC1435 nodes equipped with 2 dual-core AMD-Opteron 2.0 GHz CPUs and 4GB of main memory. The servers were interconnected with an HP ProCurve2900-48G Gigabit switch. The round trip time is 0.1 milisecond. For the experiments with disk writes we use OCZ-VERTEX3 SSDs. In the experiments every process is deployed on a dedicated machine.

3.5.2 Implementation

In this section we review several properties of M-Ring Paxos and U-Ring Paxos's implementations:

- In M-Ring Paxos each process allocates 160 Mbytes of memory for a circular buffer. Acceptors and learners use this buffer to match proposal ids to proposal contents, as these are decomposed by the coordinator. Messages received out of sequence (e.g., because of transmission losses) are stored in the buffer until they can be delivered (i.e., learned) in order. Each packet sent by the coordinator is composed of two parts: one part stores the ids of decided values, and the other stores new proposed values with their unique ids. A buffer entry is allowed to be freed only after the coordinator has received the entry's Phase 2B message and ip-multicast the corresponding decision. Unless mentioned otherwise, the size of packets in the experiments with M-Ring Paxos is 8 Kbytes.

- In U-Ring Paxos each process maintains a circular buffer to store packets. Each process devotes 16 Mbytes of its circular buffer to each proposer (e.g., with five proposers the total space needed for the buffer in one process is 80 Mbytes). In U-Ring Paxos, it is the last acceptor in the ring that checks whether a decision has been reached. Thus each message originated in this process includes the ids of decided values. This message is carried along the ring until all the processes are informed about the decided values. The coordinator can piggyback new proposals on this message before forwarding it. A buffer entry can be freed only after the corresponding Phase 2B message is received by the last acceptor and the decision is forwarded along the ring. Unless mentioned otherwise, the size of packets in the experiments with U-Ring Paxos is 32 Kbytes.

3.5.3 Ring Paxos versus other protocols

We compare the throughput of Ring Paxos to five other atomic broadcast protocols: LCR [12], Spread [33], Libpaxos [34], S-Paxos [32], and the protocol presented in [10], which hereafter we refer to as PFSB. LCR is a ring-based protocol that achieves very high throughput (see also Section 3.4). Spread is one of the most-used group communication toolkits and is based on Totem [31]. Libpaxos, PFSB, and S-Paxos are all implementations of Paxos. Libpaxos is entirely based on ip-multicast; and PFSB is entirely based on unicast. S-Paxos is a unicast-based implementation of Paxos which disseminates the task of receiving and forwarding client requests among all the acceptors.

We have implemented LCR, and Ring Paxos protocols, and used the open source disseminations of S-Paxos, Libpaxos, and Spread. The performance data for PFSB is taken from [10]. The setup reported in [10] has slightly more powerful processors than the ones used in our experiments, but both setups use a gigabit switch. We have tuned Spread for the best performance we could achieve after varying the number of daemons, number of readers and writers and their

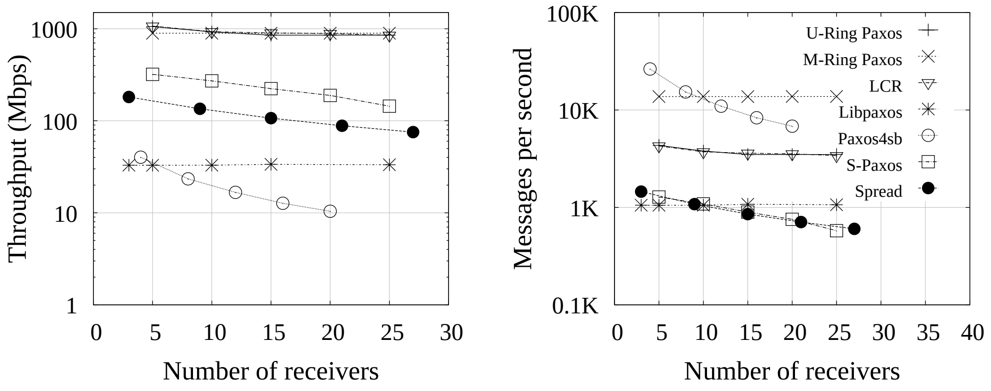


Figure 3.7. Ring Paxos versus other atomic broadcast protocols (message sizes c.f. Table 3.2). In PFSB, U-Ring Paxos, and LCR the number of receivers is equal to the total number of processes. In Libpaxos and M-Ring Paxos it is equal to the number of learners. In Spread it is equal to the number of readers. For Ring Paxos protocols f is equal to two; notice that in both graphs the y-axis is in log scale.

locations in the network, the message size, and some other parameters suggested by the support team of Spread. In the experiments that we report we have used a configuration with 3 daemons in the same segment, one writer per daemon, and a number of readers evenly distributed among the daemons.

Figure 3.7 shows the throughput in megabits per second (left graph) and the number of messages delivered per second (right graph) as the number of receivers increases. For all the protocols, with the exception of PFSB, we explored the space of message sizes and selected the value corresponding to the best throughput. Table 3.2 shows the message sizes used in our experiments. We assess the effect of different message sizes on the performance of Ring Paxos in Section 3.5.6. As it is seen in the graph on the left of Figure 3.7, protocols based on a ring only (LCR and U-Ring Paxos), on ip-multicast (Libpaxos), and on both (M-Ring Paxos) present throughput approximately constant with the number of receivers.

3.5.4 Impact of processes in the ring

We now consider how the number of processes affects the throughput and latency of the Ring Paxos protocols, LCR, and S-Paxos. In Figure 3.8, the x-axis shows the number of acceptors in M-Ring Paxos, U-Ring Paxos, and S-Paxos.

Table 3.2. Protocol efficiency and message sizes in the experiments of Figure 3.7; values used for calculating efficiency correspond to 10 processes.

Protocol	**Message size**	**Efficiency**
LCR	32 kbytes	91%
U-Ring Paxos	*32 kbytes*	*90.4%*
M-Ring Paxos	*8 kbytes*	*90%*
S-Paxos	32 kbytes	31.2%
Spread	16 kbytes	18%
PFSB	200 bytes	4%
Libpaxos	4 kbytes	3%

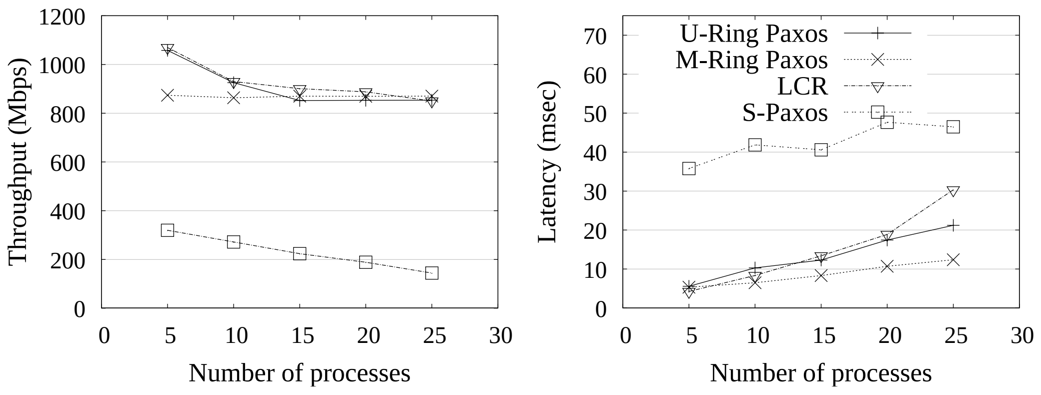


Figure 3.8. Performance when varying the number of processes in the protocols; except for S-Paxos, the number of processes represents the size of the ring.

In U-Ring Paxos every acceptor is also a proposer and a learner. LCR does not distinguish process roles and requires all processes to be in the ring.

M-Ring Paxos has constant throughput with the number of processes in the ring. Throughput of LCR and U-Ring Paxos slightly decreases as processes are added. With few processes, LCR and U-Ring Paxos can achieve efficiency greater than one, which may look counterintuitive. This happens because in a ring with n processes, $1/n$ of the messages delivered by a process are created by the process itself. Thus, the process can use its available incoming bandwidth to receive messages broadcast by the other processes [12]. In order for LCR and U-Ring Paxos to achieve high throughput, every process on the ring must broadcast messages. M-Ring Paxos does not have this constraint.

Figure 3.8 shows the latency measured at the message's proposer. In U-

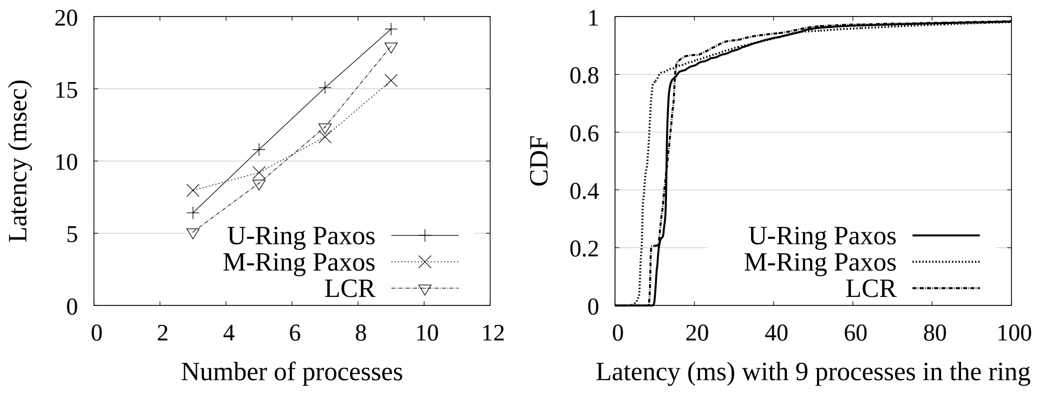


Figure 3.9. Impact of synchronous disk writes on the latency when varying the number of processes in the ring.

Ring Paxos and LCR, latencies vary according to the location of the proposer in the ring. The values reported for these two protocols are for the best-located proposer, that is, for the proposer with the lowest latency.

Latency in LCR and U-Ring Paxos increases with the number of processes; M-Ring Paxos presents a less-pronounced increase in the latency as more acceptors are placed in the ring (left graph in Figure 3.8). Notice that there is less information circulating in the ring of M-Ring Paxos than in the rings of LCR and U-Ring Paxos. In LCR and U-Ring Paxos, the content of each message is sent $n-1$ times, where n is the number of processes in the ring. Message content is propagated only once in M-Ring Paxos (using ip-multicast). LCR and U-Ring Paxos present similar latency in Figure 3.8, despite the expected difference, as presented in Table 3.1. The reason is that for a given setup with n processes, LCR can tolerate $f = n - 1$ failures, while U-Ring Paxos can tolerate $f = (n - 1)/2$ failures, for an odd n. Thus, for the same n, the number of communication steps for each protocol is, respectively, $2(n-1)$ and $2.5(n-1)$. In the experiments with S-Paxos we observed substantial variability in the results due to java's garbage collection mechanism, and average values for all experiments were above 35 ms.

3.5.5 Impact of disk writes

In this experiment we assess the performance of Ring Paxos protocols when processes store accepted values on disk and compare the results to LCR as it is the only protocol with a comparable performance. With synchronous disk writes all the techniques are essentially disk bound with a constant throughput

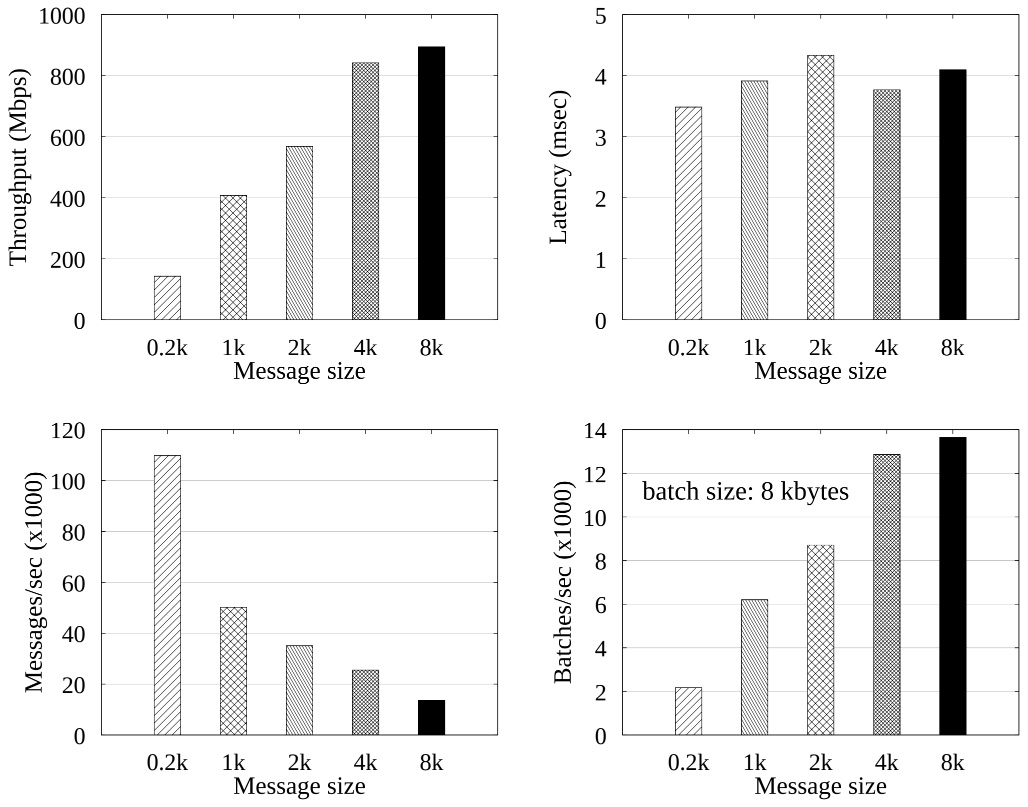


Figure 3.10. Impact of application message size on the performance of M-Ring Paxos.

of 270 Mbps, regardless the number of processes. However, as it is seen in Figure 3.9, latency increases as nodes are added to the ring. The right-most graph shows the CDF for latency when there are 9 processes in the ring. LCR and U-Ring Paxos have comparable latency. M-Ring Paxos has lower latency than LCR and U-Ring Paxos as processes write their values on disk in parallel; in LCR and U-Ring Paxos disk writes across processes happen sequentially. In all protocols, data is written on disk in units of 32 Kbytes.

3.5.6 Impact of message size

Figures 3.10 and 3.11 quantify the effects of application message size (payload) on the performance of M-Ring Paxos and U-Ring Paxos respectively. In both figures throughput (top left graphs) increases with the size of application mes-

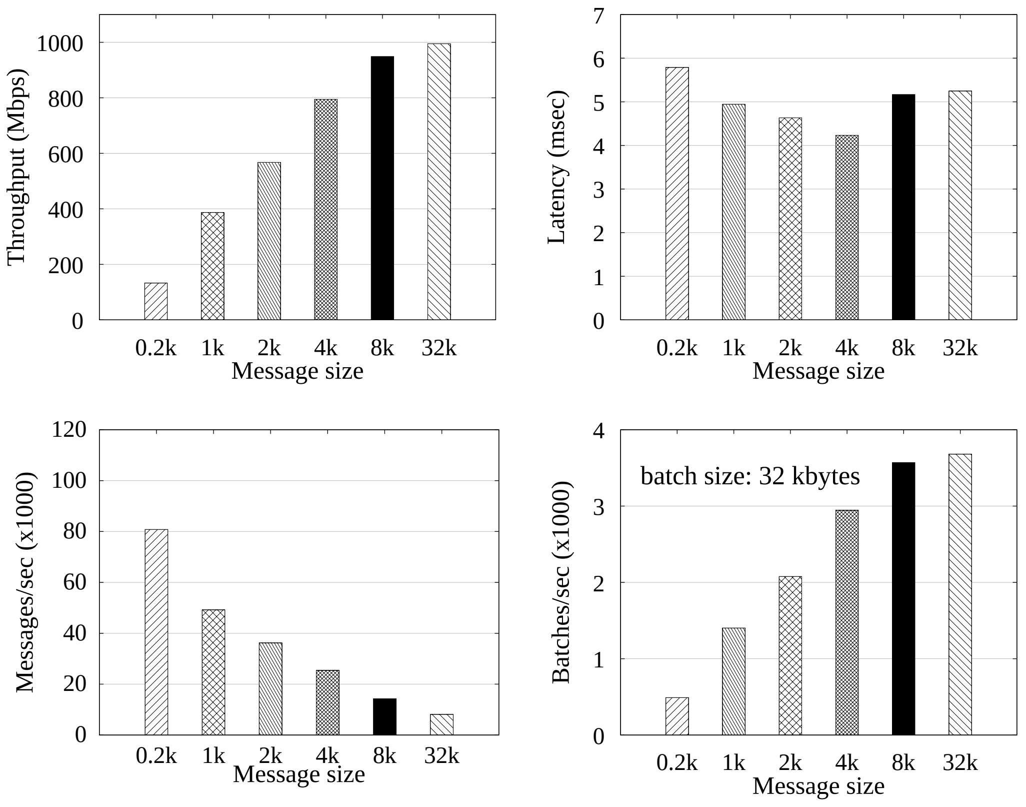


Figure 3.11. Impact of application message size on the performance of U-Ring Paxos.

sages, up to 8 Kbytes and 32 Kbytes in M-Ring Paxos and U-Ring Paxos, respectively. Notice that in our prototype ip-multicast packets are 8 Kbytes long, but datagrams are fragmented since the maximum transmission unit (MTU) in our network is 1500 bytes. In U-Ring Paxos communication is based on TCP. Latency is less sensitive to application message size (top right graphs). Figures 3.10 and 3.11 also show the number of application messages delivered as a function of their size (bottom left graphs). Many small application messages can fit in a single Paxos message and Phase 2 is executed for a batch of proposed values. As a consequence, many application messages can be delivered per time unit (left-most bars).

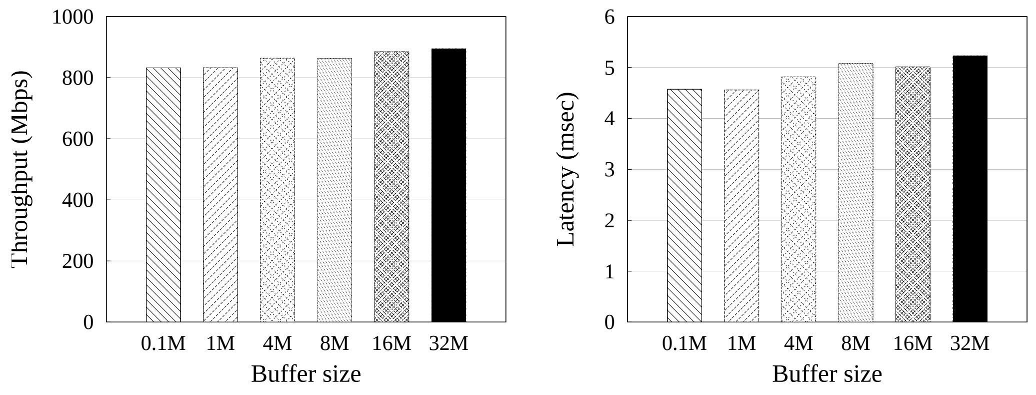


Figure 3.12. Impact of socket buffer size on the performance of M-Ring Paxos.

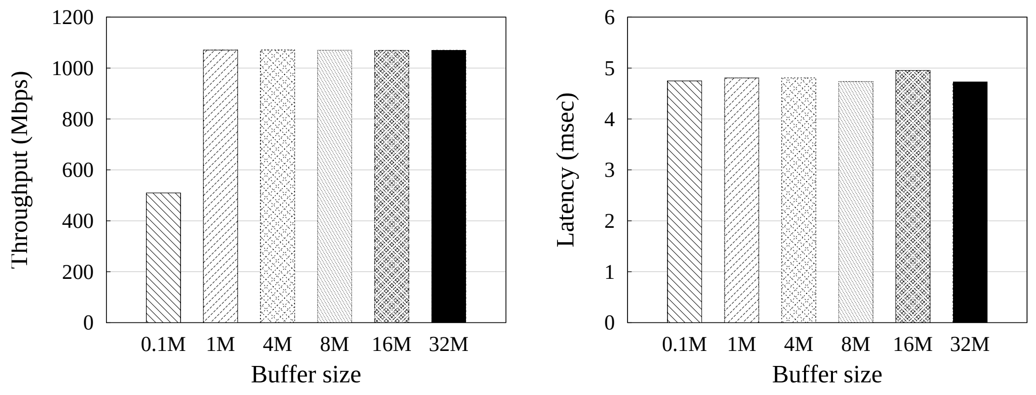


Figure 3.13. Impact of socket buffer size on the performance of U-Ring Paxos.

3.5.7 Impact of socket buffer size

Figures 3.12 and 3.13 show the effect of socket buffer sizes on the maximum throughput and latency of M-Ring Paxos and U-Ring Paxos, respectively. The reliability of unicast and ip-multicast depends on the size of the buffers allocated by the sockets. Lost messages have a negative impact on M-Ring Paxos, as they result in retransmissions. However, according to our experiments even with buffer sizes of 0.1M the attainable throughput is not far from the maximum throughput. In U-Ring Paxos we can achieve the maximum throughput with buffer sizes as big as 1M. In this protocol all the communications are based on TCP and with the buffers smaller than 1M due to the congestion protocol of TCP the maximum attainable throughput is lower. We have used socket buffer sizes of 16 Mbytes in M-Ring Paxos and 32 Mbytes in U-Ring Paxos in all the other experiments.

3.5.8 Flow control

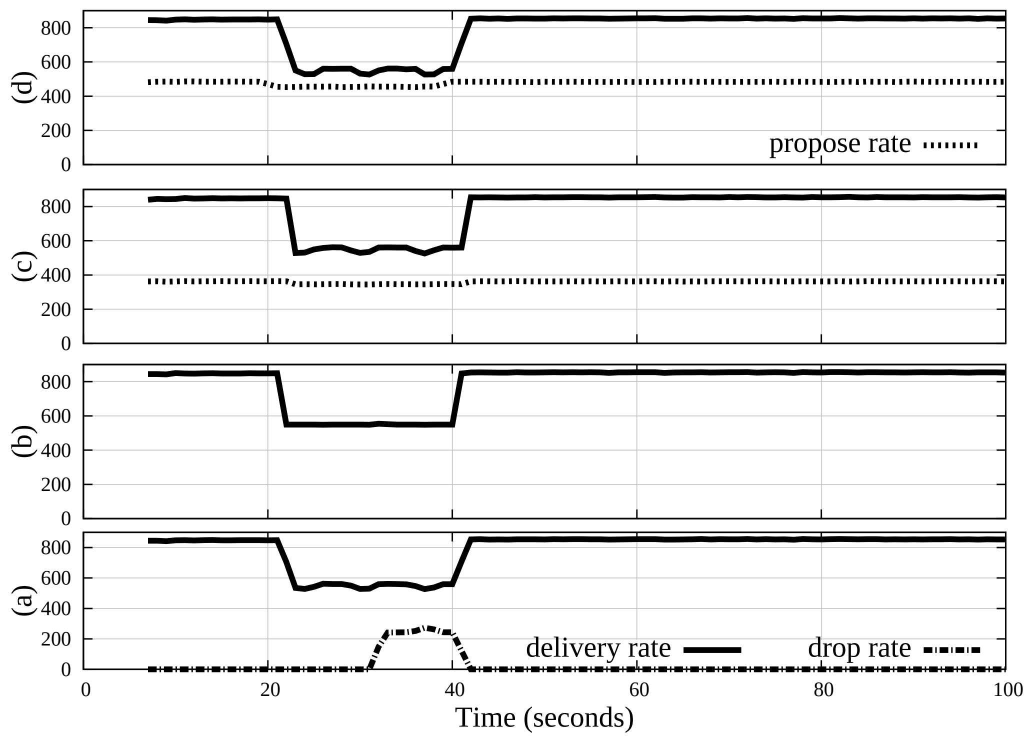


Figure 3.14. Flow control in M-Ring Paxos; in all the graphs the y-axis shows the throughput in Mbps for: (a) coordinator, (b) slow learner, (c) learner-proposer 1, (d) learner-proposer 2; the slow learner reduces its delivery speed after 20 seconds of execution and restores to its original rate after 40 seconds of execution.

Figure 3.14 illustrates the effect of the flow control mechanism in M-Ring Paxos with three learners among which two are also proposers. The aggregate proposing rate of proposers is 850 Mbps. After 20 seconds, one of the learners (graph (b)) reduces its speed in delivering Paxos instances. As soon as the number of pending instances reaches a predefined threshold, the slow learner sends a message to one of the acceptors in the ring to notify it about the situation. This notification is forwarded along the ring until it reaches the coordinator. Having received this message, the coordinator reduces its proposing rate. Since fewer

Table 3.3. CPU and memory use in M-Ring Paxos.

Role	CPU	Memory
Proposer	37.2%	90 Mbytes
Coordinator	88.0%	168 Mbytes
Acceptor	24.0%	168 Mbytes
Learner	21.3%	168 Mbytes

Table 3.4. CPU and memory use in U-Ring Paxos.

Role	CPU	Memory
proposer-acceptor-learner	48.0%	80 Mbytes

instances are proposed after receiving this message, the delivery rate also reduces. As proposers continue submitting requests at a constant rate, eventually the receiving buffer of the coordinator overflows and new requests are dropped. Proposers submit new requests and re-submit pending requests. They only reduce their proposing rate if they detect buffer overflows. After 40 seconds, the slow learner restores its original delivery rate. Since the coordinator does not receive new slow-down requests it also restores its original proposing rate.

3.5.9 CPU and memory usage

Table 3.3 shows the CPU and memory usage of M-Ring Paxos under maximum throughput. For this experiment we isolated the processes running M-Ring Paxos in a single node and measured their CPU and memory usage. Not surprisingly, the coordinator is the process with the maximum load since it should both receive a large stream of values from the proposers and ip-multicast these values.

Table 3.4 shows the results for U-Ring Paxos. In this case, all the processes play the roles of proposer, acceptor, and learner. Therefore, the CPU and memory usage of all of them is similar. In both tables memory consumption at coordinator, acceptors, and learners is mostly used by the circular buffer of proposed values. For efficiency, in our prototype the buffer is statically allocated.[7]

3.5.10 Conclusions from the experiments

We infer the following main points from our experiments:

[7]As a reference, the average CPU usage per process in LCR is in the range of 65%–70% and for S-Paxos it is about 270% (i.e., S-Paxos is multithreaded).

- By using ip-multicast in M-Ring Paxos and a topology entirely based on a ring, as in U-Ring Paxos and LCR, we could achieve throughput near the limits of the network (Section 3.5.3).
- Latency increases with the size of the ring in the Ring Paxos protocols, although M-Ring Paxos is less prone to the effects of ring size on latency (Section 3.5.4).
- Both M-Ring Paxos and U-Ring Paxos are affected by message size. M-Ring Paxos achieves high throughput with messages of 4 Kbytes or bigger; U-Ring Paxos reaches maximum performance with 8-Kbyte messages. If application messages are small, batching can improve performance (Section 3.5.6).
- In-memory deployments of M-Ring Paxos and U-Ring Paxos are network-bound; the performance of disk-based deployments is determined by the capacity of the storage device (Section 3.5.5).
- Our simple flow control mechanism in M-Ring Paxos proved effective in avoiding message losses by slowing down the rate of coordinator (Section 3.5.8).

3.6 Conclusion

In this chapter we presented Ring Paxos, composed of two atomic broadcast protocols specifically designed for achieving high throughput in local-area networks. In their design, we have embedded proper communication patterns to optimize various types of message passing among processes. We have implemented both protocols and compared them to a variety of other atomic broadcast protocols.

Our experimental results showed that both protocols have a constant throughput with respect to the number of receivers in the network, which is a desired property in clustered environments. In the absence of the ip-multicast primitive (e.g., in a data center) we suggest U-Ring Paxos over M-Ring Paxos. Furthermore, most of our findings in this chapter (specifically about communication mechanisms) can be applied to the design of other distributed systems with similar requirements.

Chapter 4

Speculation and State Partitioning in State-Machine Replication

State-machine replication improves availability, but often degrades performance in contrast to a stand-alone deployment. In this chapter we study performance limitations of state-machine replication and propose novel solutions to overcome them. First, we note that ordering the requests in state-machine replication increases the response time experienced by the clients. To alleviate the impact of ordering on latency, we build speculative replicas. Our speculative replicas parallelize the execution of requests with the ordering in the agreement layer, M-Ring Paxos, to reduce the overhead on latency. Second, state-machine replication requires all the replicas to execute all the requests. Thus performance can not be improved by adding new replicas. To scale performance with the number of replicas we partition the state and modify M-Ring Paxos accordingly. With these two techniques, speculation and state partitioning, in addition to having higher availability, a service replicated by the state-machine approach can also achieve a higher performance.

4.1 Problem statement

State-machine replication, a technique to improve availability, requires all the replicas to deterministically execute all the requests in an identical order to always remain consistent.[1] From a performance perspective, state-machine replication suffers from two shortcomings. First, it imposes extra overhead on the response time when compared to a stand-alone service. The increased response

[1]The type of consistency we are interested in is *linearizability* (see Chapter 2).

time stems from the need to order client commands before they can be executed, whereas in a stand-alone setup commands are directly sent to the servers for execution. In Chapter 3, we mitigated the overhead of the agreement layer by designing the Ring Paxos protocols. Although Ring Paxos is highly optimized, latency remains affected by its mere existence. Second, the overall performance is limited by the throughput of a single replica. If demand augments (e.g., more clients join the system) it cannot be absorbed by adding replicas to the system. The throughput limitation is a consequence of each replica storing a full copy of the service state and executing *all* the commands.

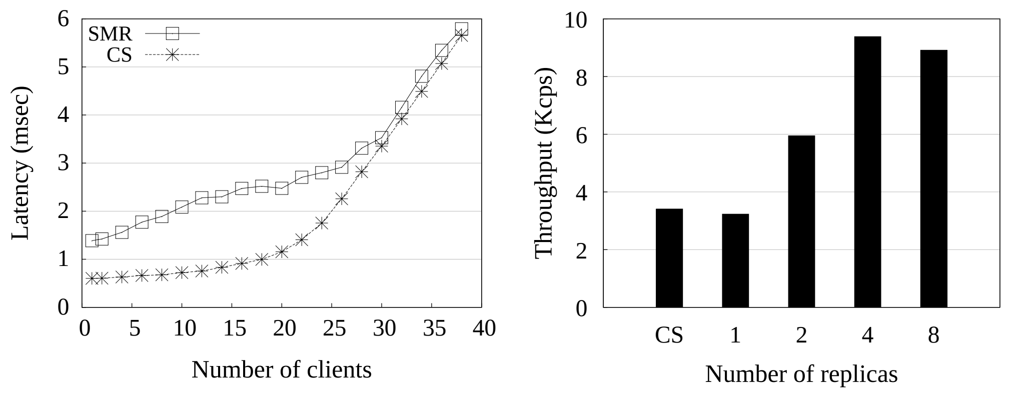


Figure 4.1. Client-server (CS) versus state-machine replication (SMR) with read-only commands; right graph shows throughput measured in Kilo commands per second (Kcps); left graph shows latency measured at clients (for SMR the values correspond to the throughput of one replica).

To investigate the negative impacts of these two items, ordering and full replication, we performed an experiment with a replicated B^+-tree service, where the workload is composed of read-only commands.[2] The left graph of Figure 4.1 shows the response time of the two systems, replicated and non-replicated, as the number of clients increases. The difference between the two curves indicates the overhead introduced by replication. The right graph of Figure 4.1 shows the scalability of the throughput with the number of replicas. Since the workload is composed of read operations only, not all the replicas have to execute all the requests. This is a workload one would expect an ideal scalability with. However, as our experiments show, replication improves the throughput up to four replicas, but after that the overhead of delivering and discarding read commands prevents the performance from scaling further.

[2]More details about these experiments are available in Section 4.4.

To conclude, state-machine replication requires commands to be ordered, and ordering commands is inherently more expensive than directly sending them to a server. Moreover, the fact that all replicas must deliver all commands—although not all commands must be executed by all replicas—limits the attainable performance (see [35] for a similar argument). In this Chapter, we apply two mechanisms, speculation and state partitioning, in the context of M-Ring Paxos protocol from Chapter 3 to overcome these overheads.

4.1.1 Outline

The rest of this chapter is structured as follows. In Section 4.2, we discuss speculative delivery and state partitioning as two well-known approaches to overcoming the inherent performance limitations of state-machine replication. Then in Section 4.3 we relate to previous works on speculation and state-partitioning. In Section 4.4 we extensively evaluate our prototypes and explain the main results. In Section 4.5 we conclude the chapter.

4.2 Overcoming limitations of state-machine replication

Our approach to improving the performance of state-machine replication is to address the overhead of ordering and full replication separately: we show how to reduce the response time and how to increase the throughput of a replicated system with speculation and state partitioning respectively. As for the ordering of client requests we will use M-Ring Paxos protocol (see Chapter 3 for more details).

Response time. To reduce the impact of agreement layer on response time we use speculative (or optimistic) execution on the replicas, a technique that has been used before in the context of replicated databases (e.g., [36; 37]). The idea is to expose servers to a command before its final order has been established. As a result, the execution of the command by the server and the ordering of the command in the agreement layer overlaps in time, improving response time. The technique is speculative because it only works if the order in which commands are executed is confirmed by the ordering protocol. If not, the commands must be rolled back and re-executed in the correct order (i.e., the order defined by the ordering protocol). In Section 4.2.1 we explain the combination of this technique with M-Ring Paxos.

Throughout. We address the throughput limitation of state-machine replication with state partitioning. In brief, we allow applications to decompose their state into sub-states and replicate each sub-state individually. Commands are directed to and executed by the appropriate partitions only. By partitioning the state of a service, we allow to process commands in parallel. This is particularly effective for services whose state partitioning is *perfect*, that is, all commands access one sub-state or another, but no command accesses two or more sub-states. Commands that access more than one sub-state must be carefully ordered to avoid inconsistencies. In Section 4.2.2 we will discuss how to efficiently integrate state partitioning with M-Ring Paxos.

4.2.1 Speculative execution

The response time experienced by a client of a replicated service is the result of the aggregated overhead of the following four steps:

(a) Proposing the command by the client.

(b) Ordering the command in the agreement layer.

(c) Executing the command at the servers.

(d) Transmitting the response to the client.

A reduction in the duration of any of these activities will likely decrease response time. In the context of M-Ring Paxos this is not trivial since the protocol is highly optimized and it seems unlikely that it can be significantly improved to accommodate high throughput and lower response time. Moreover, the delay incurred by the execution of a command and the transmission of its response is mostly service specific.

We resort to a speculative (or optimistic) strategy which consists in overlapping part of the ordering protocol with the execution of commands. In M-Ring Paxos, a command reaches the servers before its ordering information (see Chapter 3, Section 3.3.2). When a command arrives, it is buffered by the server and only executed once its order is known. We propose to execute the command immediately after it is received, avoiding any buffering. In doing so, servers can start processing the command before its order is confirmed, saving some time. A server can only respond to a client after it has executed the command and the order of the command is confirmed. The mechanism is speculative because it works as long as the order in which commands arrive at the servers (and thus the order in which they are executed) match. In rare occasions (discussed below) commands may be executed out-of-order. If the order in which one or more commands were executed is not confirmed, the server must *rollback* them and re-execute the commands in the proper order. Rolling back a command is service-specific and can be done physically (e.g., by using an undo log) or logically (e.g., by executing an action that reverses the effects of the out-of-order command) [38]. We briefly discuss logical rollback in Section 4.4.2 for a simple B^+-tree service.

Fortunately, in M-Ring Paxos the order assigned by the coordinator when a command is ip-multicast is always confirmed by the acceptors. The only situation in which the execution order of a command may change is when the coordinator is replaced by another process (e.g., due to a crash), a rare event. Lost messages do not cause commands to be executed out-of-order since each command (or batch of commands) contains a consensus instance number, which allows a server to detect missing commands.

We can estimate the improvements expected from speculative execution with a simple formulation. Let δ be the time it takes for a client to send a command to the coordinator and for a server to respond to the client with its results. Assume further that Δ_o is the time needed to order the command (in M-Ring Paxos this means the time difference between the first and the second ip-multicast related to the command) and Δ_e is the time needed to execute the command. Without speculative execution, the response time expected by a client in the absence of contention is $2\delta + \Delta_o + \Delta_e$. With speculative execution, it depends on the values of Δ_o and Δ_e: if $\Delta_o < \Delta_e$ then response time is $2\delta + \Delta_e$; otherwise response time is $2\delta + \Delta_o$. Thus, we can expect an improvement of the order of $min(\Delta_o, \Delta_e)$.

4.2.2 State partitioning

A service implemented by means of state-machine replication has limited or no scalability at all, as a consequence of replicas storing the full service state, and receiving and handling all client commands. To make the system scalable, we must partition the service's state into "sub-states". If the partitioning is *perfect*, that is, all commands access one sub-state or another, but no command accesses two or more sub-states, then the technique can be trivially implemented: it suffices to replicate each partition individually, using different and independent instances of M-Ring Paxos, and submit client commands to the appropriate partition.

Some services, however, may not allow perfect partitioning. This is the case when a service's state is partitioned into sub-states such that some of the commands access more than one partition. We illustrate this case with an example.

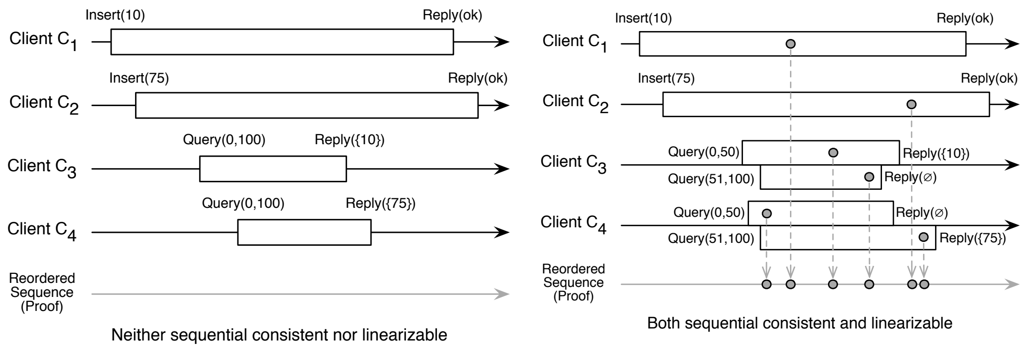


Figure 4.2. (Left) A non-linearizable execution that cannot happen if a B$^+$-tree is replicated with state-machine replication. (Right) How the same execution can happen if sub-trees of the B$^+$-tree are replicated independently.

Consider a B$^+$-tree service with `insert` and `query` commands (see Section 4.4.2 for more details). We can partition the B$^+$-tree into sub-trees by assigning to each sub-tree a non-overlapping key interval and replicate each sub-tree using state-machine replication. An `insert` command is directed to a single replicated sub-tree. A `query` command that requests a set of keys within a certain range may be addressed to a single sub-tree or to multiple sub-trees, depending on the range and the key intervals assigned to each sub-tree. If the `query` command addresses multiple sub-trees, then it is divided into "sub-commands", one for each sub-tree; the client builds the final response from the results received from each sub-tree. Such a service, however, cannot be implemented by independent instances of M-Ring Paxos, as we now explain.

To understand the reason, consider the execution on the left of Figure 4.2. Under linearizability, this execution cannot happen since client C_3 sees C_1's insert before C_2's, and C_4 sees C_2's insert before C_1's. If we partition the B$^+$-tree into two independent sub-trees, however, as in the execution on the right of Figure 4.2, then clients may observe a non-linearizable behavior. In this execution, C_3's and C_4's Query$(0, 100)$ command is composed of two subcommands, Query$(0, 50)$ and Query$(51, 100)$. The problem is that while C_3's Query$(0, 50)$ succeeds C_4's Query$(0, 50)$ in one partition, C_3's Query$(51, 100)$ precedes C_4's Query$(51, 100)$ in the other partition, and thus, C_3's Query$(0, 100)$ neither precedes nor succeeds C_4's Query$(0, 100)$. To ensure linearizability we must be able to establish a total order on all commands, not only on sub-commands. Notice that this happens in spite of the fact that the execution of each sub-tree is individually linearizable.

We now define *state partitioning ordering*, a guarantee needed to ensure that an execution with commands involving multiple service partitions is linearizable. Let a service state be decomposed into partitions $P_1, ..., P_k$, each one replicated and implemented as a series of consensus executions—the i-th consensus instance decides on the i-th sub-command of partition P_k. Let command C_x be composed of sub-commands $\{c_{x,i} \mid c_{x,i}$ is a subcommand of C_x in $P_i\}$. We define directed graph $G = (V, E)$ such that V contains all commands C_x in the execution and E contains directed edges $C_x \rightarrow C_y$ such that $c_{x,i}$ precedes $c_{y,i}$ in P_i. State partitioning ordering requires that G be acyclic.

A consequence of G being acyclic is that it can be topologically ordered, and therefore for any two commands C_x and C_y, if $c_{x,i}$ precedes $c_{y,i}$ in partition P_i, then in no partition P_j, $c_{y,j}$ precedes $c_{x,j}$, where $c_{x,i}, c_{x,j} \in C_x$ and $c_{y,i}, c_{y,j} \in C_y$. We state the property as an acyclic graph of commands to cover more complex cases involving relations between more than two commands (see [39] for an example).

State partitioning with M-Ring Paxos. We have integrated state partitioning order into M-Ring Paxos as follows. First, there is one ip-multicast address associated with each partition and one ip-multicast address associated with decisions. Differently than M-Ring Paxos, we do not piggyback decision messages with commands. Learners (i.e., replicas) listen on the partition addresses they are interested in and on the decision address. Acceptors listen on all addresses. A command contains information about the partitions it accesses. For each partition accessed by the command, the coordinator ip-multicasts one Phase 2A message (with the command) using the address associated with the partition. If a process receives the same message more than once, it simply discards the duplicates. When order is established, the coordinator ip-multicasts the decision message using the decision address. Learners may receive decision messages for partitions they are not interested in, in which case they discard the messages.

To conclude, the state partitioning technique improves the scalability of state-machine replication but it may not be applicable in some cases or it may impose restrictions on how the state of a service can be partitioned. Consider a service whose state contains variables x and y, and a command that modifies x based on the value of y. In this case, the service's state can only be partitioned such that both x and y belong to the same partition. While this constraint limits the number of services that can benefit from state partitioning, we show later in Section 4.4 that the technique is general enough to allow the implementation of a high performance fault-tolerant B^+-tree service.[3]

4.3 Related work

In this section we overview related work on speculative execution and state partitioning. Since we evaluate our prototypes with a B^+-tree we also review literature on parallel B-tree implementations.

Speculative execution. Optimistic or speculative execution has been suggested before as a mechanism to reduce the latency of agreement problem. For example, in [40; 41], clients are included in the execution of the protocol to reduce the latency of Byzantine fault-tolerant agreement. In [36; 37] the authors introduce atomic broadcast with optimistic delivery in the context of replicated databases. The motivation is similar to ours: overlapping the execution of transactions or commands with the ordering protocol. Optimistic delivery relies on spontaneous ordering of messages, typical in local-area networks. The property holds in the absence of contention. If too many commands are submitted simultaneously, then out-of-order deliveries can happen more frequently and the technique becomes less interesting. M-Ring Paxos can use speculative execution under high contention as it does not depend on spontaneous message ordering.

State Partitioning. Partitioning the state of a replicated service is conceptually similar to partial replication of databases [39]. Partial database replication addresses scalability issues identified in fully replicated databases. Several partial database replication protocols have been proposed, some optimized for local-area networks (e.g., [42; 43; 44; 45]) and some topology-agnostic (e.g., [46; 47; 48; 49]). Partitioning the state of a replicated service differs from partially replicating a database with respect to the granularity of the data and the consistency criterion. Databases are usually organized as collections of data items. Partitioning such a state is simpler than partitioning the state of a service, which may not have been designed with partitioning as a goal. With respect to consistency, the two main consistency criteria used in replicated databases are one-copy serializability [50] and a generalized form of snapshot isolation [51; 52]. These criteria do not take real-time dependencies between operations into account and therefore admit more efficient implementations than linearizability. M-Ring Paxos equipped to implement the state partitioning technique resembles an atomic multicast protocol [53]. In fact, our state

[3]For a proof of correctness see Appendix.

partitioning ordering is inspired by the acyclic order property of atomic multicast [39]. To the best of our knowledge, however, no previous work has explored multicast communication in the Paxos family of protocols, and no speculative or optimistic multicast protocol has been proposed so far.

Parallel B-tree. The closest work to our B$^+$-tree service is [54], where the authors implement and evaluate a distributed B$^+$-tree built on top of Sinfonia [55]. Sinfonia is a distributed, fault-tolerant storage engine that offers a low-level address space in which application processes can store their data. Sinfonia offers a minitransaction interface to its clients. Minitransactions are short-lived operations similar to a generalized compare-and-swap operation. The authors exploit the flexibility offered by Sinfonia to implement a scalable B+Tree. As an optimization, inner nodes are replicated on all Sinfonia client nodes. On the one hand this allows nodes to traverse a tree locally, without contacting any other node; on the other hand, all nodes must be involved in the update of inner nodes. Sinfonia relies on stronger system assumptions than the ones assumed in this chapter. This is due to the use of a two-phase commit protocol to terminate minitransactions.

4.4 Experimental evaluation

In this section, we first review some details about our implementations and experimental setup and then evaluate our prototypes with respect to the following aspects:

1. **The cost of replication.** Although replication strengthens a system's fault tolerance, it often negatively affects the performance. The goal of this experiment is to measure this effect on latency and throughput by comparing a replicated service against a single-copy deployment (Section 4.4.3).

2. **Speculative execution.** Latency of a replicated service is higher than a non-replicated service due to the additional tasks that are performed before a request can be executed. This experiment is performed to see whether speculative execution can bring the latency of the replicated service closer to that of a single-copy deployment (Section 4.4.4).

3. **State partitioning.** In a fully replicated service, except for certain workloads, all the replicas receive and execute all the requests. Therefore,

adding more replicas does not increase throughput. We perform this experiment to see if partitioning the state can help to scale the throughput when more replicas are added (Section 4.4.5).

4. **Speculative execution and state partitioning.** We perform this experiment to study the combined effect of speculation and partitioning on the latency and throughput of a replicated service (Section 4.4.6).

4.4.1 Hardware settings

All the experiments are performed in a cluster of Dell SC1435 servers equipped with 2 dual-core AMD-Opteron 2.0 GHz CPUs and 4GB of main memory. The servers are interconnected through an HP ProCurve2900-48G Gigabit switch whereas the round trip time is 0.1 mili second. In all the experiments clients and servers are deployed on separate machines.

4.4.2 Implementation and experimental setup

In our prototypes the speculative server is composed of four active threads, whose tasks can be described as follows: (1) delivering the commands and their order decided by M-Ring Paxos, (2) tracking the commands once their order is decided, (3) processing the commands, and (4) sending responses to the clients after the commands are successfully processed and their order is known. Thus once the first thread receives a command, puts it in a shared buffer from which the third thread will later remove the command and process it. This implies that each command spends some time waiting in the buffer until the third thread is free to process it. Therefore, in practice due to the implementation overheads a request is not processed immediately after its arrival. A non-speculative server is implemented by three threads: a thread receives commands, another thread executes commands, and a third thread responds to clients. In all of our experiments each thread is assigned to a different processor.

We evaluate our prototypes with an open source implementation of B^+-tree [56]. The in-memory B^+-tree stores `(key, value)` tuples, where keys and values are 8-byte integers. Three types of operations are defined on the tree: `insert(key, value)`, `delete(key)`, and `query(key_min, key_max)`. An `insert` operation inserts a new tuple in the tree if it does not already exist. A `delete` operation deletes a tuple from the tree if it exists in the tree. A `query` searches the tree to retrieve the values for the range of the keys specified in the command (hereafter, we refer to `insert` and `delete` operations as `updates`).

`Update` operations return a small reply to the clients about the result of the operation and a range `query` returns the set of the values extracted from the tree. Thus, the size of the replies for range queries is often bigger than the size of replies for update operations. We measure performance with the following three workloads:

1. `Queries`: in this workload, a command issued by clients is a query for a range over an interval of 1000 keys, where the keys are chosen randomly following a uniform distribution.

2. `Ins/Del (single)`: in this workload, each command includes one update operation, either an insert or a delete.

3. `Ins/Del (batch)`: in this workload, each command includes seven updates. In addition, the coordinator of M-Ring Paxos batches the requests into packets of 8 Kbytes (in both `Ins/Del (batch)` and `Ins/Del (single)` workloads, the portion of delete and insert commands are such that the size of the trees does not change over the time).

In the experiments with full replication, the tree on all the replicas is initialized with 12 million keys in the range of `[1, 12M]`. To preserve the size of the tree, in the experiments with partial replication we have a bigger range of keys: `[1,12M * num_partitions]`, whereas a tree in each partition is initialized with 12 million distinct keys. Thus in both full and partial replication the tree on each replica is initially populated with `12 million (key, value)` tuples. A command that accesses more than one partition is broken into sub-commands by the client (i.e., by a client replication library) and submitted to each concerned partition. Responses received from multiple partitions are merged at the client. In the fully replicated B$^+$-tree, all the replicas receive all the operations. All the replicas execute `insert` and `delete` commands on their local tree but only one replica returns the result to the client. For range queries, only one replica executes the command and responds to the client. In the experiments assessing speculation, in case of mismatches between the receive and delivery order, operations must be rolled back. As range queries do not change the state of the trees there is no need to roll them back. But to roll back an insert, a delete command should be executed. For rolling back a delete, an insert is performed given that the value of the deleted key should be kept until the final ordering is known.

The size of client commands is 256 bytes in all the workloads. Responses are 8 Kbytes for ranges and 256 bytes for inserts and deletes. In addition, our experiments are performed in the absence of process failures, but message losses

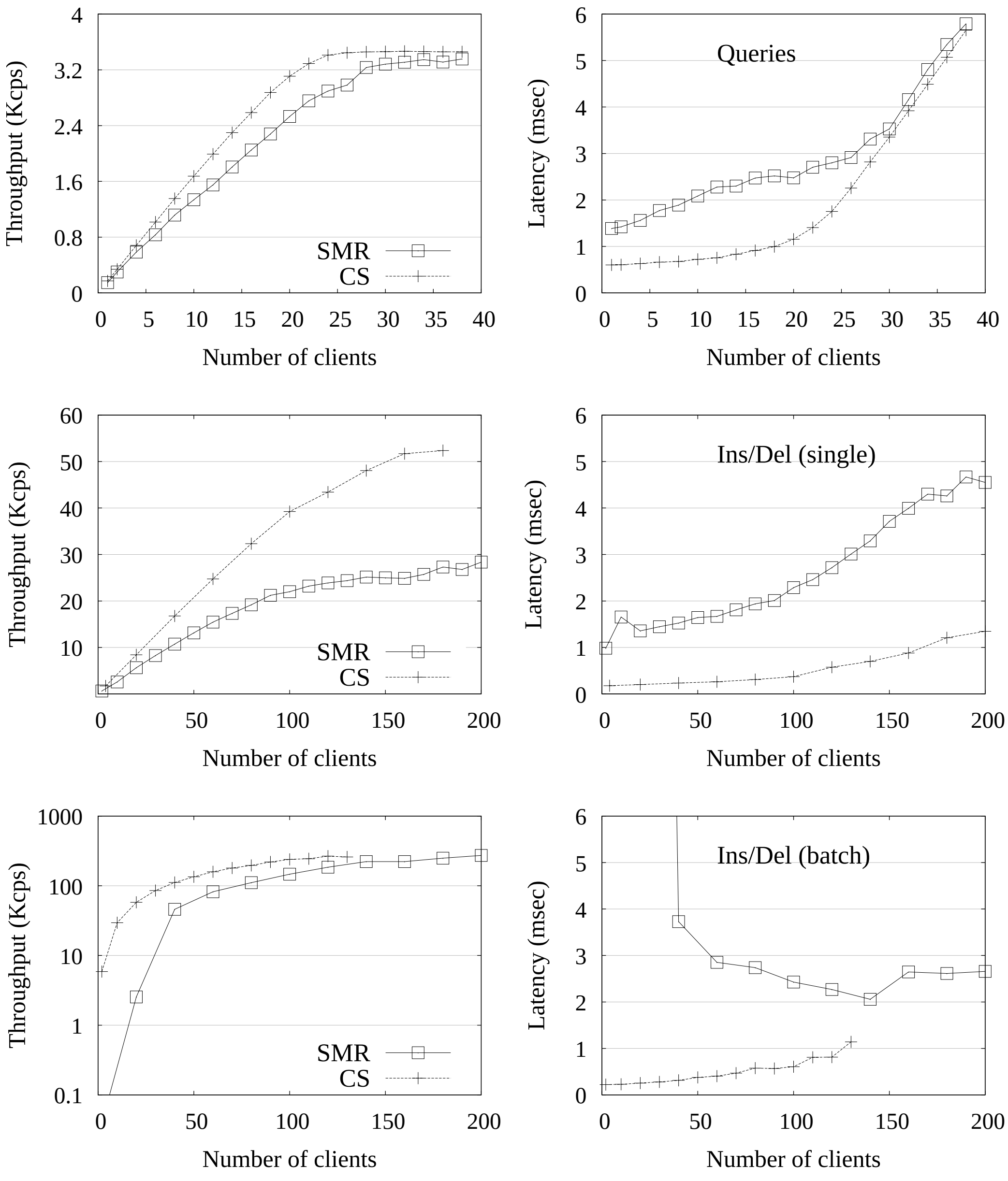


Figure 4.3. Performance of client-server (CS) versus state-machine replication (SMR) with three workloads; the pair of graphs on top are for queries, the ones in the middle are for update operations without batching, and the graphs in the bottom are for the batched update operations; left graphs show throughput measured in Kilo commands per second (Kcps) (the values in y axis of the bottom-most graph are in log scale); right graphs show corresponding latency measured at clients.

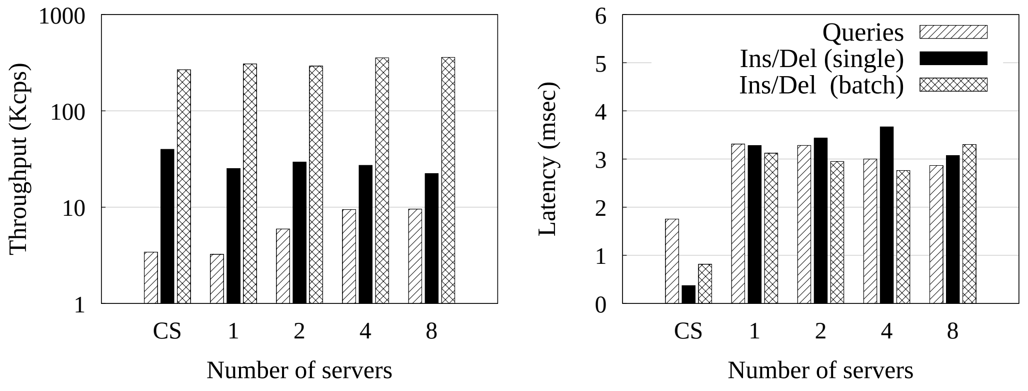


Figure 4.4. Performance of client-server (CS) versus state-machine replication (SMR) with increasing number of replicas for three workloads; left graph shows throughput measured in Kilo commands per second (Kcps) (y-axis is in log scale); right graph shows the corresponding latency.

are possible. Process failures are rare events, however, message losses happen relatively often because of high network traffic. In all the graphs each point is obtained over a 60-second run of which the first and the last 10 seconds are discarded.

4.4.3 The cost of replication

Our first set of experiments evaluates the cost of state-machine replication (SMR) with respect to a non-replicated client-server (CS) setup (see Figures 4.3 and 4.4). For queries and batched updates, replication does not introduce a cost in throughput. In these cases, the executions are CPU-bound. For single updates, the replicated setting cannot reach the same throughput as a client-server configuration because the execution of the former is limited by the maximum number of instances per second that can be run by M-Ring Paxos. In all cases, however, replication imposes a cost in response time, as shown by the graphs in the right column of Figure 4.3. Latency with fewer number of clients is high for SMR with batched updates (bottom-right graph in Figure 4.3) because in lower loads the transmission of M-Ring Paxos packets is triggered by timeouts; the effect disappears as clients are added and messages are sent as soon as an 8-Kbyte packet is full.

Adding replicas can help improve the throughput of read-only commands. This is evident by looking at the values of the left-most bars in the left graph of Figure 4.4. For update commands, however, no improvement in throughput

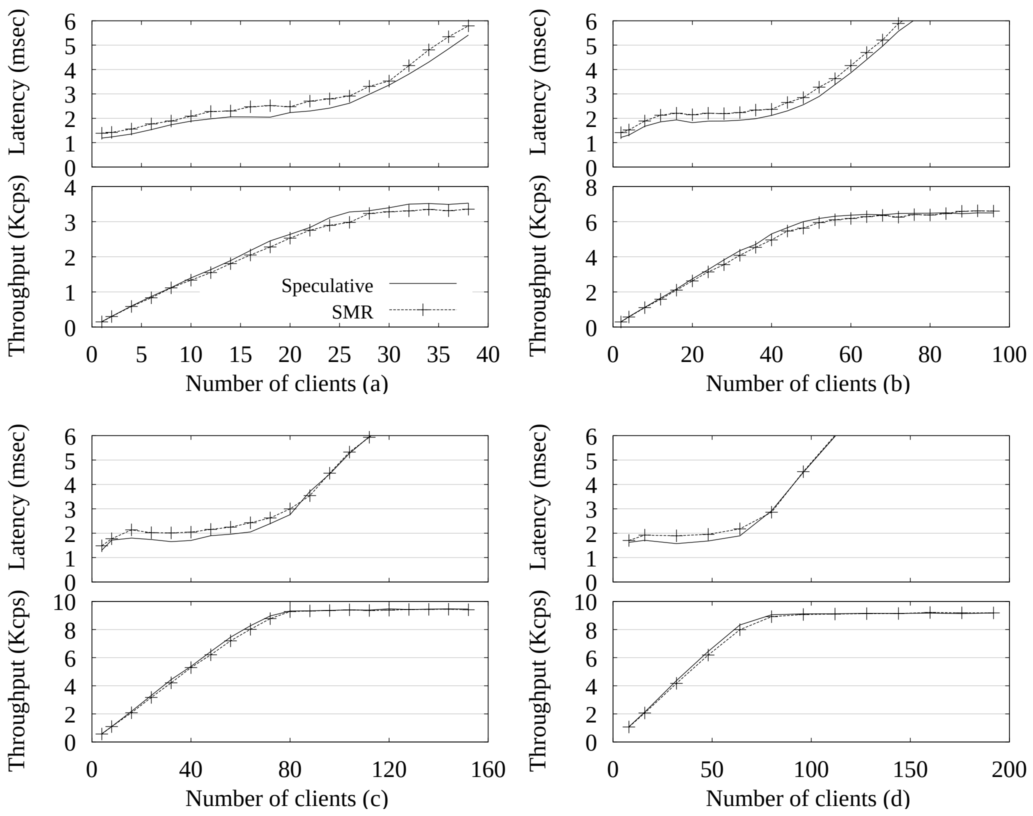


Figure 4.5. The impact of speculative execution on the performance of state-machine replication with a workload composed of queries only; number of replicas is (a): 1, (b): 2, (c): 4, (d): 8; throughput is measured in Kilo commands per second (Kcps).

is possible since all replicas must be involved in the operations, even if only to receive the commands in the right order, as discussed in Section 4.1. Figure 4.4 also shows the corresponding latency.

4.4.4 Speculative execution

We report our assessment of speculative execution for configurations with 1, 2, 4, and 8 servers using the `Queries` and the `Ins/Del (batch)` workloads (see Figures 4.5 and 4.6). In all scenarios speculation reduces response time with respect to state-machine replication, although the results are more visible with the `Ins/Del (batch)` workload. By reducing response time, the technique also proportionally improves throughput, a direct consequence of Little's law [57].

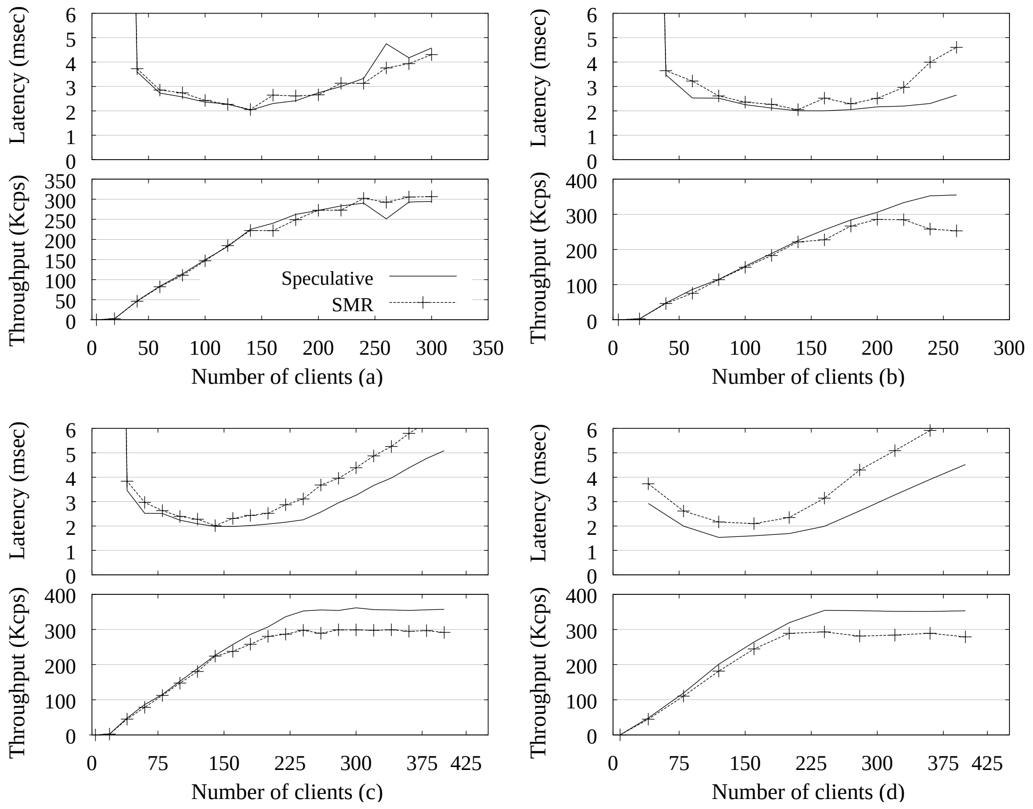
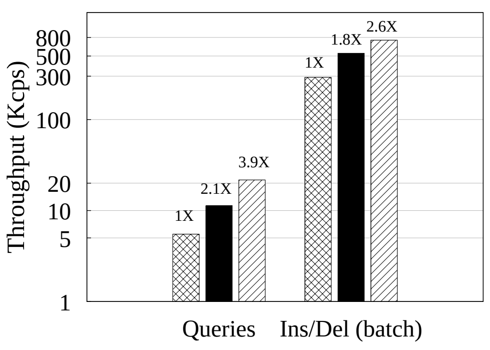


Figure 4.6. The impact of speculative execution on the performance of state-machine replication with a workload composed of batched inserts and deletes; number of replicas is (a): 1, (b): 2, (c): 4, (d): 8; throughput is measured in Kilo commands per second (Kcps).

4.4.5 State partitioning

To assess the state partitioning strategy, we consider two configurations, one with the B$^+$-tree state divided into two partitions and the other with the B$^+$-tree state divided into four partitions (labels “2 P” and “4 P”, respectively, in Figure 4.7). In both configurations each partition has two replicas. In executions with cross-partition query commands (Figures 4.8 and 4.9), a cross-partition query accesses two partitions, regardless the number of existing partitions.

The graph on the left of Figure 4.7 shows that for queries, the throughput increases by a factor of 2.1 from SMR to two partitions, and by a factor of nearly four from SMR to four partitions. The improvement with the `Ins/Del (batch)` workload is not as remarkable as on queries, although the system throughput increases by factors of 1.8 and 2.6 for two and four partitions, respectively. The



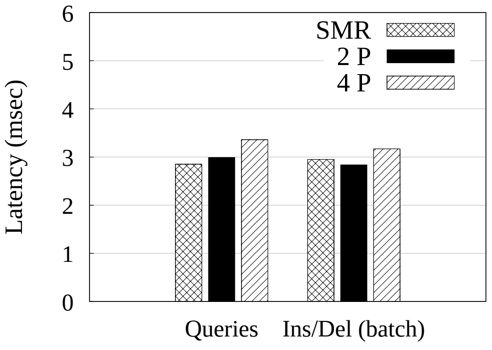


Figure 4.7. Performance of state partitioning (2 and 4 partitions) versus state-machine replication for queries and batched updates with no cross-partition commands; left graph shows throughput measured in Kilo commands per second (Kcps) (y-axis is in log scale); numbers in this graph show the speedup over SMR; right graph shows corresponding latency.

graph on the right shows that such an increase in throughput does not incur in significant changes in response time with respect to SMR. Although these experiments were run using no cross-partition queries, as we show next, this is not the most favorable setup for state partitioning.

Figure 4.8 demonstrates the effects of cross-partition queries in the state partitioning technique with two partitions in an execution with query commands whereas there are 2 replicas in each partition. The graphs show that for lower load (i.e., 100 clients) there is almost no difference in throughput and response time between different configurations. For higher loads, configurations with 50% and 75% of cross-partition queries reach higher throughputs. In fact, the lowest throughput and highest response time is obtained with a configuration without cross-partition queries. To understand the reason, we must look at how CPU is used in a server. The bottom-right graph in Figure 4.8 shows the CPU usage for threads responsible for execution and responses. The thread that receives commands has low use. While in configurations with no cross-partition queries, 98% of the processor is used for command execution, in configurations with 25% and 100% of cross-partition queries, the processor for command execution and response is 95% used. Finally, in configurations with 50% and 75% of cross-partition queries, the processors are used less than 90%. The 50% configuration has slightly higher throughput than the 75% configuration because it uses less bandwidth.

The reason for the execution processor-use to decrease with the increase in

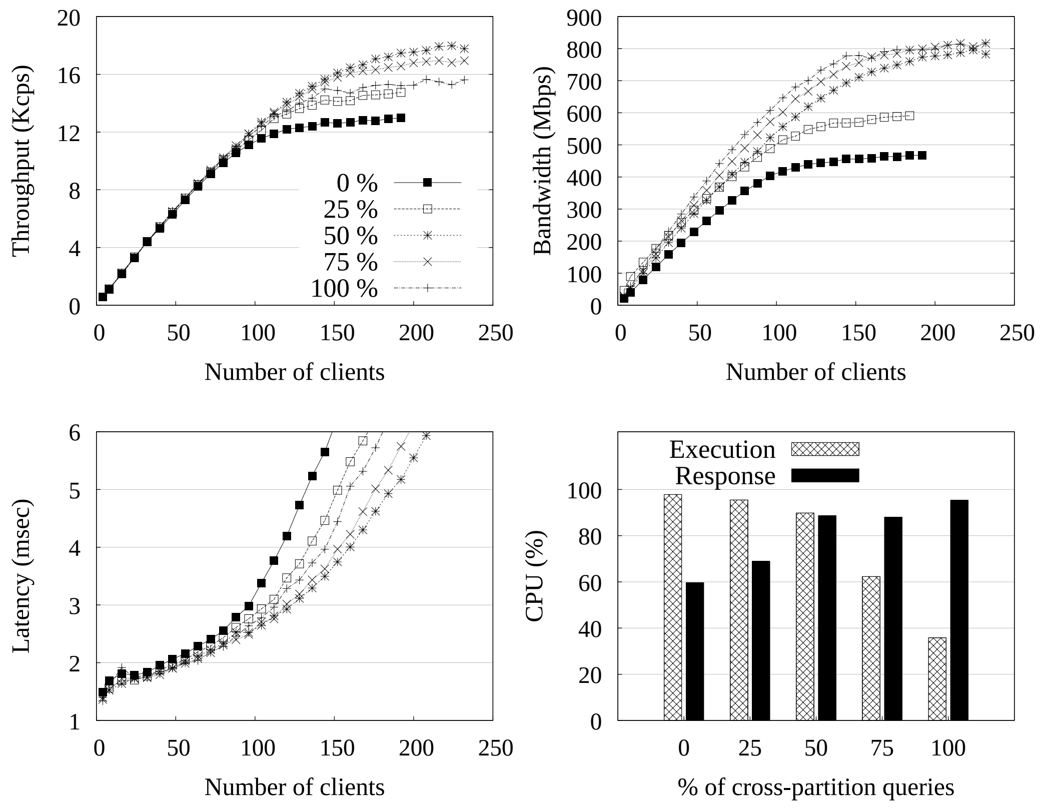


Figure 4.8. Impact of cross-partition queries on the performance with 2 replicas in each partition; top-left graph shows throughput measured in Kilo commands per second (Kcps); top-right graph shows outgoing bandwidth per replica measured in Mega bits per second (Mbps); bottom-left graph shows the corresponding latency; bottom-right graph shows the corresponding CPU utilization.

the number of cross-partition queries is that a cross-partition query is "cheaper" to execute than a single-partition query since it processes fewer elements in the B$^+$-tree. However, the response thread's processor-use increases with the number of cross-partition queries because a cross-partition query is split into two queries (and thus there are more queries) and servers respond to queries with fixed-size messages, regardless the amount of information contained in the message. The top right graph shows the outgoing bandwidth per server for the cross-partition queries. As expected, by increasing the percentage of cross-partition queries the outgoing bandwidth for each server increases. However, it seems that for 75% and 100% cases the bandwidth is not scaling as expected. To avoid the server's outgoing bandwidth as a bottleneck, one can keep adding

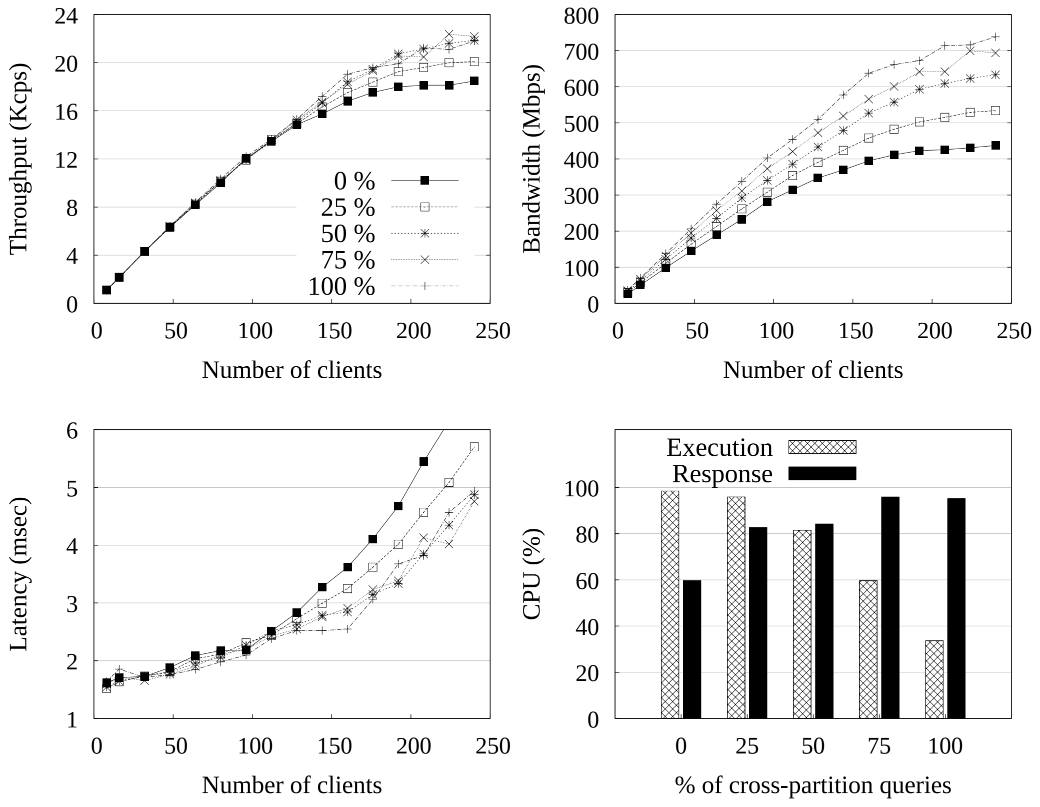


Figure 4.9. Impact of cross-partition queries on the performance with 3 replicas in each partition; top-left graph shows throughput measured in Kilo commands per second (Kcps); top-right graph shows outgoing bandwidth per replica measured in Mega bits per second (Mbps); bottom-left graph shows the corresponding latency; bottom-right graph shows the corresponding CPU utilization.

more replicas to each partition. The effect of 3 replicas in each partition is shown in Figure 4.9 where the maximum achievable throughput for all the cases is increased compared to the 2-replica case (see Figure 4.8). The bottom-right graph depicts the CPU usage for threads responsible for execution and responses. As the percentage of cross-partition queries increase, the responding thread consumes more CPU and the executing thread's CPU usage decreases. Moreover, compared to the 2-replica case the outgoing bandwidth per server is no more a bottleneck.

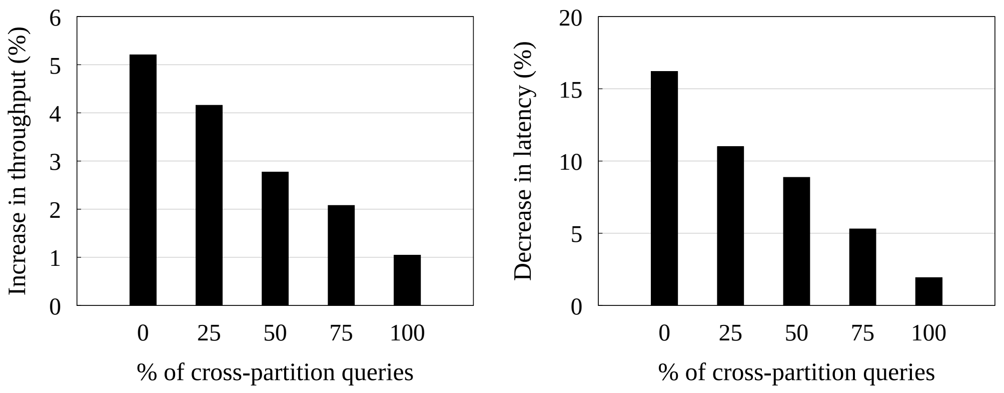


Figure 4.10. Improvements of performance over SMR when speculative execution is combined with state partitioning for cross-partition queries; left graph shows increase in throughput and right graph the reduction in latency.

4.4.6 Speculation and partitioning

Our final set of experiments considers the combined effects of speculative execution and state partitioning. Figure 4.10 shows the relative improvements of the speculative execution technique over state-machine replication with state partitioning for different percentages of cross-partition queries. In all configurations the technique is effective in that it decreases response time, with minor improvements in throughput. The reason for the improvement to decrease with the number of cross-partition queries is that the execution time in a server of a cross-partition query is smaller than the execution time of a single-partition query, as explained above. Therefore, the window of opportunity for speculative execution is narrower (see Section 4.2.1).

4.4.7 Conclusions from the experiments

The following conclusions can be drawn from our experiments:

- Regardless the workload, once the service is replicated, response time increases. Speculative delivery is effective in reducing the negative effect of ordering on latency but as we observed in our experiments the improvement is not significant. M-Ring Paxos is a highly optimized protocol and reducing its latency via speculation was challenging and required extensive optimizations in the implementation. The effect of speculation in improving performance depends on the specific properties and efficiency of the atomic broadcast protocol and duration of execution on replicas.

- Our experiments also indicate that state partitioning can greatly improve the throughput regardless the workload.

4.5 Conclusion

State-machine replication is a well-known approach to building fault tolerant services. The idea is to fully replicate the service state on several servers and execute every client command in every nonfaulty server in the same order. Although some optimizations for performance are possible, inherently the technique introduces an overhead in service response time and is limited by the throughput of a single server. To mitigate these drawbacks, in this chapter we have studied the effect of speculative execution and state partitioning on state-machine replication while using the M-Ring Paxos protocol.

To enable speculative delivery we have implemented a server in which several threads co-operate to execute the requests before their final order is determined by the M-Ring Paxos protocol. The results of our experiments on a B^+-tree service show that speculative execution at best reduces the response time by only 16.2%. M-Ring Paxos is a highly efficient ordering protocol and the execution of operations in our B^+-tree service are quite fast. Therefore, the improvement obtained by speculative execution is not significant. To improve throughput, we have slightly modified the M-Ring Paxos protocol to broadcast instances via several ip-multicast groups to which different partitions subscribe. Our experiments show that state partitioning allows our B^+-tree service to scale with a throughput near 4 times greater than classic state-machine replication.

Chapter 5

Multi-Ring Paxos

Performance of an atomic broadcast protocol is restrained from scaling when the resources of an individual participant rather than the aggregate resources of all the participants in the protocol, are fully consumed. In this situation adding more participants increases the availability and fault-tolerance of the atomic broadcast protocol without any positive impact on the scalability of the performance. We observe that Ring Paxos is also subject to this problem. In this chapter we propose Multi-Ring Paxos as an atomic multicast protocol to address the scalability issues of Ring Paxos. Multi-Ring Paxos is a collection of independent instances of Ring Paxos that are coordinated via a set of predefined parameters. As our evaluations show, performance of Multi-Ring Paxos scales linearly as new participants and thus resources are added to the system.

5.1 Problem statement

State-machine replication requires all the replicas to execute all the requests in the same order. As mentioned in the previous chapters, delivering requests to all the replicas in a unique order is often done by atomic broadcast protocols. Atomic broadcast protocols must be efficient enough not to prevent a capable service with a high execution capacity from meeting the increasing demands of its clients. In other words, ordering requests must cost less than executing them, or otherwise the maximum performance will be determined by the atomic broadcast protocol rather than the execution power of the replicas.

Similarly to our procedure in designing Ring Paxos protocols in Chapter 3, an atomic broadcast protocol can be optimized to achieve high efficiency in ordering. However, no matter how efficient, the maximum throughput will be limited as soon as a participant's resources such as CPU, network, or disk are saturated.

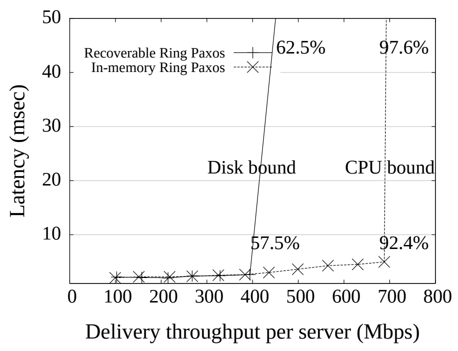


Figure 5.1. Performance of In-memory and Recoverable Ring Paxos.

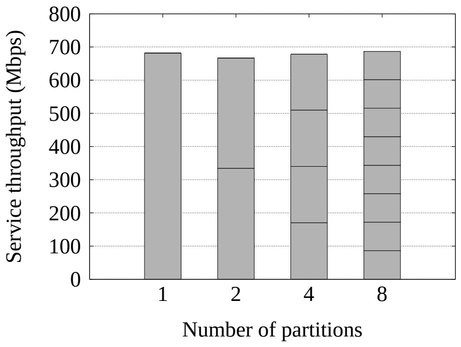


Figure 5.2. Performance of a partitioned service using In-memory Ring Paxos.

After reaching this point, adding more nodes although improves availability, does not enhance performance. As an example we observed this problem in the experiments of Ring Paxos protocols (see Chapter 3 Sections 3.5.4 and 3.5.5). Thus, the maximum throughput of an atomic broadcast protocol is dictated by the capacity of the individual participants and not by the aggregated capacity of all the participants. In this sense atomic broadcast protocols are not scalable. We define scalability to be the ability of a group communication system to increase throughput, measured in number of messages ordered per time unit, when resources (i.e., nodes) are added.

We illustrate the scalability problem with an experiment of an open source implementation of M-Ring Paxos protocol [58].[1] Hereafter we distinguish between two versions of M-Ring Paxos protocol, In-memory and Recoverable. The durability of a consensus instance is configurable: if a majority of *acceptors* is always operational, then consensus decisions can be stored in the main memory of acceptors only (In-memory Ring Paxos). Without such an assumption, consensus decisions must be written on the acceptors' disks (Recoverable Ring Paxos). The maximum throughput of In-memory Ring Paxos is determined by what the CPU or the network interface of an acceptor can handle, whichever becomes a bottleneck first. In Recoverable Ring Paxos, the maximum throughput is limited by the bandwidth sustained by an acceptor's disks. Figure 5.1 shows the performance of In-memory and Recoverable Ring Paxos. In-memory Ring Paxos is CPU-bound: throughput can be increased until approximately 700 Mbps, when the *coordinator*, reaches its maximum processing capacity. When this happens,

[1]For experimental details see Section 5.4

even small increases in the coordinator's load result in large increases in delivery latency. Recoverable Ring Paxos is bounded by the bandwidth of the acceptors' disks. At maximum throughput, around 400 Mbps, the acceptors approach the maximum number of consensus instances they can store on disk per time unit. Notice that at this point the coordinator has moderate processing load, around 60%. In either case, adding resources (i.e., acceptors) will not improve performance.

Effect of state partitioning in overcoming the scalability issue. If executing requests is more costly than ordering them, then throughput will be dictated by the number of requests a server can execute per time unit and not by the number of requests that Ring Paxos can order. In such cases, one solution is to partition the service into sub-services (e.g., Chapter 4, [59]), each sub-service replicated using state-machine replication. Requests concerning a single partition are submitted to and executed by the involved partition only; requests concerning multiple partitions must be consistently ordered across partitions and executed by all involved partitions. As a result, if most requests affect a single partition (or few partitions), the scheme improves performance as the various partitions can execute requests in parallel. As presented in Chapter 4, Ring Paxos can be configured to work with partitioned services by ordering all messages and selectively delivering them to the concerned partitions only. By partitioning a service, the cost of executing requests can be distributed among partitions.

But if a service can be partitioned into a large number of sub-services, then throughput may be limited by the overall number of requests that Ring Paxos can order and deliver to the various partitions, and not by the capacity of the servers to execute the requests. Figure 5.2 illustrates this with a multi-partition service implemented using In-memory Ring Paxos. To emphasize our point, we assess the overall system throughput of a dummy service: delivered messages are simply discarded by the servers, that is, requests take no time to be executed. In this experiment, all submitted requests are single-partition and evenly distributed among partitions. The graph shows that the throughput of M-Ring Paxos does not increase as partitions and nodes (three per partition) are added. Instead, since the total throughput sustained by M-Ring Paxos is approximately the same for the various configurations, the more partitions a configuration has, the less throughput can be allocated to each partition.

If requests access only a single partition, then one can expect a system with *n* partitions to provide *n* times the throughput of a single-partition system. Such a system is a scalable system. In reality, as shown in Figure 5.2, this only happens if the group communication primitive itself scales, meaning that the number of messages per time unit ordered and delivered by the primitive grows with the size of the system. In this chapter we present Multi-Ring Paxos, an atomic multicast primitive with this property. The key insight in Multi-Ring Paxos is to compose an unbounded number of parallel instances of Ring Paxos in order to scale throughput. While the idea behind Multi-Ring Paxos is conceptually simple, its realization entailed non-obvious engineering decisions, which we discuss in the following sections.

5.1.1 Outline

The remainder of this chapter is organized as follows. In Section 5.2 we present Multi-Ring Paxos, discuss the algorithm, and explain several optimizations to improve its performance. In Section 5.3 we review related work. In Section 5.4 we evaluate Multi-Ring Paxos and compare its performance to other protocols. We end this chapter by presenting our conclusions in Section 5.5.

5.2 Multi-Ring Paxos

Multi-Ring Paxos implements atomic multicast, a group communication abstraction whereby senders can atomically multicast messages to groups of receivers; atomic multicast ensures ordered message delivery for receivers that deliver messages in common (see Chapter 2 for more details). In brief, Multi-Ring Paxos assigns one instance of M-Ring Paxos to each group (or set of groups). Receivers that subscribe to a single group will have their messages ordered by the M-Ring Paxos instance responsible for this group. Receivers that subscribe to multiple groups will have multiple sources of messages and use a deterministic merge mechanism to ensure ordered delivery. Most of the complexity of Multi-Ring Paxos lies in its deterministic merge procedure, which accounts for dynamic load and imbalances among the various instances of M-Ring Paxos, without sacrificing performance or fault tolerance. In this section we discuss its properties in detail.

5.2.1 Overview

Multi-Ring Paxos uses multiple independent instances of M-Ring Paxos to scale throughput without sacrificing response time—hereafter, we refer to a M-Ring Paxos instance as a “ring” and assume the existence of one ring per group (we revisit this assumption in Section 5.2.4). Learners subscribe to the groups they want to deliver messages from. Within a group, messages are ordered by the ring responsible for the group. If a learner subscribes to multiple groups, it uses a deterministic procedure to merge messages coming from different rings. Although deterministically merging messages from multiple rings is conceptually simple, its implementation has important performance consequences, as we explain next.

In brief, learners implement the deterministic merge in round-robin fashion, delivering a fixed number of messages from each group they subscribe to in a pre-defined order. More precisely, each group has a unique identifier, totally ordered with any other group identifier. If a learner subscribes to groups $g_{l_1}, g_{l_2}, ..., g_{l_k}$, where $l_1 < l_2 < ... < l_k$, then the learner first delivers M messages from g_{l_1}, then M messages from g_{l_2}, and so on, where M is a parameter of the algorithm. In order to guarantee ordered delivery, the learner may have to buffer messages that do not arrive in the expected pre-defined order.

This scheme has two drawbacks, which we illustrate with an example. Assume that a learner subscribes to groups g_1 and g_2, which generate messages at rates λ_1 and λ_2, respectively, where $\lambda_1 < \lambda_2$. *First*, the learner's delivery rate will be $2\lambda_1$, as opposed to the ideal $\lambda_1 + \lambda_2$. *Second*, the learner's buffer will grow at rate $\lambda_2 - \lambda_1$ and will eventually overflow.

One way to address these two drawbacks is as follows. Rather than defining a global value for M we define a value of M for each group that accounts for different rates: If for each M_1 messages delivered for g_1, the learner delivers $M_2 = M_1\lambda_2/\lambda_1$ messages for g_2, then its total delivery rate will tend to the ideal. In the general case of a learner that subscribes to groups $g_{l_1}, g_{l_2}, ..., g_{l_k}$, it follows that $M_{l_1}/\lambda_{l_1} = M_{l_2}/\lambda_{l_2} = ... = M_{l_k}/\lambda_{l_k}$ must hold in order for the learner to deliver messages at the ideal rate of $\lambda_{l_1} + \lambda_{l_2} + ... + \lambda_{l_k}$. Such a mechanism, however, requires estimating the message rate of each group and dynamically adapting this estimate during the execution. Moreover, to avoid buffer overflows, learners have to quickly adapt to changes in the message rate of a group. Our strategy does not require adapting to a group's message rate. Instead, we define λ, the *maximum expected message rate* of any group, a parameter of the system.

The coordinator of each ring monitors the rate at which messages are generated in its group, denoted μ, and periodically compares λ to μ. If μ is lower than λ, the coordinator proposes enough "skip messages" to reach λ. Skip messages waste minimum bandwidth: they are small and many can be batched in a single consensus instance.

Figure 5.3 illustrates an execution of Multi-Ring Paxos with two groups where $M = 1$. Learner 1 subscribes to group g_1; learner 2 subscribes to groups g_1 and

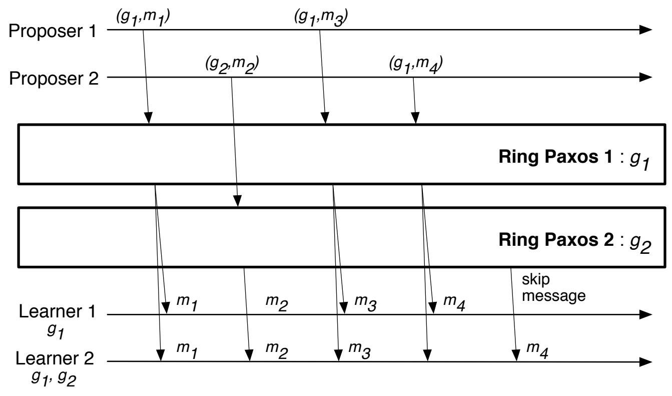


Figure 5.3. Muli-Ring Paxos with two rings and $M = 1$.

g_2. Notice that after receiving message m_4 learner 2 cannot deliver it since it must first deliver one message from group g_2 to ensure order. Therefore, learner 2 buffers m_4. Since learner 1 only subscribes to g_1, it can deliver all messages it receives from first ring as soon as it receives them. At some point, the coordinator of the second ring realizes its rate is below the expected rate and proposes to skip a message. As a consequence, learner 2 can deliver message m_4.

5.2.2 Multi-Ring Paxos in detail

Algorithm 1 presents Multi-Ring Paxos in detail. To multicast message m to group g, a proposer sends m to the coordinator of g (lines 3–4), which upon receiving m, proposes m in consensus instance k (lines 11–12). The acceptors execute consensus instances as in M-Ring Paxos (line 22). For simplicity, in Algorithm 1 only one message is proposed in every consensus instance (in our prototype, multiple messages are batched and proposed in a single instance).[2] Since consensus instances decide on batches of fixed size, if we set λ to be the maximum expected consensus rate, as opposed to the maximum expected message rate, we can easily determine λ since we know the maximum throughout of M-Ring Paxos.

The coordinator sets a local timer (lines 9 and 20), which expires in intervals of Δ time units. In each interval, the coordinator computes μ, the number of consensus instances proposed in the interval (line 14). If μ is smaller than λ (line 15), the coordinator proposes enough *skip instances*, i.e., empty instances,

[2]A consensus instance is triggered when a batch is full or a timeout occurs. We use batches of 8 Kbytes, as this results in high throughput (see Chapter 3 for more details).

```
1: **Algorithm 1: Multi-Ring Paxos** (executed by process $p$)
2: Task 1 (proposer)
3: To multicast message $m$ to group $g$:
4:    send $m$ to coordinator of $g$
5: Task 2 (coordinator)
6: Initialization:
7:    $k \leftarrow 0$
8:    $prev\_k \leftarrow 0$
9:    set timer to expire at current time $+ \Delta$
10: upon receiving $m$ from proposer
11:    propose($k, m$)
12:    $k \leftarrow k + 1$
13: upon timer expires
14:    $\mu \leftarrow (k - prev\_k)/\Delta$
15:    if $\mu < \lambda$ then
16:       $skip \leftarrow prev\_k + \Delta\lambda$
17:       for $k \leftarrow k$ to $skip$ do
18:          propose($k, \perp$)
19:    $prev\_k \leftarrow k$
20:    set timer to expire at current time $+ \Delta$
21: Task 3 (acceptor)
22: execute consensus (Phases 1 and 2 of Ring Paxos)
23: Task 4 (learner)
24: Initialization:
25:    for $i \leftarrow 1$ to $\gamma$ do
26:       if $p \in g_i$ then $k_i \leftarrow 0$
27: repeat forever
28:    for $i \leftarrow 1$ to $\gamma$ do
29:       if $p \in g_i$ then
30:          repeat $M$ times
31:             wait for decide($k_i, v$)
32:             if $v \neq \perp$ then deliver $v$
33:             $k_i \leftarrow k_i + 1$
34: Algorithm variables:
35:    $k$ : current consensus instances in a group (coordinator)
36:    $prev\_k$ : value of $k$ at the beginning of an interval
37:    $\mu$ : number of consensus instances per time in a group
38:    $skip$ : consensus instances below optimum in last interval
39:    $k_i$ : the next consensus instance at group $g_i$ (learner)
40: Algorithm parameters:
41:    $\gamma$ : number of groups
42:    $\Delta$ : duration of an interval (i.e., time between samplings)
43:    $M$ : number of consecutive messages delivered for a group
44:    $\lambda$: expected number of consensus instances per $\Delta$
```

to make up for the missing ones (lines 16–18). Notice that although in Algorithm 1 the coordinator executes a propose for each missing instance, in our prototype this is implemented much more efficiently by proposing a batch of instances using the same physical messages. The coordinator then sets the timer for the next interval (line 20).

For each group g_i to which the learner subscribes, the number of the next consensus instance in which it will participate is stored in variable k_i (lines 25–26). The procedure at the learner consists in deterministically delivering M messages (lines 30–32) multicast to each group g_i subscribed by the learner (lines 28–29). Since groups are totally ordered according to their unique identifiers, each two learners will round robin through the groups they subscribe to in the same order, and hence respect multicast order.

5.2.3 Failures and reconfigurations

Algorithm 1 assumes that rings guarantee progress individually. Therefore, for each ring, up to $f < n/2$ acceptors can fail, where n is the total number of acceptors in a ring. To reduce response time, M-Ring Paxos keeps $f+1$ acceptors in the ring only (see Chapter 3); the remaining acceptors are spares and could be shared by multiple rings in Multi-Ring Paxos, similarly to Cheap Paxos [14].

When an acceptor is suspected to have failed, its ring must be reconfigured, excluding the suspected acceptor and including a new one, from the spares. Until the ring is reconfigured, learners that subscribe to this ring cannot deliver messages broadcast to this ring and to any other ring the learner also subscribes. We assess the effects of reconfiguration in Section 5.4.7. Recovering from lost messages is done with retransmissions, as in M-Ring Paxos protocol (see Chapter 3).

5.2.4 Extensions and optimizations

Algorithm 1 can be optimized for performance in a number of ways. The coordinator does not propose a single message in a consensus instance, but a batch of messages. Moreover, multiple skip instances for an interval are executed together. Thus, the cost of executing any number of skip instances is the same as the cost of executing a single skip instance. Another issue concerns the mapping of groups to rings (i.e., instances of M-Ring Paxos). If there are as many rings as groups, then we can have one group per ring—this is the setting used in our experiments. Alternatively, multiple groups can be mapped to the same ring. The drawback of such a setting is that some learners may receive messages from groups they do not subscribe to. Such messages will not be delivered to the application, but they waste the learner's incoming bandwidth and processor. While there are many strategies to address this issue (e.g., a simple one is to assign the busiest groups to different rings), we note that mapping γ groups to δ rings, where $\gamma > \delta$, is an optimization problem with implications that go beyond the scope of this work [60]. [3]

5.2.5 Additional properties of Multi-Ring Paxos

In this section we discuss two advantages provided by Multi-Ring Paxos:

State-partitioning is not a must. To use Multi-Ring Paxos, a service does not have to partition its state. Requests can be submitted to different rings and ordered independently. Deterministic merge guarantees that all the replicas hosting the full state will deliver and process requests in an identical order. This is important as for example not all the services are easy to be partitioned. Thus partitioning the state is not a must to benefit from the high performance offered by Multi-Ring Paxos.

Machines can be shared among rings. The amount of work performed by processes in Ring Paxos differs from one role to the other. For example often the machine on which an acceptor is located has less CPU utilization compared to the machine on which the coordinator is located. By deploying multiple rings and carefully positioning their processes on the same set of machines Multi-Ring Paxos can make a better use of the same number of machines without compromising fault tolerance. For example a coordinator from one ring can be co-located with an acceptor from another ring on the same machine.

5.3 Related work

Multi-Ring Paxos is an atomic multicast protocol. Differently from atomic broadcast, atomic multicast protocols can be made to scale under certain workloads. In the following we focus the discussion mostly on atomic multicast and review atomic broadcast protocols that share some similarities with Multi-Ring Paxos. For a more comprehensive review on atomic broadcast refer to Chapter 3.

[3]For a proof of correctness see Appendix.

Atomic multicast. Although the literature on atomic broadcast protocols is vast [**?**] (see also chapter 3), few atomic multicast algorithms have been proposed. Possibly, the first atomic multicast algorithm is presented in [61] which is an algorithm for failure-free scenarios. In this algorithm, the destination processes of a message *m* exchange timestamps and eventually decide on *m*'s final timestamp. The destinations deliver messages according to the message's final timestamp. The algorithm scales under certain workloads since only the destinations of a message are involved in its ordering.

Several papers have proposed extensions to render the algorithm in [61] fault tolerant [62; 63; 64; 65]. The basic idea behind these algorithms is to replace failure-prone processes by fault-tolerant groups of processes; each group implementing the logic of the original algorithm by means of state-machine replication. Different algorithms have proposed different optimizations of this basic idea, all based on the assumption that groups do not intersect. An algorithm that departures from the previous proposals appears in [66]. The idea is to daisy-chain the set of destination groups of a message according to the unique group ids. The first group runs consensus to decide on the delivery of the message and then hands it over to the next group, and so on. Thus, the latency of a message depends on the number of destination groups.

Most previous work on atomic multicast had a theoretical focus. One notable exception is the Spread toolkit [33]. Spread is a configurable group communication system, which supports the abstraction of process groups. It relies on interconnected daemons, essentially the components that handle the physical communication in the system, to order messages. Participants connect to a daemon to multicast and deliver messages. The abstraction of groups in Spread, however, was not created for performance, but to simplify application design. In Section 5.4 we experimentally compare Multi-Ring Paxos and Spread.

Skipping instances. Mencius is a protocol that implements state-machine replication in a wide-area network [27]. Mencius is a multi-leader protocol derived from Paxos. The idea is to partition the sequence of consensus instances among the leaders to amortize the load and better balance the bandwidth available at the leaders. Similarly to Multi-Ring Paxos, leaders can account for load imbalances by proposing skip instances of consensus. Differently from Multi-Ring Paxos, Mencius does not implement the abstraction of groups; it is essentially an atomic broadcast protocol.

Deterministic merge. Multi-Ring Paxos's deterministic merge is conceptually similar to the work proposed in [67], which totally orders message streams in a widely distributed publish-subscribe system. Differently from Multi-Ring Paxos merge scheme, the mechanism proposed in [67] uses approximately synchronized clocks to estimate the expected message rates of all publishers and then merges messages throughout the network in the same way.

5.4 Experimental evaluation

In this section, we briefly describe some implementation details and then evaluate our protocol with respect to the following aspects:

- **Scalability of Multi-Ring Paxos.** As we argued before, performance of an atomic broadcast protocol does not scale with the number of its participants. The aim of this experiment is to compare the scalability of our proposed solution, Multi-Ring Paxos, with several other protocols (Section 5.4.3).

- **Impact of Δ, M, and λ on performance.** Rings in Multi-Ring Paxos may be imbalanced with respect to client load. Therefore, in certain intervals (Δ) the coordinator of each ring skips some of its instances—given that its ring is under-loaded based on a pre-specified global parameter (λ). This is important since a slow ring should not prevent a learner from delivering messages from other rings. We recall that a learner uses a deterministic merge strategy to deliver M instances from each ring in a round-robin mode. We perform three experiments to investigate the effects of Δ, M, and λ on the performance of a learner who subscribes to more than one ring (Sections 5.4.4, 5.4.5, and 5.4.6).

- **Impact of discontinued communication on performance.** Learners can choose to subscribe to more than one ring. This experiment is performed to investigate the effect of ring failures or the effect of interrupted communication with rings on the performance of a learner that subscribes to multiple rings (Section 5.4.7).

5.4.1 Hardware settings

We ran the experiments in a cluster of Dell SC1435 servers equipped with 2 dual-core AMD-Opteron 2.0 GHz CPUs and 4GB of main memory. The servers are interconnected through an HP ProCurve2900-48G Gigabit switch whereas the round trip time is 0.1 mili second.

5.4.2 Implementation and experimental setup

We have implemented a prototype of Multi-Ring Paxos based on an open-source version of M-Ring Paxos [58]. Furthermore, we differentiate between In-memory and Recoverable M-Ring Paxos, where in the latter acceptors persist their data on disk.

In all the experiments, unless specified otherwise, λ, Δ, and M are set to 9000 consensus instances per interval, 1 millisecond, and 1 message, respectively and the size of application-level messages is 8 Kbytes. In all the experiments each group has a dedicated ring. Recoverable Multi-Ring Paxos uses asynchronous disk writes. Thus, in the experiments both In-memory and Recoverable Multi-Ring Paxos assume that a majority of acceptors is operational during each consensus instance. To remove peaks in latency due to flushes to disk, we report the average latency after discarding the 5% highest values. Whenever a disk write happens, clients experience high values for latency. Thus eliminating 5% of the highest values allows to represent the latency that a client experiences most frequently. When analyzing throughput, we report the aggregated throughput of the system, which combines the throughput of all the groups.

5.4.3 Scalability of Multi-Ring Paxos

Depending on the number of learners and groups, there can be many configurations of Multi-Ring Paxos. Two extreme cases are when (1) each learner subscribes to only one group and (2) each learner subscribes to all the groups. The first case assesses the scalability of Multi-Ring Paxos since throughput is not limited by the incoming bandwidth of a learner. The second case assesses the ability of learners to combine messages from multiple rings.

(1) Best case. When each learner subscribes to only one group (see Figure 5.4), the throughput of the learner is limited by the maximum throughput of the ring in charge of the learner's group. This is because before the learner uses up its local resources, the coordinator of each ring in In-memory Multi-Ring Paxos saturates its CPU and the acceptors in Recoverable Multi-Ring Paxos reach their maximum disk bandwidth (see also Figure 5.1). The throughput of both In-memory and Recoverable Multi-Ring Paxos protocols grows linearly with the number of partitions, peaking at more than 5 Gbps with In-memory Multi-Ring Paxos and about 3 Gbps with Recoverable Multi-Ring Paxos. As a reference, we also present the performance of Spread, M-Ring Paxos, and LCR. Spread implements the abstraction of groups but does not scale with the number of groups. LCR [12] is

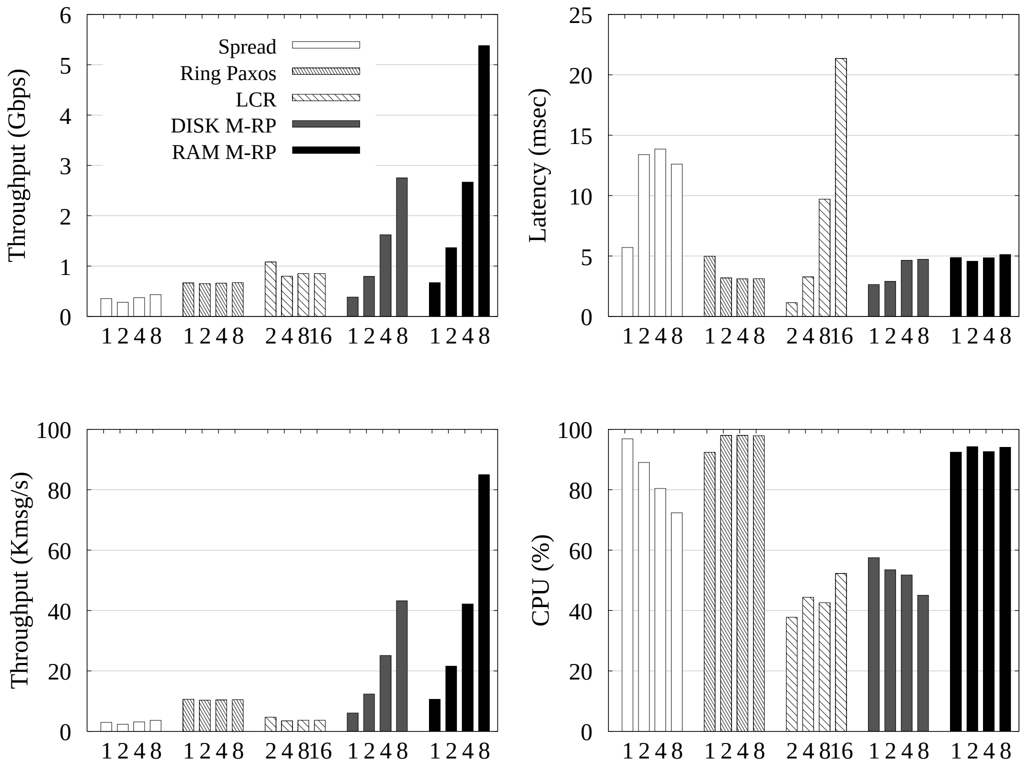


Figure 5.4. Performance of In-memory Multi-Ring Paxos (RAM M-RP) and Recoverable Multi-Ring Paxos (DISK M-RP), compared with Spread, M-Ring Paxos and LCR; the x-axis shows number of partitions for RAM M-RP, DISK M-RP and M-Ring Paxos; number of daemons/groups for Spread; and number of nodes in the ring of LCR; there are 2 acceptors per partition in RAM M-RP and DISK M-RP, and a fixed number of 2 acceptors in M-Ring Paxos; the CPU graph shows the CPU of the most-loaded node, which for RAM M-RP, DISK M-RP and M-Ring Paxos is the coordinator; the latency graph shows the corresponding latencies.

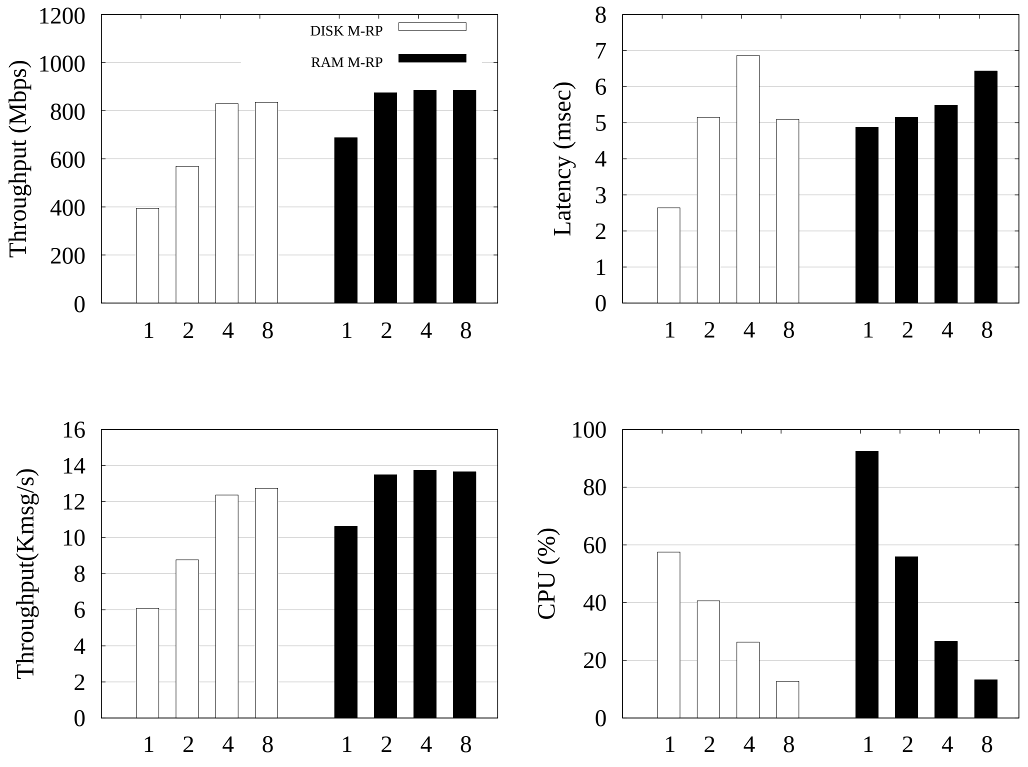


Figure 5.5. Performance of Multi-Ring Paxos when each learner subscribes to all the groups.

a high performance atomic broadcast protocol and does not implement groups, we have varied the size of the ring to measure its throughput. The packet size used for Spread and LCR are 16 and 32 Kbytes respectively.

(2) Worst case. Figure 5.5 shows the performance of Multi-Ring Paxos when learners subscribe to all the groups. For both Multi-Ring Paxos protocols, with one ring the bottleneck is the single M-Ring Paxos instance. As groups (i.e., rings) are added, the aggregate throughput of the various rings eventually saturates the learners' incoming links. To reach the maximum capacity of a learner, In-memory Multi-Ring Paxos needs two rings and Recoverable Multi-Ring Paxos needs three rings. This experiment illustrates how Multi-Ring Paxos can combine multiple "slow" atomic broadcast protocols (e.g., due to disk writes) to build a much faster protocol.

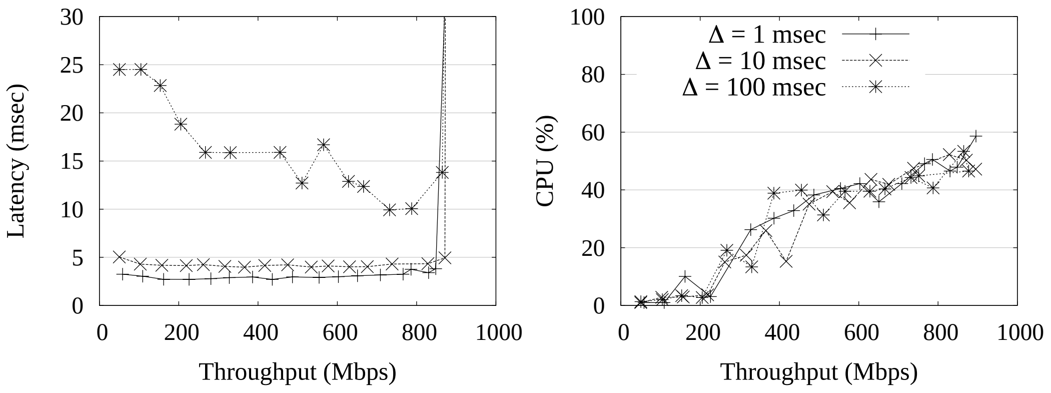


Figure 5.6. The impact of Δ on Multi-Ring Paxos. Latency versus throughput (left) and CPU at the coordinator of one of the rings (right).

5.4.4 Impact of Δ on Multi-Ring Paxos

Recalling from Section 5.2.2, Δ is the interval in which the coordinator of a ring samples the number of executed consensus instances to then check for the need of skip instances. The value assigned to Δ should be big enough to avoid unnecessary samplings, and small enough to allow quick corrections in the rate of the ring. To investigate the effects of Δ, we have deployed In-memory Multi-Ring Paxos with two rings and one learner that subscribes to both rings. The load on both rings is equal and remains constant during the experiment.

As the left-most graph of Figure 5.6 suggests, a large Δ results in higher latency at the learner. Notice that even though each ring has the same rate, small variations in the transmission and handling of messages can lead to the buffering of messages at the learners and increased latency. For large values of Δ (e.g., 100 milliseconds), latency decreases with the throughput. This happens since fewer skip instances are needed and thus the negative effect of a large Δ on the latency diminishes. Unlike latency, the maximum throughput is not affected by Δ, as all configurations reach approximately the same value. This implies that to attain both low latency and high throughput, small values of Δ are preferred.

The right-most graph of Figure 5.6 shows the processing cost of Δ measured as the CPU usage at one of the coordinators. As it is seen, small values of Δ have no additional processing cost. To conclude, we suggest choosing small values for Δ as it results in better performance with no additional cost.

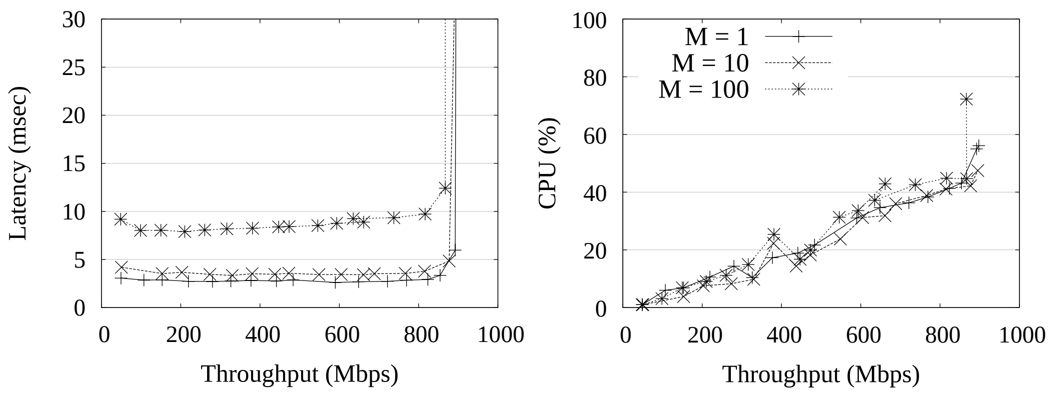


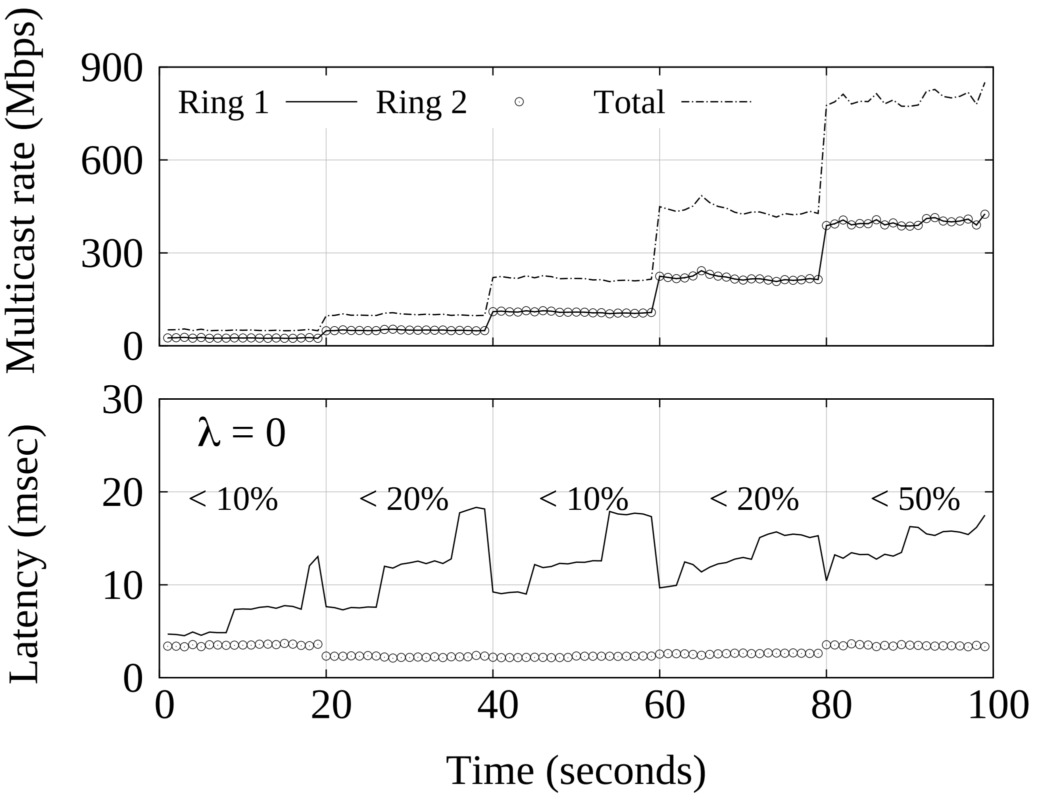
Figure 5.7. The impact of M on Multi-Ring Paxos. Latency versus throughput (left) and corresponding CPU usage in the learner (right).

5.4.5 Impact of M on Multi-Ring Paxos

In this section we evaluate the effect of M in the execution. We recall that M is the number of consensus instances that a learner handles at a time from each ring it expects messages from. In these experiments, we have deployed an In-memory Multi-Ring Paxos with two rings, and one learner that receives messages from both of the rings. As the left graph of Figure 5.7 implies, by increasing the value of M, the average latency increases. The reason is that while M instances of a ring are handled in the learner, instances of other rings are buffered and delayed. As M increases, this delay increases and so does the average latency. As it is evident in Figure 5.7 (right side), M has no effect on the throughput and CPU usage of the learner. Therefore, choosing a smaller value for M to keep the latency low has no additional costs.

5.4.6 Impact of λ on Multi-Ring Paxos

If a learner subscribes to several groups, each with a different message rate, slow groups will delay the delivery of messages multicast to faster groups, and therefore negatively affect the latency and overall throughput observed by the learners. Multi-Ring Paxos copes with these issues by skipping consensus instances and by carefully setting λ, the maximum expected consensus rate of any group. In the following, we investigate the effect of λ on the system. We have conducted three sets of experiments using In-memory Multi-Ring Paxos with two rings and one learner. In the first experiment (see Figure 5.8) proposers multicast messages to the two groups at a fixed and equal rate. In the second



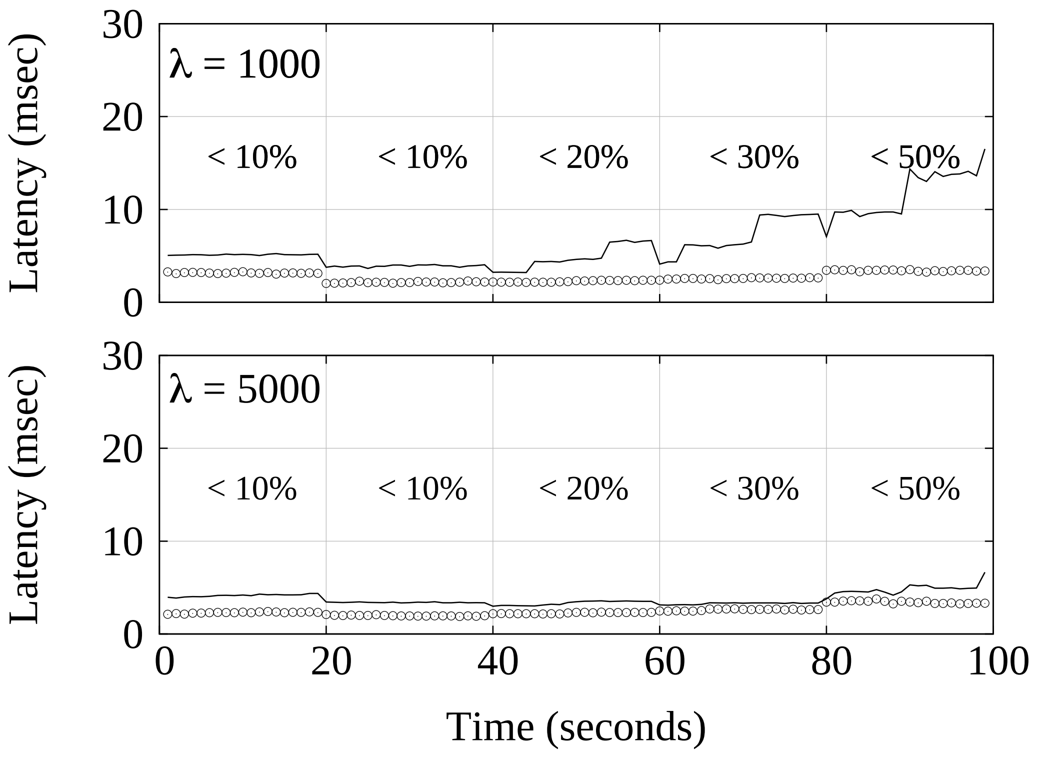


Figure 5.8. The impact of λ when the rates of the rings are constant and equal (percentages show CPU load at ring coordinators).

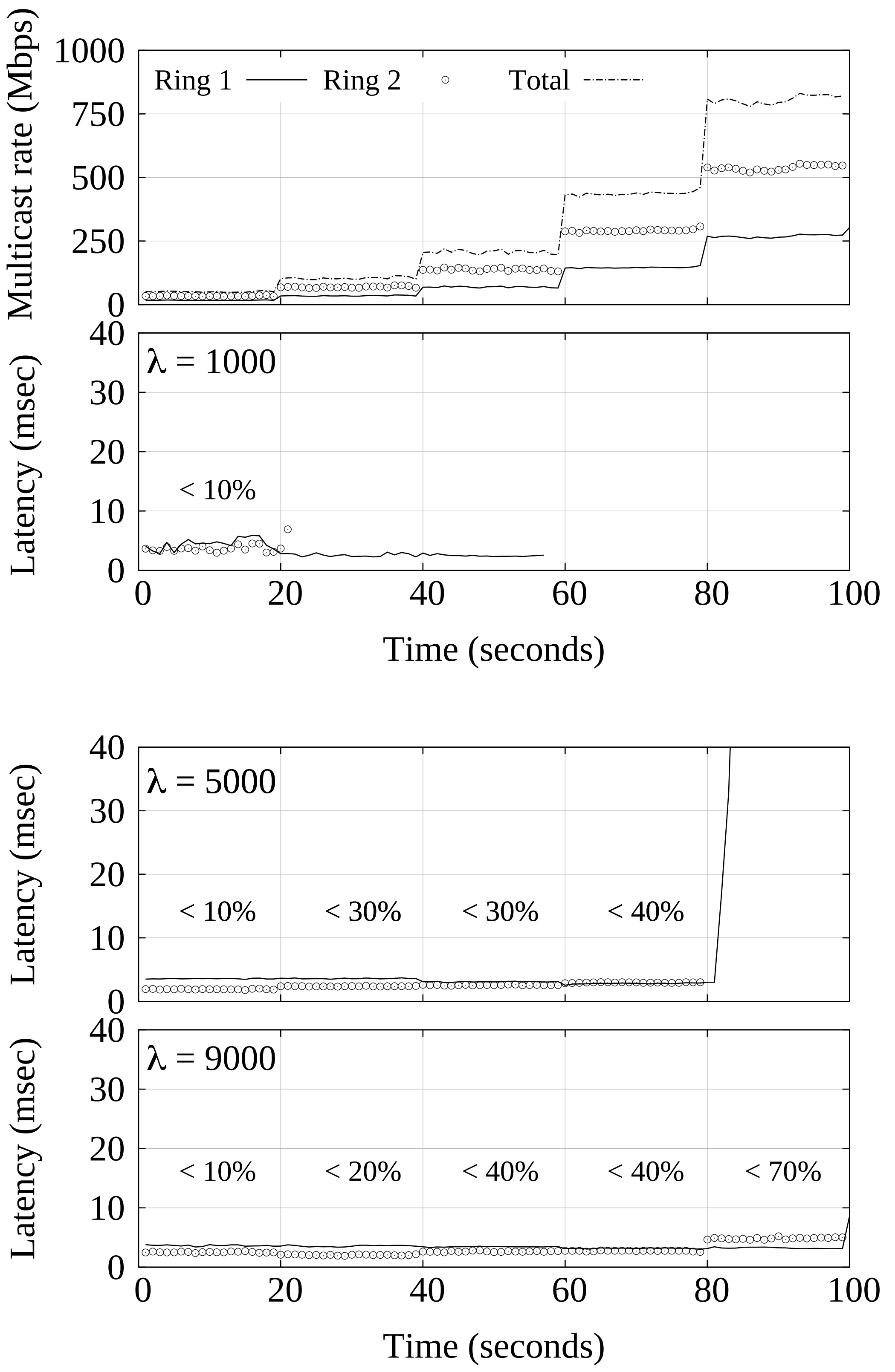
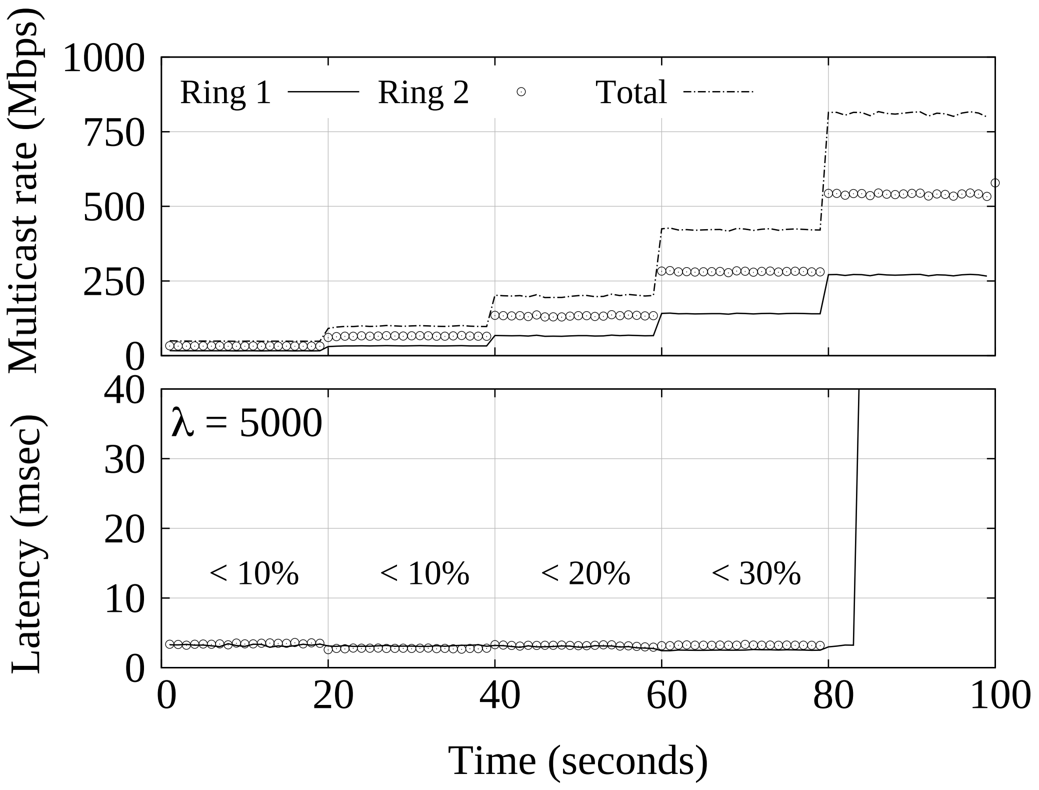


Figure 5.9. The impact of λ when the rates of the rings are constant and one is twice the other (percentages show CPU load at ring coordinators).



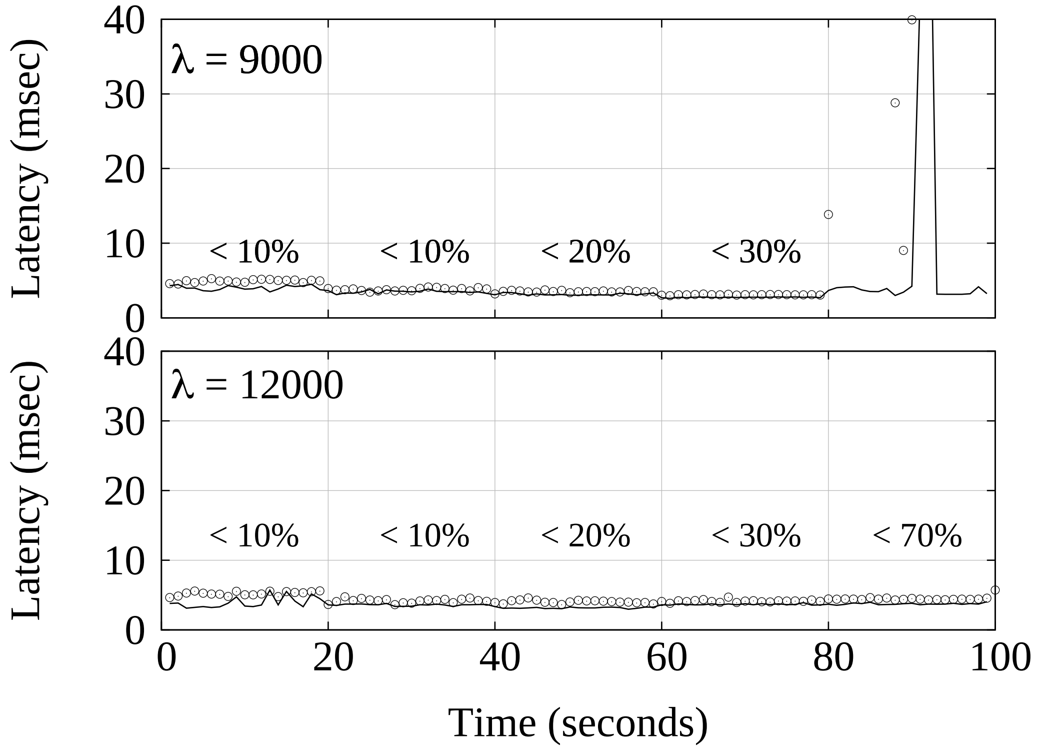


Figure 5.10. The impact of λ when the rates vary over time and on average one is twice the other (percentages show CPU load at ring coordinators).

experiment (see Figure 5.9) the ratio of multicast messages to one of the groups is twice the other, though the multicast rate is constant in both groups throughout the execution. In the last experiment (see Figure 5.10), not only the ratio of multicast messages to one of the groups is twice the other, but their submission rates oscillates over the time such that the average is the same as in the previous experiment. In all the cases, we increase the multicast rate every 20 seconds. In all the figures, the top left graph shows the individual multicast rate per group and the total multicast rate in the system.

In Figure 5.8, we initially set λ to 0 (i.e., no mechanism to skip consensus instances). Even though the group rates are the same, even under low rates the traffic from the rings gets "out-of-sync" at the learner and messages have to be buffered, a phenomenon that the learner does not recover from. With λ equal to 1000, latency remains stable with higher loads, but the problem still exists at very high load. With λ set to 5000 the problem is solved. Figure 5.9 illustrates the problem when the learner's buffer overflows (i.e., $\lambda = 1000$ after 20 seconds and $\lambda = 5000$ after 80 seconds). A buffer overflow brings the learner to a halt since it cannot deliver buffered messages and new messages keep arriving. A large value of λ is enough to handle the most extreme loads in this experiment. Figure 5.10 shows a similar situation, which is only solved when λ is set to 12000. Skipping up to 12000 consensus instances in an interval of one second, where each instance decides on messages of 8 kB, corresponds to "skipping" up to 750 Mb of data per second, approximately the maximum throughput achieved by a ring. We recall that all such instances are skipped using a single consensus execution.

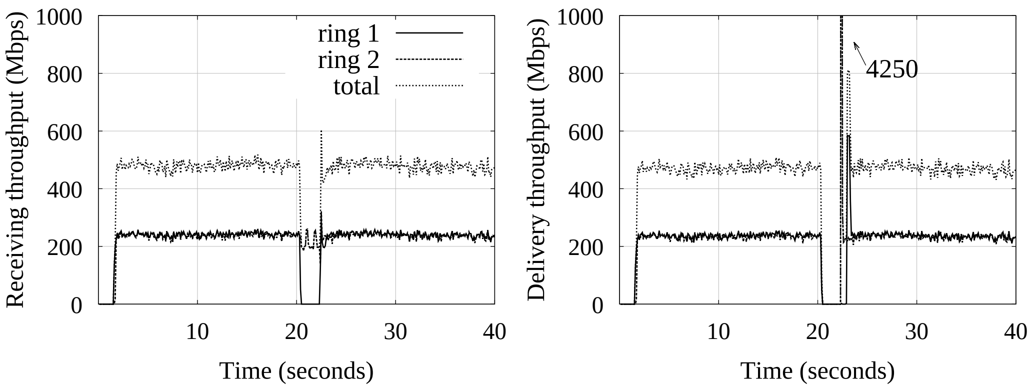


Figure 5.11. The impact of a coordinator failure in a learner of In-memory Multi-Ring Paxos.

5.4.7 Impact of discontinued communication

We now investigate the effect of discontinued communication (e.g., due to a coordinator failure) in Multi-Ring Paxos. In this experiment we deploy two rings and a learner that subscribes to these rings. Each ring generates messages with the same constant rate of approximately 4000 messages per second in average. In steady state, the learner receives and delivers approximately 500 Mbps of data (see Figure 5.11). After 20 seconds we stop the coordinator of ring 1, bringing the receiving throughout from this ring at the learner to zero. Although messages still arrive at the learner from ring 2, the learner buffers such messages as it cannot execute its deterministic merge procedure. The result is that the delivery throughput at the learner drops to zero (right-most graph of Figure 5.11). Notice that after ring 1 stops, the incoming throughput from ring 2 decreases, as the learner does not acknowledge the delivery of messages from group 2 to the node that multicasts to ring 2 and this one slows down its sending rate.

Three seconds later the execution at ring 1 proceeds. We forced a restart after three seconds to emphasize the effects of the discontinuity of traffic. In reality, it takes much less time to detect the failure of a coordinator and replace it with an operational acceptor. When the coordinator of the first ring starts, it notices that no consensus instances were decided in the last intervals and proposes to skip multiple consensus instances. As a result, the learner delivers all messages it has enqueued, momentarily leading to a high peak in the delivery throughput. Then the execution proceeds as normal.

5.4.8 Conclusions from the experiments

The following conclusions can be drawn from our experiments:

- Performance of Multi-Ring Paxos scales as new rings are added to the ensemble. Therefore by using Multi-Ring Paxos the performance of the replicated service will always be determined by the number of requests the service is capable of processing rather than the number of requests the ordering layer can order.

- Although Multi-Ring Paxos uses a few configuration parameters to coordinate the rings, assigning proper values to these parameters does not impose extra processing costs on the system.

- In the presence of failures, given that the failed rings can be replaced in a reasonable time, learners are not subject to buffer overflow that could otherwise paralyze their delivery.

5.5 Conclusion

In this chapter we revisited the scalability of atomic broadcast protocols and proposed the Multi-Ring Paxos protocol that implements atomic multicast. While atomic broadcast induces a total order on the delivery of messages, atomic multicast induces a partial order. Differently from previous atomic multicast algorithms, Multi-Ring Paxos exploits the abstraction of groups in a different way: in Multi-Ring Paxos, messages are addressed to a single group only, but processes can subscribe to multiple groups. In all atomic multicast algorithms we are aware of, messages are multicast to one or more groups, and often groups cannot intersect.

The results of our experiments are promising: by composing eight instances of In-memory Ring Paxos, for example, we can reach an aggregated throughput of more than 5 Gbps, eight times the throughput of a single Ring Paxos instance. Recoverable Ring Paxos has similar scalability, linear in the number of Ring Paxos instances.

Chapter 6

Replicating Parallel Applications with State-Machine Replication

The advent of multi-core processors and their wide availability has revolutionized systems programming and application development strategies. To increase their capacity in serving clients and to benefit from the new hardware, service providers have to parallelize their services. Similarly to their sequential predecessors, however, parallel services must also be continually available to the clients, despite failures. State-machine replication is a well-established strategy to make services fault tolerant. In this chapter, we will look at the capability of the state-machine approach in replicating parallel services. On the one hand, parallel applications require concurrent processing of requests to provide high performance and on the other hand, state-machine replication requires sequential processing of requests to preserve consistency. We will consider the possibility of uniting these two apparently incompatible models and will propose a scalable solution to achieve it.

6.1 Problem statement

Replicas in state-machine replication execute an ordered sequence of client requests sequentially and deterministically. Sequential execution of requests is an important means for ensuring strong consistency promised by state-machine replication. Multithreaded applications on the other hand, allow multiple threads to concurrently process client requests. Concurrent execution of requests is important from a performance perspective, in particular when the servers have access to multi-core processors. Replicating parallel applications by state-machine replication is not straightforward due to their incompatible execution models.

The problem under study in this chapter is to modify state-machine replication to make its integration with parallel services a reality, while preserving the high performance of multithreaded applications and the strong consistency guarantees of state-machine replication.

It has been observed earlier that replicas in state-machine replication can relax the order among the independent commands and execute them concurrently [68]. Two commands are *independent* if they access different variables or they only read the values of the common variables. Two commands are *dependent* if they access at least one common variable, v, and at least one of the commands modifies the value of v. As an example, consider a service composed of three objects x, y, and z and assume commands C_x, C_y, C_z, C_{xy}, where the indices indicate the objects accessed and modified by the commands. Commands C_x, C_y, and C_z access disjoint objects. Thus, they are independent and can be executed in parallel at each replica. Command C_{xy} depends on commands C_x and C_y and must be serialized with C_x and C_y. C_{xy} can be executed in parallel with C_z, however. Several techniques have built on the command interdependencies to introduce parallelism in the execution of commands on replicas [69; 70]. In the next sections, we review these techniques and by identifying their main shortcomings we propose a novel and high performance approach to implementing parallelism in state-machine replication.

6.1.1 Outline

The rest of this chapter is organized as follows: In Section 6.2 we review parallel approaches to state-machine replication. In Section 6.3 we present our model, P-SMR, and discuss its algorithmic details. In Section 6.5 we empirically compare several techniques used in implementing parallel services. Finally in Section 6.6 we conclude this chapter.

6.2 A Survey on Parallel State-Machine Replication

In this section, we review typical architectures for client-server communications and survey several techniques that have adapted state-machine replication to multi-core architectures.

6.2.1 Non-replicated setup

A typical way for clients to interact with a stand-alone (non-replicated) server is by means of remote procedure invocations [71; 72]. Clients access the service by invoking service commands with the appropriate parameters. Client proxies intercept client invocations and turn them into requests that include a command identifier and the marshaled parameters. Requests are delivered by the server proxies, which re-assemble invocations and issue them against the local service. Similarly to remote procedure calls, the client and client proxy (respectively, server and server proxy) can be implemented as a single process, sharing a common address space. The command's response follows the reverse path to the client using one-to-one communication. As depicted in Figure 7.1 (a), in a non-replicated service (i) client requests are communicated to the server directly, without passing through an agreement layer,[1] and (ii) execution of client requests at the server can be multithreaded.

6.2.2 Sequential State-Machine Replication (sequential SMR)

As repeatedly seen in the previous chapters, state-machine replication provides clients with the illusion of a non-replicated service, that is, replication is transparent to the clients. A command issued by a client is handled by the client proxy, which multicasts the command to all replicas and waits for the response from one replica (see Figure 7.1 (b)). Before requests can be executed on the replicas, they are ordered by the agreement layer. Since replicas execute commands deterministically and in the same order, every replica produces the same response after the execution of the same command.

Differently from a non-replicated service, clients remain oblivious to failures, as the service remains operational despite the failure of some of its replicas. In failure-free scenarios, however, a non-replicated service is often more efficient than a replicated service since in the replicated case requests reach the servers through an agreement layer and execution is single-threaded.

6.2.3 Pipelined State-Machine Replication (pipelined SMR)

Having replicas execute commands sequentially by a single thread does not imply that the whole replica's logic must be single-threaded; multiple threads on a

[1]The agreement layer exists in replicated schemes and often encapsulates a communication primitive such as atomic broadcast or atomic multicast, and provides important guarantees to the replication model.

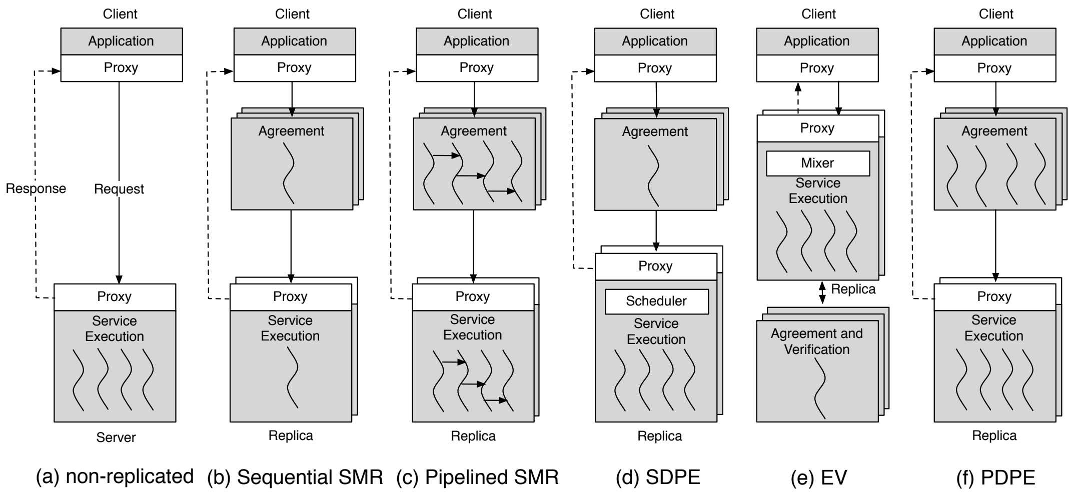


Figure 6.1. Architecture differences among (a) non-replicated service, (b) sequential state-machine-replication, (c) pipelined state-machine replication, (d) sequential delivery-parallel execution (SDPE), (e) execute-verify, and (f) parallel delivery-parallel execution. Agreement layer and replicas are fault-tolerant.

replica can cooperatively handle the requests. For example, one thread receives the requests, another executes the requests, and a third thread responds to the clients. In [73], the authors propose a pipelined architecture to exploit the processing power of multi-core servers. The agreement layer (atomic broadcast) and the replicas are organized as a collection of modules connected through shared message queues where messages are totally ordered (see Figure 7.1 (c)). Although pipelining improves the throughput of state-machine replication, there is always only one thread sequentially executing the commands.

6.2.4 Sequential Delivery-Parallel Execution (SDPE)

Replicas in sequential state-machine replication execute all the commands sequentially by adhering to the order decided by the agreement layer. As we mentioned in Section 6.1, a replica can execute commands that access disjoint variables (independent commands) concurrently without jeopardizing consistency.

To benefit from command inter-dependencies and parallelize execution, some proposals add a deterministic scheduler (also known as parallelizer) to the replicas [70]. The scheduler delivers all the commands ordered through the agreement layer, examines command dependencies, and distributes them among a pool of worker threads for execution (see Figure 7.1 (d)). To distribute the commands among threads, besides considering dependencies, the scheduler can also balance the load among threads. Threads that are less occupied can be given more commands to execute if their execution does not conflict with the commands that are being executed by other threads.

Although thanks to the scheduler the execution is parallelized, the scheduler delivers and dispatches commands sequentially, which restrains the overall performance from scaling. For this reason, we identify these techniques as Sequential Delivery-Parallel Execution (SDPE). Adapting a sequential policy for delivery has its roots in the requirements of SMR where replicas deliver one and only one stream of ordered commands. Synchronization between the scheduler and the worker threads for dispatching commands is yet another performance overhead of this model.[2]

6.2.5 Execute-Verify (EV)

One of the shortcomings of the SDPE model is the agreement layer, where only one stream of ordered requests is generated. Eve [69] addresses this issue by first executing the requests on replicas and then verifying the correctness of the states through a verification stage, hence named as Execute-Verify (EV) (see Figure 7.1 (e)). Eve distinguishes one of the replicas as the primary to which clients send their requests. The primary replica organizes the requests into batches and assigns to each batch a unique sequence number. The primary then transmits the batched requests to the other replicas. All the replicas, including the primary, are equipped with a deterministic *mixer*. Using the application semantics, the mixer converts a batch of requests in to a set of parallel batches such that all the requests in a parallel batch can be executed in parallel. Once the execution of a batch terminates, replicas calculate a token based on their current state and send their token to the verification stage. The verification stage checks the equality of the tokens. If the tokens are equal, replicas commit the requests and respond to the clients. Otherwise, replicas must roll back the execution and re-execute the requests in the order that was determined by the primary when it was batching the requests. The verification stage also adds to Eve the advantage of detecting concurrency bugs [69].

Similar to the scheduler in the SDPE model, the mixer in the Eve may restrict the execution performance since the content of all the requests must be scrutinized by the mixer before they can be executed. Moreover, the primary replica might be overwhelmed by the amount of requests it receives. The verification

[2]We also refer to this model as semi-parallel SMR, or sP-SMR for short.

	Sequential SMR	Pipelined SMR	SDPE	EV	PDPE
Single coordination point	Yes	Yes	Yes	Yes	No
Scalability	None	Limited	Limited	Limited	Unlimited
Order on commands	Total	Total	Total	Total	Partial
Load balancing	None	None	Yes	Yes	Approximative
Application semantics	No	No	Yes	Yes	Yes
Dependency tracking	No	No	Server-side	Server-side	Client-side
Execution strategy	Conservative	Conservative	Conservative	Optimistic	Conservative
Rollback	No	No	No	Yes	No

Table 6.1. A comparison of parallel approaches to state-machine replication.

stage is another synchronization point that besides the mixer and the primary replica can threaten the scalability of this approach.

6.2.6 Parallel Delivery-Parallel Execution (PDPE)

Motivated by the shortcomings of the previous models, In this Chapter we propose P-SMR to parallelize command delivery in addition to command execution; we categorize this model as Parallel Delivery-Parallel Execution (PDPE). P-SMR has no scheduler and several threads on replicas concurrently deliver and execute multiple disjoint streams of ordered commands. To preserve correctness, commands in each stream must be independent from the commands in any other stream. To ensure independency among the concurrently delivered streams, unlike previous approaches in which command dependencies are determined at the replicas, in P-SMR command dependencies are determined by the clients, before commands are ordered. Commands in P-SMR are ordered by an atomic multicast library and clients multicast independent commands to different multicast groups. P-SMR implements a fully parallel model in which independent commands are ordered, delivered, and executed in parallel. Dependent commands are ordered through dedicated multicast groups and executed sequentially (see Figure 7.1 (f)). We will thoroughly describe P-SMR in Section 6.3.

6.2.7 Summary

Table 6.1 shows the main differences among the techniques we have discussed. Both SDPE and EV have centralized entities that can limit scalability: the scheduler and the agreement layer in SPDE; the mixer, the primary replica, and the verification layer in EV. PDPE does not include central roles in its design. Moreover, differently from other approaches, PDPE orders requests using an atomic multicast, as opposed to an atomic broadcast.

The parallelizer in SDPE and the mixer in EV also perform load balancing on the server side. Although in a limited way, clients in PDPE can try to distribute the load evenly among server threads (e.g., by multicasting read commands to different groups).

SPDE, EV, and PDPE rely on tracking command dependencies to parallelize execution on replicas. In SDPE and EV, command dependencies are checked on the server side. In PDPE, however, it is the clients that track dependencies and submit commands to the appropriate multicast groups. Unlike other techniques, due to its optimistic nature, EV may be subject to rollbacks.

Having highlighted its main differences with existing techniques, in the next section we describe our proposed approach in more detail.

6.3 Parallel State-Machine Replication (P-SMR)

In this section we present P-SMR, a new approach to parallelizing state-machine replication. We first discuss the main goals in P-SMR's design and then present its architecture and algorithmic details.

6.3.1 Design goals

P-SMR's design is guided by two main goals:

(1) Preserving replication transparency. In the state-machine replication architecture, replication is transparent for clients: details about communicating with multiple replicas are hidden from the clients and handled by the client proxies and the multicast library (see Figure 7.1 b). Similarly to SMR, P-SMR should not expose replication details to the client application.

(2) Optimizing performance for the common case. P-SMR targets workloads dominated by independent commands, when concurrency is possible. Services whose state is mostly read (e.g., name services) or can be partitioned so that most commands fall in one partition or another but rarely in both (e.g., file systems) are the most suitable services to benefit from P-SMR.

In the following sections, we elaborate on these design principles further.

6.3.2 Client and server organization

P-SMR follows the transparent architecture of state-machine replication, where client and server proxies are created based on the following metrics:

(a) the *signature* of each service command, including the command's identifier and a description of the command's input and output parameters together with the types of these parameters, and

(b) the *command dependencies (C-Dep)*, specifying which commands depend on each other.

Therefore, in addition to providing the server's code, the service designer must also provide the command signatures and the C-Dep.[3] At the clients, command signatures are used by the client proxy to create a request from client invocation and return a response to the client. At the servers, command signatures are used by the server proxy to turn delivered requests into local server invocations and assemble the response of commands.

C-Dep is used to automate the computation of the *Command-to-Groups (C-G)* function, used by both the client proxy to determine the multicast groups a request must be multicast to, and also by the server proxy to coordinate the local execution of dependent commands. Similarly to SMR, a client application in P-SMR will be oblivious to replication. Moreover, since coordination among worker threads, in the case of dependent commands, is handled by the server proxy, a service designed for state-machine replication will work unchanged in P-SMR.

In the rest of this section we define C-Dep, and C-G function together with MPL, a parameter of the system for specifying the multiprogramming level.

C-Dep: defining command dependencies. In our prototype, C-Dep encodes two levels of dependency information:

(a) commands that depend on each other, regardless their parameters (e.g., commands to create and delete objects), and

(b) commands that may be dependent, according to their parameters (e.g., two updates on the same object).

[3]Although the C-Dep can be automatically generated from the signatures and the server's code, in our prototype C-Deps were created manually.

C-Dep includes all such interdependencies; if no entry exists in C-Dep asserting the dependency of two commands, they are independent. Although our encoding is simple, more complex schemes could be used (e.g., [70]). In Section 6.5 we show how this scheme can represent interdependencies in a key-value store.

MPL: multiprogramming level. The multiprogramming level is a parameter of the system that defines the number of worker threads at the servers. It can be set, for example, based on the number of processing units (i.e., cores) at the servers. In a configuration where MPL is set to k, we identify worker threads as $t_1, ..., t_k$. P-SMR organizes threads in k multicast groups such that the i-th thread of each replica, t_i, belongs to group g_i.

C-G: mapping commands to destination groups. The client proxy determines the destination groups of a command using a Command-to-Group (C-G) function that maps the command id and its input parameters to a set of multicast groups. The C-G is part of the client proxy and is created based on the MPL and the C-Dep. Allowing independent commands to execute concurrently is achieved by assigning them to different groups; ensuring proper synchronization amounts to assigning at least one common group to any two dependent commands. The amount of concurrency in a service depends on the interdependencies among the service's commands. These interdependencies are defined by the code that implements each command. In P-SMR the interdependencies among commands are captured by the command dependencies list (C-Dep) and the command code that runs at the replicas. Therefore, a C-Dep that precisely captures interdependencies will likely result in more concurrency at the replicas. For example, consider a service with the following two commands:

- `get_state(in: int x, out: char[] v)`, and
- `set_state(in: int x, char[]v)`,

where `x` is an object identifier and `v` an object value. A simple C-Dep would state that `set_state` depends on any other command, regardless the object accessed. Defining such a C-Dep requires inspecting commands `get_state` and `set_state` and concluding that the first command reads the service's state and the second command modifies the service's state. The C-G for this C-Dep assigns a `get_state` command to a single group (randomly chosen between 1 and k, k being the multiprogramming level) and a `set_state` command to all groups, as shown next:

```
function C-G($cid$)
  switch ($cid$)
    case get_state: return(random(1..$k$))
    case set_state: return($ALL\_GROUPS$)
```

A more complex C-Dep identifies that `set_state` depends only on other commands on the same object. In this case, the C-G can assign commands on the same object to the same group and commands on different objets to different groups:

```
function C-G($cid, x$)
  return($(x \bmod k) + 1$)
```

Besides client proxy, each thread on server proxy also uses C-G function to determine the set of groups concerned by a delivered command. If commands are assigned to different groups they can execute concurrently, even if they modify the state of objects. Otherwise the execution of commands must be synchronized among threads. Moreover, additional information, if available, can be used when computing the C-G function. For example, objects that are commonly accessed could be assigned to different groups, allowing increased concurrency.

6.3.3 Protocol design

P-SMR takes as input the command dependencies (C-Dep) of a service and the desired multiprogramming level (MPL) at the replicas to define how independent commands can be executed concurrently and dependent commands are synchronized.

Basic principle. A client proxy executes command C by multicasting a request with C to a set of destination groups, computed by the C-G function. Worker threads at the server proxy deliver commands and invoke their execution against the local server. The execution of a worker thread alternates between two modes:

- The thread is in *parallel mode* when it delivers a command multicast to a single group. Upon delivering C, thread t_i executes C, sends C's response to the client and waits for the next command.
- The thread is in *synchronous mode* when it delivers a command multicast to multiple groups. Threads that deliver C, hereafter identified as τ,

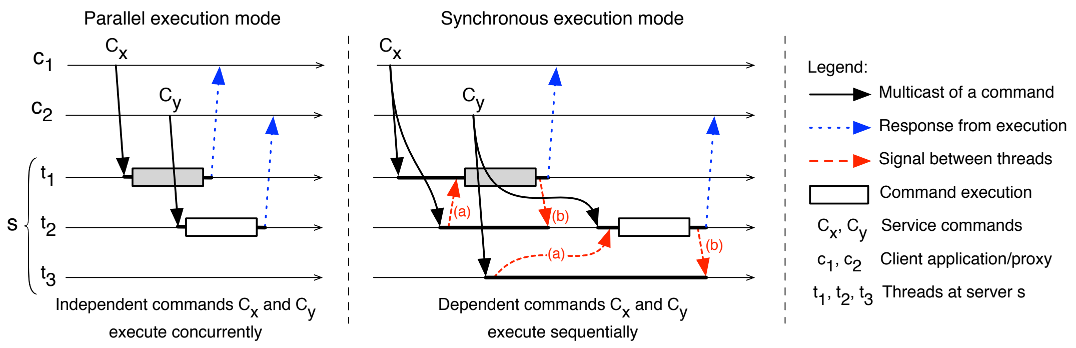

Parallel execution mode
Synchronous execution mode
c_1
c_2
s
t_1
t_2
t_3
C_x
C_y
(a)
(b)
Independent commands C_x and C_y execute concurrently
Dependent commands C_x and C_y execute sequentially
Legend:
Multicast of a command
Response from execution
Signal between threads
Command execution
C_x, C_y Service commands
c_1, c_2 Client application/proxy
t_1, t_2, t_3 Threads at server s

Figure 6.2. Two execution modes in P-SMR, parallel (left) and synchronous (middle). For clarity, we show the execution of clients c_1 and c_2 against a single server replica s with three worker threads, t_1, t_2 and t_3.

synchronize using barriers: threads in τ send a signal to one designated thread $t_i \in \tau$ (signal (a) in Figure 6.2) and wait for a signal from t_i; after t_i receives the signals it executes C, sends C's response to the client, and signals threads in τ to continue with the next command (signal (b) in Figure 6.2).

6.3.4 P-SMR: algorithm in detail

To execute command C, invoked by an application client (line 1 in Algorithm 1), the client proxy determines all groups γ involved in the command using the service's C-G function (line 2) and multicasts C and its input parameters to groups in γ (line 3). The client proxy then waits for the first response from the replicas (line 4), assigns the response received to the output parameters of C (line 5), and returns to the application (line 6). Upon delivering C (line 8), thread t_i at a server first uses the C-G function to determine the set of groups concerned by the command (line 9). If C was multicast to a single group, then t_i continues in parallel mode: t_i executes C (line 12) and returns the response to the client (line 13). If C was multicast to multiple groups, then t_i continues in synchronous mode and determines the thread t_e, among C's destinations, that will execute C (line 16). If t_i is in charge of executing C (lines 18–23), it waits for a signal from every other thread in C's destination set (lines 18–19), executes C (line 20), sends the response to the client (line 21), and then signals all other threads in C's destination set to continue their execution (lines 22–23). If t_i is not in charge of C's execution, it signals thread t_e (line 25) and waits for C's execution to complete (line 26).[4]

Algorithm 1: Parallel State-Machine Replication (P-SMR)

```
 1: A client proxy c executes a call to command C with identifier cid and input and output
    parameters as follows:
 2:   $\gamma \leftarrow C\text{-}G(cid, input)$          {$\gamma$ is the set of groups involved in C}
 3:   multicast($\gamma$, [$cid, input$])
 4:   wait for first response
 5:   output ← response
 6:   return
 7: Thread $t_i$ at a server proxy executes a command as follows:
 8:   upon deliver([$cid, input$]), multicast by c
 9:     $\gamma \leftarrow C\text{-}G(cid, input)$
10:     if $\gamma$ is a singleton then
11:       // Thread $t_i$ is in parallel mode
12:       execute cid with input and output parameters
13:       send response to c
14:     else
15:       // Thread $t_i$ is in synchronous mode
16:       $e \leftarrow min\{j : g_j \in \gamma\}$          {pick a thread deterministically}
17:       if $i = e$ then
18:         for each $j \neq i$ such that $g_j \in \gamma$
19:           wait for signal from $t_j$
20:         execute cid with input and output parameters
21:         send response to c
22:         for each $j \neq i$ such that $g_j \in \gamma$
23:           signal $t_j$
24:       else
25:         signal $t_e$
26:         wait for signal from $t_e$
```

6.4 Related Work

In Section 6.2 we have provided a thorough discussion on parallel state-machine replication and reviewed the related work. In this section we review general-purpose approaches that can be used to implement parallel replicas.

[4]For a proof of correctness see Appendix.

General-purpose approaches. Allowing multiple threads to execute commands concurrently may result in state and output inconsistencies if dependent commands are scheduled differently in two or more replicas. In [74; 75; 76; 77] the authors propose different approaches to enforcing deterministic multithreaded execution of commands. These solutions impose performance overheads and may require re-development of the service using new abstractions. Another solution is to allow one of the multithreaded replicas to execute commands nondeterministically and log the execution path, which will be later replayed by the rest of the replicas. Logging and replaying have been mainly developed for debugging and security rather than fault tolerance [78; 79; 80; 81; 82; 83; 84]. These approaches typically have high overhead due to logging and may suffer from inaccurate replay, leading to differences among original and secondary copies.

Using semantics to improve performance. Other works have proposed the use of application semantics to improve the performance of state-machine replication (e.g., [85; 86; 87]). These are based on the assumption that if two commands commute (e.g., incrementing a counter), then different replicas can execute them in different order and still reach the same final state. These works aim at reducing the delay to deliver a command by avoiding an expensive ordering protocol when possible.

6.5 Experimental evaluation

In this section, we first describe our implementations and outline several details about the experimental setup. We then introduce a key-value store as an application to compare P-SMR against SMR, sP-SMR, and two non-replicated single-server architectures: (a) a scheduler-worker server that uses a scheduling policy similar to servers in sP-SMR (hereafter, no-rep) and (b) a multithreaded server that relies on locks to synchronize the execution, without a scheduler (BDB). In no-rep and sP-SMR a scheduler at the server is responsible for scheduling incoming commands for execution at worker threads.

We configure Berkeley DB version 5.3 (BDB) to use the in-memory B-tree access method with transactions disabled and multithreading and locking enabled. Differently from P-SMR and sP-SMR and no-rep, BDB uses locks to synchronize the concurrent execution of commands. As a result, there is no scheduler between clients and server threads: each server thread receives requests through a separate socket, executes them, and responds to clients. We compare all these strategies for the following aspects:

- **Performance with independent-only workload.** Parallel execution of commands is only possible if commands are independent (i.e., they do not have any variables in common or if they do, they only read the values of the common variables). In Section 6.5.3 we perform an experiment to compare the performance of all the aforementioned techniques under workloads that are composed of independent-only commands and allow maximum amount of parallel execution.

- **Performance with dependent-only workload.** To execute dependent commands (i.e., commands that modify the values of the same variables), each technique uses a mechanism to synchronize the access of the concurrent threads to the common variables. In Section 6.5.4 we perform an experiment to both compare the performance of all the techniques with workloads that are composed of dependent commands only and also to implicitly compare the performance implications of various synchronization methods.

- **Performance with mixed workload.** P-SMR performs best in workloads dominated by independent commands. In the experiment of Section 6.5.5, we seek to determine "P-SMR's breakeven point": the percentage of dependent commands in the workload that make P-SMR neither better nor worse than SMR.

- **Scalability.** It is important to determine the scalability of each server model with the number of threads. To investigate this, in Section 6.5.6 we perform an experiment to evaluate the performance with an increasing number of threads and to measure the amount each thread contributes to the overall throughput.

- **Load balancing.** Due to the absence of a scheduler thread in P-SMR, it is not subject to the overhead of sequential delivery and scheduling. On the negative side, however, unlike sP-SMR, P-SMR has limited load balancing and can not evenly distribute load across worker threads. We perform an experiment in Section 6.5.7 to compare these two techniques under skewed workloads.

6.5.1 Hardware settings

We have performed all the experiments on a cluster with two types of nodes: (a) HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8GB of main memory, and (b) Dell SC1435 nodes equipped with two dual-core AMD Opteron processors running at 2.0 GHz and 4GB of main memory. The HP nodes are connected to an HP ProCurve Switch 2910al-48G gigabit network switch, and the Dell nodes are connected to an HP ProCurve 2900-48G gigabit network switch. Each node is equipped with two network interfaces. The switches are interconnected via a 20 Gbps link. The nodes ran CentOS Linux 6.2 64-bit with kernel 2.6.32. Clients were deployed on the Dell nodes and cceptors of Paxos and servers were deployed on the HP nodes.

6.5.2 Implementation and experimental setup

Key-value store. Our in-memory key-value store implements a B$^+$-tree where each entry has an 8-byte integer key, used as the tree index, and an 8-byte value. The store implements the following commands:

- `insert(in: int k, char[] v, out: int err)`. An insert adds a new entry with key k and value v to the tree and possibly returns an error code (e.g., out of memory).
- `delete(in: int k, out: int err)`. A delete removes the entry corresponding to k from the tree or returns an error code if the entry does not exist.
- `read(in: int k, out: char[] v, int err)`. A read returns the value of k. An error code is returned if the key does not exist.
- `update(in: int k, char[] v, out: int err)`. An update replaces the current value of k with v. An error code is returned if the key does not exist.

While a read does not result in any changes in the tree, an update changes a single entry, the one corresponding to the provided key (if present). Inserts and deletes may modify multiple entries, depending on the structure of the tree when the command is executed (i.e., requiring partitioning and joining of tree cells). We define the following dependencies between commands: inserts and deletes depend on all commands; an update on key k depends on other updates on k, on reads on k, and on inserts and deletes.

To generate enough load to reach maximum performance, each client maintains a window of outstanding requests that can contain up to 50 commands. The tree is initialized with 10 million keys on each replica and unless specified otherwise, clients select the keys uniformly. Except for the no-rep and BDB techniques, in which there is only one replica, the key-value store is fully replicated on two replicas.

sP-SMR. We next review some implementation details of the sP-SMR model. Server and client proxies in sP-SMR are similar to P-SMR except for the following differences. First, at the server proxy, sP-SMR differentiates between two types of threads: scheduler and worker. The scheduler thread sequentially delivers the commands and dispatches them among the worker threads. Worker threads are the threads that execute the commands and respond to clients. In a configuration where MPL is set to W, we identify worker threads as t_1,..., t_W and the scheduler thread as t_s. Second, sP-SMR uses an atomic broadcast primitive rather than an atomic multicast primitive to order the requests and deliver them to the replicas. Therefore, a client proxy does not need to map commands to groups as in the P-SMR model. As a consequence, unlike P-SMR, client proxies are oblivious to the command dependencies and the C-Dep structure. However, C-Dep is used by the scheduler thread at the server proxy to parallelize command execution.

Similar to P-SMR, sP-SMR takes as input the command dependencies (C-Dep) of a service to decide on the concurrent execution of independent commands. A client proxy executes command C by atomically broadcasting a request including C to all replicas. The scheduler thread at the server proxy delivers commands and detects their interdependencies to appropriately disseminate them among the worker threads. Worker threads scan their queues and invoke the execution of requests against the local server. The execution at the server proxy alternates between two modes:

- The execution is in *parallel mode* when the scheduler thread delivers an independent or a dependent command that does not require global synchronization. Upon delivering C, the scheduler thread t_s, selects a worker thread t_w, to execute C. The scheduler inserts C in t_w's queue. t_w executes C, sends C's response to the client, and waits for the next command.

- The execution is in *synchronous mode* when the scheduler thread delivers a dependent command that needs global synchronization. After delivering a dependent command C, the scheduler thread first waits for all the worker threads to finish the execution of all the commands in their queues and then selects a worker thread t_w to execute the dependent command. All the other worker threads remain idle while t_w is executing C. The scheduler thread can deliver other requests meanwhile, but dispatches them only after the execution of C is finished. Thread t_w executes C, sends C's response to the client, and signals the scheduler thread to continue with the next command.

The scheduler thread shares with each worker thread a queue. Therefore, independent of the workload, the insertion of the commands in the queues of the worker threads requires some synchronization mechanism, such as locking, to prevent anomalies that can happen due to the concurrent enqueueing and dequeueing of the commands. Worker threads do not communicate with each other and do not share any information.

Atomic multicast. For multicast library we use Multi-Ring Paxos protocol from Chapter 5. As a reminder, Multi-Ring Paxos implements the abstraction of groups by composing multiple parallel instances of M-Ring Paxos whereas each multicast group is mapped to one or more M-Ring Paxos instances. A message can be addressed to a single group only, and not to multiple groups. In our P-SMR prototype, each thread t_i belongs to two groups: one group, g_i, to which no other thread in the server belongs, and one group g_{all}, to which every thread in each server belongs. Threads deliver messages from multiple streams and use the deterministic merge mechanism of Multi-Ring Paxos to ensure ordered delivery. This is enough to implement both C-G functions presented in Section 6.3.3. Commands multicast to a group are batched by the group's coordinator (i.e., the coordinator in the corresponding M-Ring Paxos instance) and order is established on batches of commands. Each batch has a maximum size of 8 Kbytes. The system was configured so that each M-Ring Paxos instance uses 3 acceptors and can tolerate the failure of one acceptor. For sP-SMR and SMR techniques that demand total ordering we also use Multi-Ring Paxos where all the replicas subscribe to all the groups and the deterministic merge guarantees the total order delivery.

6.5.3 Performance of independent commands

In this section we perform a set of experiments to compare all the techniques with workloads that allow maximum parallelism (Figure 6.3). We use a work-

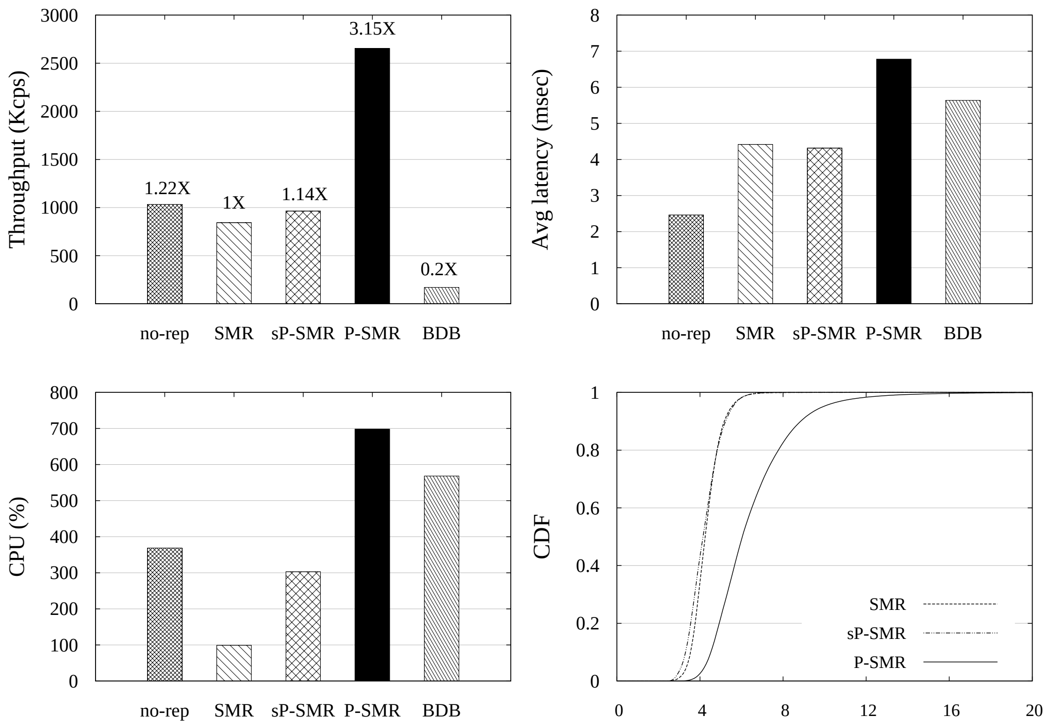


Figure 6.3. Performance of independent commands; throughput in Kilo commands executed per second (Kcps) (top-left); CPU usage (bottom-left); average latency in milli seconds (top-right); CDF of latency (bottom-right).

load composed of read commands only. The values we report correspond to the peak throughput of each technique and are obtained with 8 threads for P-SMR, 2 threads for sP-SMR and no-rep, 1 thread for SMR, and 6 threads for BDB. In the case of norep and sP-SMR, the number of threads excludes the scheduler.

The throughput of P-SMR is about 3.15 and 2.75 times higher than SMR and sP-SMR, respectively (Figure 6.3). The scheduler in sP-SMR and no-rep becomes CPU-bound and caps performance. The throughput of SMR is limited by what a single thread can achieve, whereas no-rep is multithreaded and achieves higher throughput. The throughput of no-rep is slightly higher than sP-SMR as no-rep does not rely on atomic multicast. BDB has the lowest throughput due to high overhead with locking, reflected in the CPU usage. Latency of P-SMR is the highest at peak throughput. Although not shown in the figure, under similar throughput P-SMR has latency comparable to the other techniques. Latency of no-rep is lower than all other techniques as it is not subject to the overhead

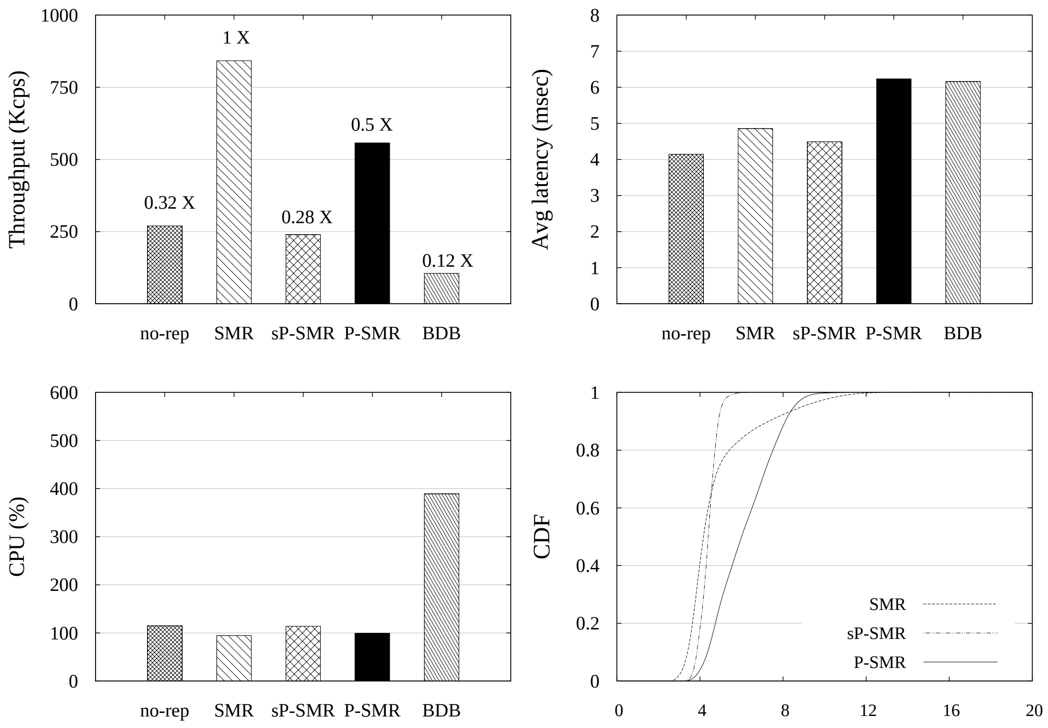


Figure 6.4. Performance of dependent commands; throughput in Kilo commands executed per second (Kcps) (top-left); CPU usage (bottom-left); average latency in milliseconds (top-right); CDF of latency (bottom-right).

of multicast library. Latency of sP-SMR is affected by both the overhead of the ordering and scheduling and is higher than the latency of no-rep.

6.5.4 Performance of dependent commands

In this experiment we study the performance of all the techniques under workloads that induce maximum synchronization among threads (Figure 6.4). To this end, we determine the maximum throughput of the key-value store service when commands are inserts and deletes. The values are obtained with 4 threads for BDB and with 1 thread for all the other techniques. In case of norep and sP-SMR the number of threads excludes the scheduler. These are the configurations that correspond to the peak throughput of each technique (for the performance of dependent commands under different number of threads see Section 6.5.6). SMR is not subject to synchronization overhead, which allows it to reach the highest throughput (Figure 6.4). Moreover, throughput in SMR remains con-

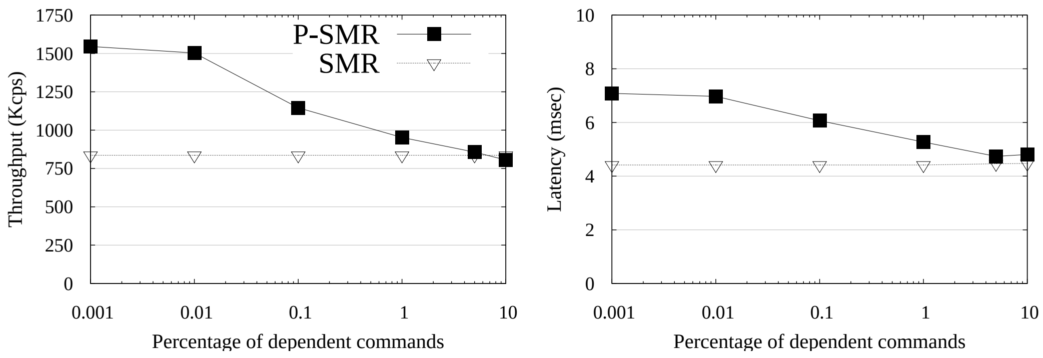


Figure 6.5. Performance of mixed workloads (both independent and dependent commands); throughput measured in Kilo commands executed per second (Kcps) (left); the average latency measured in milliseconds (right); x-axis is in log scale.

stant at about 842 Kcps, both with independent and dependent commands; in BDB the throughput decreases from 140 Kcps to 105 Kcps. P-SMR's latency is higher than SMR's and sP-SMR's. The long tail in the CDF graphs suggests that P-SMR's latency is subject to more variation than SMR's and sP-SMR's.

6.5.5 Performance of mixed workloads

We now assess the performance of P-SMR under workloads with a mix of dependent and independent commands (Figure 6.5). To this end, we measure the maximum throughput of the key-value store service with workloads composed of inserts, deletes, and reads. The x-axis shows the percentage of dependent commands (inserts and deletes) with respect to all the commands in the workload. P-SMR uses 8 worker threads in this experiment. We compare the performance of P-SMR to SMR, the only approach that is not subject to synchronization overhead and therefore has the highest performance under dependent commands.

SMR's throughput remains constant with the mixed workload. This is expected since most of the cost to execute a read, insert, and delete operation is related to traversing the tree (statistics gathering starts after the tree is initialized; thus, few inserts and deletes involve changes in multiple levels of the tree). P-SMR's throughput is above SMR's up to about 10% of dependent commands. The reduction in performance is due to synchronization overhead. P-SMR's latency decreases as the percentage of dependent commands increases.

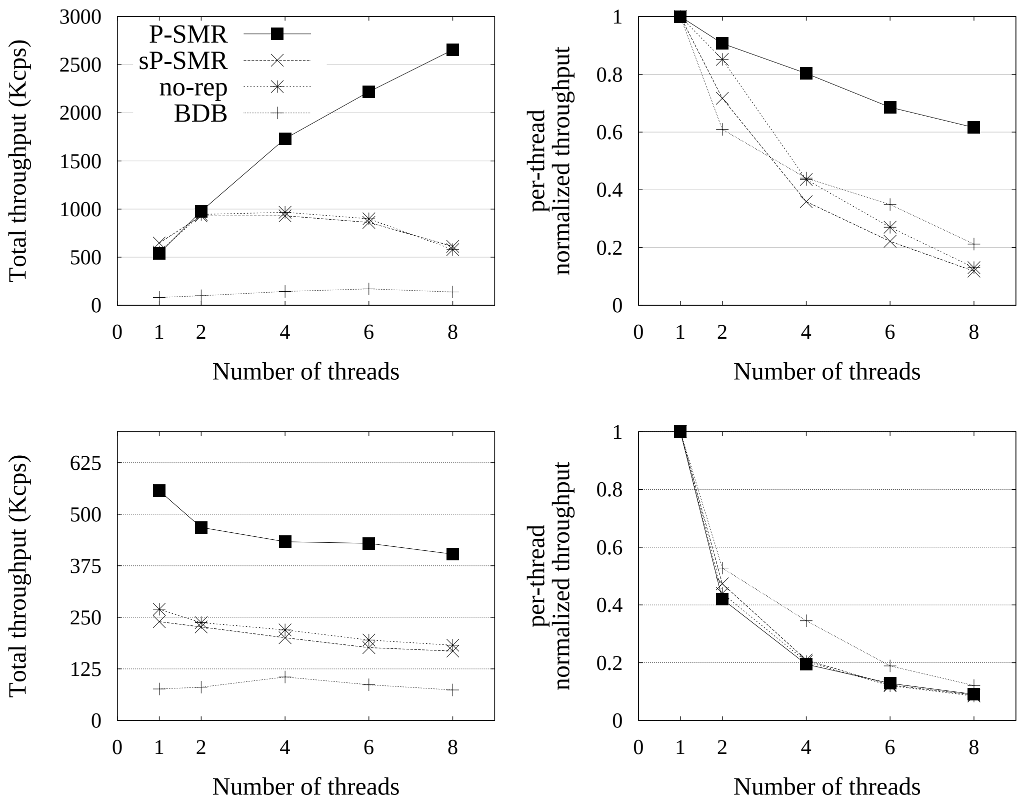


Figure 6.6. The effect of the number of threads on the performance of independent commands (top) and dependent commands (bottom); maximum throughput in Kilo commands executed per second (Kcps) (left graphs); normalized per-thread throughput (right graphs).

The decrease in latency corresponds to a reduction in throughput.

6.5.6 Scalability

In this experiment we assess the scalability of sP-SMR, P-SMR, no-rep, and BDB with respect to the number of threads (Figure 6.6). We measure the maximum throughput of the key-value store service while the number of threads changes from one to eight when commands are independent (top graphs) and when dependent (bottom graphs). In sP-SMR, the number of threads reflects the number of worker threads excluding the scheduler. We report absolute values for the peak throughput and also the normalized throughput of an individual thread. If perfectly scalable, the throughput of each thread must remain constant as worker threads are added.

With independent commands only, the throughput of all the techniques, except for BDB, compare equally with one thread. As threads are added, the throughput of all the techniques, except for P-SMR, decreases. For sP-SMR and no-rep this happens due to scheduling overhead at the scheduler. P-SMR has better scalability than the other techniques (see top left graph). With dependent-only commands, in all the approaches, except BDB, throughput decreases with the number of worker threads. The throughput of BDB increases up to 4 threads and then it also decreases.

6.5.7 Performance of skewed workloads

In this experiment, we compare the effect of skewed workloads on the performance of P-SMR and sP-SMR (Figure 6.7). The workload is composed of 50% updates and 50% reads against the key-value store. We evaluate the scalability of each approach with both uniform and Zipfian distributions for key selection. In the latter case, clients select keys following a Zipfian distribution with exponent value of one. In skewed distributions, communication is expected to be uneven across multicast groups. In sP-SMR, the number of threads reflects the number of worker threads excluding the scheduler. Besides absolute values for the maximum throughput, we also show the normalized throughput of an individual thread.

With a uniform selection of keys, commands are evenly distributed across groups and P-SMR's throughput increases up to the capacity of each available core. With a Zipfian distribution, however, P-SMR's throughput is bounded by the most-loaded multicast group (point with 8 threads). sP-SMR is not bounded by a single multicast group as is P-SMR, but rather by the load the scheduler can handle until it becomes CPU-bound. Increasing the number of worker threads after two threads has a negative impact on sP-SMR's performance since the scheduler spends more time synchronizing with worker threads. Also notice that with 1 and 2 threads the throughput of sP-SMR with a uniform workload is lower than its throughput with a Zipfian distribution. In the Zipfian distribution, some keys are accessed more often than the others and there are higher chances that these keys are cached at the processor. According to the normalized per-thread throughput, P-SMR scales better with the number of cores than sP-SMR under both uniform and Zipfian distributions.

6.5.8 Conclusions from the experiments

We draw the following main conclusions from the experiments:

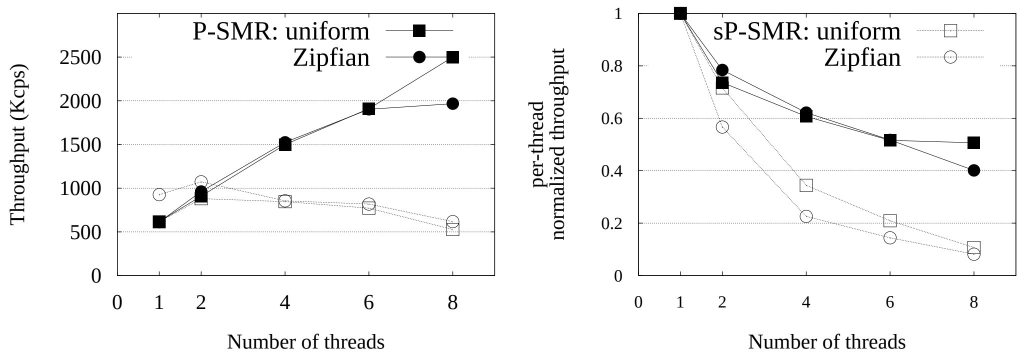


Figure 6.7. The effect of the number of threads on the performance of a skewed workload; maximum throughput in Kilo commands executed per second (Kcps) (left); normalized per-thread throughput (right).

- P-SMR is optimized for workloads that mostly include independent commands and as the experiments illustrated, P-SMR outperforms other sequential and parallel approaches with this type of workload.

- When the workload is dominated by dependent commands, a sequential implementation of state-machine replication outperforms the parallel implementations. This is a result of synchronization among the threads in parallelized approaches.

- Although P-SMR lacks the flexibility of sP-SMR's model in distributing the load among the worker threads, we have seen that with skewed workloads P-SMR has still a higher throughput. This is because sP-SMR's performance is limited by the overhead of the tasks the scheduler does before its advantages in load balancing reveal themselves.

6.6 Conclusion

In this chapter, we have questioned the sequentiality of state-machine replication and the possibility of its integration with parallel services. State-machine replication is widely used in making systems fault tolerant and parallel systems are no exception. We have designed and implemented P-SMR, a new parallelized model for state-machine replication. In summary, we have found that for independent commands, P-SMR outperforms sequential state-machine replication by a factor of more than 3 and other approaches by a factor of more than 2. For dependent commands, although P-SMR has better performance than other parallel techniques, it is defeated by SMR, the sequential implementation of state-machine replication that is not subject to the overhead of synchronization. Moreover, P-SMR scales better with the number of cores whereas other techniques show poor scalability.

Chapter 7

Experimenting with Paxos in the Cloud

Paxos is often present at the core of state-machine replication and has been the most central component of this study. Implementations of Paxos are currently used in many prototypes and production systems in both academia and industry. Given its wide usage, Paxos's performance and behavior in the presence and absence of failures is critical to the overall performance of a system built on top of it. In this chapter, we present the results of an extensive performance evaluation conducted using four open-source implementations of Paxos deployed in Amazon's EC2. Although all protocols surveyed in this chapter implement Paxos, they are optimized in a number of different ways, resulting in very different behaviors. In addition to reporting our findings in a variety of configurations with and without failures, we propose and assess additional optimizations to existing implementations.

7.1 Problem Statement

In this chapter we present the results of an extensive performance evaluation we conducted with open-source implementations of Paxos [4] deployed in Amazon's EC2, a public cloud-computing environment.[1] Our study is motivated by the fact that many online services deployed in the cloud require both high availability and sustained performance. High availability is achieved by means of replication, using techniques such as state-machine replication [68] and primary-backup replication [88]. At the core of these techniques lies an agreement protocol (e.g., [4; 89]). A large variety of such agreement protocols exist in the literature that solve the problem under many different system assumptions [**?**].

[1]http://aws.amazon.com/ec2/

Among these protocols, Paxos has received much attention in recent years both from the industry (e.g., [87; 16]) and the academia (e.g., [10; 90]). Several recent efforts (including the first three chapters of this thesis) have reported on the performance of Paxos implementations, mostly under "normal conditions" (e.g., [10; 32]), that is, deployments with homogenous nodes, balanced communication links, and the absence of failures. While differences in the implementations impact overall performance, these reports typically show steady behavior in the normal case. Yet, anecdotal evidence tells that under less favorable conditions (e.g., after the failure of a node), Paxos may lose its sustained performance. Intuitively, this is explained by the fact that by relying on a quorum of acceptors for progress, Paxos may proceed at the pace of the quorum of faster acceptors, leaving slower acceptors lagging behind with an ever-increasing backlog of requests. Paxos implementations generally read and process messages in arrival order, hence even if the messages in question relate to protocol actions that have been completed long time before, they will be read and processed just as if they are associated with pending decisions. All of this will take time, hence should a fast acceptor fail and a slow one be needed to form a quorum, the system may experience a performance hiccup, corresponding to the time it takes for the slower acceptor to catch up. Bursty behavior is undesirable because it can cascade into the application, within which end-user requests may be piling up and replicas falling behind.

We set out to understand to what extent existing implementations genuinely suffer from this phenomenon and if so, under what conditions. To this end we evaluated four open-source implementations of Paxos: S-Paxos, OpenReplica, U-Ring Paxos, and Libpaxos under different message sizes in four configurations: (a) a homogeneous set of nodes in the same availability zone (i.e., data center); (b,c) two heterogeneous configurations with nodes in the same availability zone; and (d) homogeneous nodes distributed across different availability zones. In each case, we considered executions with and without participant failures. These configurations represent the deployment of many current online services in the cloud. Placing replicas on a set of nodes with similar hardware characteristics (configuration (a)) is probably the most common configuration used in experimental evaluations. Heterogeneous settings (configurations (b) and (c)) may arise involuntarily (e.g., if applications run in a virtual machine whose physical node turns out to be shared among other applications) or voluntarily: a designer might choose to deploy Paxos in this manner, perhaps to reduce the perceived risk of correlated failures, or to reduce cost, for example by paying for 3 powerful nodes and 1 or 2 weaker backup nodes (e.g., Cheap Paxos [14] is a variation of Paxos that exploits this alternative). In addition to these two configurations, the participants of a service can be geographically distributed (configuration (d)) to improve locality and availability. Locality reduces user-perceived latency and is achieved by moving the data closer to the users. Availability improves as the service can be configured to tolerate the crash of a few nodes within a data center or the crash of multiple data centers.

By evaluating the four open-source Paxos libraries under these configurations, we show that standard Paxos implementations sometimes have unexpected behavior and long delays, although the phenomenon varies and depends very much on the details of the implementations: some protocols are more prone to problematic behavior; others are more robust but at the price of reduced performance.

7.1.1 Outline

The remainder of this chapter is organized as follows. In Section 7.2 we describe the libraries used in our performance evaluation with emphasis on their flow control mechanisms. In Section 7.3 we detail our experimental setup and present the results. In Section 7.4 we discuss the main lessons we have learnt while interacting with these libraries and we conclude this chapter in Section 7.5.

7.2 Open-source Paxos libraries

In our evaluation, we worked with four open-source Paxos implementations. Recalling from Chapter 3, Paxos requires a majority-quorum for progress (i.e., it remains operational despite the failure of f acceptors out of $2f+1$). As soon as a participant (e.g., leader) receives a majority of Phase 2B messages for a value in an instance, the participant knows the instance is decided. We call this quorum the participant's *first majority-quorum*. Different participants may have distinct first majority-quorums, but if an acceptor is “slow”, then it is unlikely to participate in any first majority-quorum. In fact, one can expect that a first majority-quorum will likely contain “fast” acceptors only.

An acceptor can be slow for many reasons. For example, perhaps the slow acceptor cannot keep up with the fast acceptors because it is running on a node with less processing power than the fast acceptors or its CPU is shared among several processes. It could also be that its communication links are subject to higher latencies than the other nodes' links. Whatever the reason, the notions of slow and fast acceptors are important because Paxos is quorum-based, moving from one consensus instance to the other as soon as a majority of acceptors is prepared to do so. In the following, we argue that in principle such a distinction between acceptors may have performance implications, notably in the case of failures. In the subsequent section, we assess this phenomenon experimentally.

7.2.1 S-Paxos

S-Paxos [32] is implemented in Java[2] and is composed of a set of replicas, each one playing the combined roles of proposer, acceptor, and learner. One of the replicas is elected as the leader. The key idea in S-Paxos is to load-balance request reception and dissemination among all the replicas. A client selects an arbitrary replica and submits its requests to it. After receiving a request, a replica forwards it (or possibly a batch of requests) to all the other replicas. A replica that receives a forwarded request sends an acknowledgement to all the other replicas. When a replica receives $f + 1$ acknowledgements, it declares the request stable. This is needed because in S-Paxos ordering is performed on request ids. As in classic Paxos, the leader is responsible for ordering requests. A participant considers an instance decided after receiving $f+1$ Phase 2B messages from the acceptors. All the replicas execute all the requests but only the replica who receives the request responds to the client. S-Paxos strives to balance CPU and network resources, but many messages must be exchanged before a request can be ordered. Due to the high number of messages exchanged, S-Paxos is CPU-intensive and benefits from deployment on powerful multi-core machines.

S-Paxos uses blocking I/O for the communications among replicas. As mentioned earlier, a replica forwards batches of requests to all the other replicas. If a replica is slow in handling its incoming traffic, another replica will block upon sending new messages to the slow replica since communication is based on TCP. Thus, faster acceptors cannot transfer more batches to the slow replica and we expect the performance of the system to follow the speed of the slowest replica. Moreover, since S-Paxos is designed around the idea of distributing the load among acceptors, reducing the number of acceptors (e.g., due to failures) may result in reduced performance.

7.2.2 OpenReplica

OpenReplica is an open-source library implemented in Python[3] that enables automatic replication of user-provided objects [91]. OpenReplica is composed of a

[2]https://github.com/nfsantos/S-Paxos

[3]https://pypi.python.org/pypi/concoord

set of replicas and a set of acceptors. Replicas are the processes that replicate an object and in Paxos's parlance, they play the "learner" role. One of the replicas is also the leader in Paxos. In OpenReplica, clients send their requests to a client proxy who batches the requests. The client proxy then connects to the pool of replicas to send the batched requests; OpenReplica ensures that the requests are forwarded to the leader to be ordered. Replicas deliver and execute the sequence of requests in the order dictated by instance identifiers. After executing a request, replicas respond to the clients.

The leader in OpenReplica uses non-blocking I/O to communicate with the acceptors. If the transmission of a message to an acceptor cannot happen immediately (e.g., because the communication buffer associated with the acceptor is full), the leader is notified and retries the transmission until it succeeds. If an acceptor is slower than the others, its buffers will fill up faster and communications with it will cause retransmissions at the leader. This affects performance because the leader will work harder and some portion of its I/O bandwidth will be lost to retransmissions. If during the time it takes for the slow acceptor to catch up a fast acceptor crashes, we can expect a further reduction in performance since a majority-quorum of acceptors will not be available immediately given that the slow acceptor is needed to form a majority-quorum.

7.2.3 U-Ring Paxos

U-Ring Paxos is an open-source implementation of U-Ring Paxos in Java.[4] (There is also a C implementation of M-Ring Paxos[5] that relies on ip-multicast; we use the Java version, which is based entirely on unicast communication and is the most suitable for Amazon's infrastructure. See Chapter 3 for detailed explanations of M-Ring Paxos and U-Ring Paxos protocols). As presented in Chapter 3, U-Ring Paxos disseminates all the processes on a logical uni-directional ring to make a balanced usage of the available bandwidth. In the implementation considered, a process of U-Ring Paxos can play the roles of acceptor, proposer, learner, and coordinator. One of the acceptors is elected as the leader. U-Ring Paxos handles leader election and the ring's configuration via Zookeeper.[6] Clients submit their requests to those processes in the ring that assume the role of proposers. Proposers batch the requests and forward them along the ring. The leader initiates Paxos for the batches of requests that it assembles and the batches it receives from other processes in the ring. Acceptors create Phase 2B

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

[5]http://sourceforge.net/projects/libpaxos/files/RingPaxos/

[6]http://zookeeper.apache.org/

messages and send them to their successors. Processes that are not acceptors simply forward Phase 2B messages they receive to their successors. The final decision is made by the acceptor that receives $f + 1$ Phase 2B messages. The decision circulates in the ring until all processes receive it. The learners deliver instances following instance identifiers.

Processes in the ring communicate using TCP; learners send replies to the clients through UDP. All the communications is based on non-blocking I/O. Both clients and processes in the ring can batch messages. In a ring, a slow process can negatively affect the overall performance as it may become a system's bottleneck. We expect the ring to operate at the speed of the slowest acceptor. If an acceptor leaves the ring, U-Ring Paxos will reconfigure the ring and during reconfiguration performance may suffer.

7.2.4 Libpaxos

Libpaxos is implemented in C.[7] It distinguishes proposers, acceptors, and learners, where proposers are also learners. Libpaxos does not handle leader election. Applications must decide how to ensure the existence of a single leader (e.g., one option is to use Zookeeper). To submit requests, clients connect directly to the proposers. Acceptors send their Phase 2B messages, including the agreed value, to the proposers and to the learners. Upon receiving $f + 1$ Phase 2B messages from the acceptors, the learners and the proposers declare an instance as decided. The learners deliver instances following instance identifiers.

Processes in Libpaxos communicate using non-blocking buffered I/O provided by the libevent library.[8] Libpaxos does not explicitly batch the requests; batching is implemented by the buffered communication provided by libevent. Besides sending Phase 2B messages, an acceptor also sends values to the learners and proposers, therefore, the acceptor's outgoing traffic is higher than its incoming traffic. A slow acceptor may become overwhelmed by a high volume of incoming messages, in which case messages will pile up at the sender's side, or by a high rate of outgoing messages, in which case messages will pile up at acceptor's side. In either case, until a slow acceptor becomes overwhelmed, performance in Libpaxos will be driven by the faster acceptors. If a fast acceptor crashes and a slow acceptor is needed to form a majority quorum, the system may experience periods of inactivity until the slow acceptor processes its backlog of requests.

[7]https://bitbucket.org/sciascid/libpaxos
[8]http://libevent.org

7.2.5 Libpaxos$^+$

Motivated by our observations on the behavior of Libpaxos, presented in Section 7.3, we created Libpaxos$^+$, an extension to the original protocol. The key idea is for proposers to selectively involve acceptors in Paxos's Phase 2 based on how the acceptors performed in previous instances. If an acceptor was not in the first majority-quorum of past instances, then it might be a slow acceptor and should be spared in the next few instances. Libpaxos$^+$ thus attempts to reduce the backlog of slow acceptors in order to allow them to catch up, so that they can achieve better response times later in instances in which they participate.

We modify a proposer so that its execution is divided into *steps*, where a step is a sequence of Paxos instances. In the first instances of a step, the proposer sends Phase 2A messages to all acceptors and records the number of instances each acceptor is included in the first majority-quorum. In the next instances in the step, the proposer sends Phase 2A messages to a majority-quorum only, composed of those acceptors who appeared most often in the initial instances. A step finishes when a pre-determined number of instances are executed or the proposer suspects the crash of an acceptor among the selected ones to participate in the instance.

7.3 Experimental evaluation

In this section, we describe the experimental setup, explain our methodology for the experiments, report on the peak performance of each library under various conditions, and analyze each library under failures.

7.3.1 Experimental setup

Hardware setup. All the experiments are performed in Amazon's EC2 infrastructure with a mix of small, micro, and large instances, as detailed next. In all the experiments each process runs on a separate Amazon EC2 instance. In the following, one EC2 compute unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor, and vCPU represents the number of virtual CPUs for the instance.

- Micro: up to 2 ECUs (EC2 Compute Unit), 1 vCPUs, 0.613 GBytes memory, very low network capacity.
- Small: 1 ECUs, 1 vCPUs, 1.7 GBytes memory, 1x 160 GBytes of storage, low network capacity.

- Large: 4 ECUs, 2 vCPUs, 7.5 GBytes memory, 2x 420 GBytes of storage, moderate network capacity.

All servers run Ubuntu Server 12.04.2-64 bit and the socket buffer sizes are equal to 16 MBytes.

Configurations. We measure the performance of S-Paxos, OpenReplica, U-Ring Paxos, and Libpaxos in four different configurations (see Table 7.1). In S-Paxos and U-Ring Paxos one of the acceptors plays the role of the leader and thus in Table 7.1 the leader represents acceptor A1 for S-Paxos and U-Ring Paxos. For each configuration, all the libraries are evaluated with three request sizes: 200 Bytes, 4 KBytes, and 100 KBytes. All the libraries are in-memory in our experiments. U-Ring Paxos relies on Zookeeper for ring configuration. Session timeout for Zookeeper is set to 3 seconds in all the experiments.

Configuration	Type	Environment	Leader*	A1	A2	A3	Learner†
(a)	Homogeneous	LAN	Small				
(b)	Heterogeneous	LAN	Small			Micro	Small
(c)	Heterogeneous	LAN	Large	Small		Micro	Small
(d)	Homogeneous‡	WAN	Small				

Table 7.1. Configurations used in the evaluations. (Legend: *The leader in S-Paxos and U-Ring Paxos is also acceptor A1 and the leader in OpenReplica is also replica. †The concept of an independent learner only exists in Libpaxos and U-Ring Paxos. ‡ Although machines in this configuration are homogenous, the acceptor in the remote data center, A3, is connected to other processes with a lower bandwidth.)

In the experiments performed in a LAN, all the instances are deployed in the US-West-2c region. In the experiments performed in a WAN, processes are distributed among three availability zones: In the experiments with Libpaxos and OpenReplica, the leader, A2, the learner, and clients are located in US-West-2c, A1 is located in US-West-2b, and A3 is located in US-East-1b. In the experiments with S-Paxos and U-Ring Paxos, the leader, A1, the learner (in U-Ring Paxos), and the clients are located in US-West-2c, A2 is located in US-West-2b, and A3 is located in US-East-1b. Moreover, in all the experiments with U-Ring Paxos, a stand-alone version of Zookeeper is deployed on a micro instance located in US-East-1c. As a reference, the RTT value is 1.5 ms (millisecond) in US-West-2c,

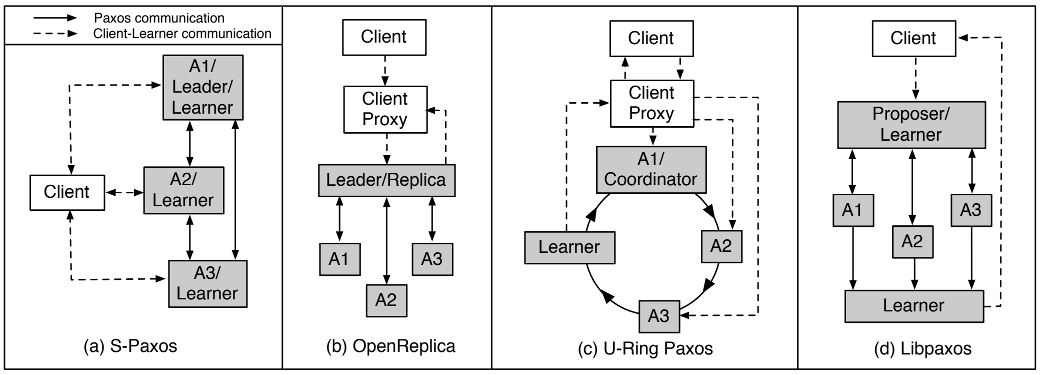


Figure 7.1. The communication pattern and architectural differences among the four libraries as deployed in the experiments (f is equal to one). In all the libraries the learner/replica sends the responces back to the client; in S-Paxos and OpenReplica clients send their requests to the nodes that also assume the learner/replica role.

3.9 ms between US-West-2b and US-West-2c, 82 ms between US-West-2c and US-East-1b, and 90 ms between US-West-2b and US-East-1b.

Architectural differences. Figure 7.1 illustrates the inter-process communication patterns and architectural differences of the four libraries while preserving their specific terminology (for the details see Section 7.2). Notice that the terms *leader*, *proposer*, and *coordinator* convey the same concept and so do the terms *replica* and *learner*. In U-Ring Paxos, however, proposer refers to any node that receives requests from clients and forwards them to other processes. In all the experiments, there are three acceptors in all the libraries and f is equal to one. In S-Paxos and U-Ring Paxos, the leader (or the coordinator) role is assumed by one of the acceptors (acceptor A1). In OpenReplica and Libpaxos, a separate process is elected as the leader (or proposer).[9]

In the experiments with OpenReplica, each client process has a client proxy (as a separate module in the client process) that batches the requests of the client and sends them to the leader. The client waits for the responses before submitting new requests. Similarly to OpenReplica, in U-Ring Paxos a client has a module for batching the requests. The client proxy batches the requests and sends them to an acceptor that also plays the role of proposer. The size of a batch

[9]We emphasize that the differences in the deployments are due to the unique properties of the libraries rather than the choices made by the author.

is 12 KBytes in all the experiments. Batching in U-Ring Paxos has been disabled throughout the experiments. In the experiments with S-Paxos, a client sends a request to a randomly chosen replica and waits for its response before sending a new request. A replica batches requests before disseminating them to the other replicas. In our experiments, this batch size is 1 KByte. Also note that the batch sizes in these two libraries are chosen to get the best performance. In the experiments with Libpaxos, to ensure progress we have configured a single proposer. Clients send a request to the proposer and wait for the request's response from the learner before sending a new request.

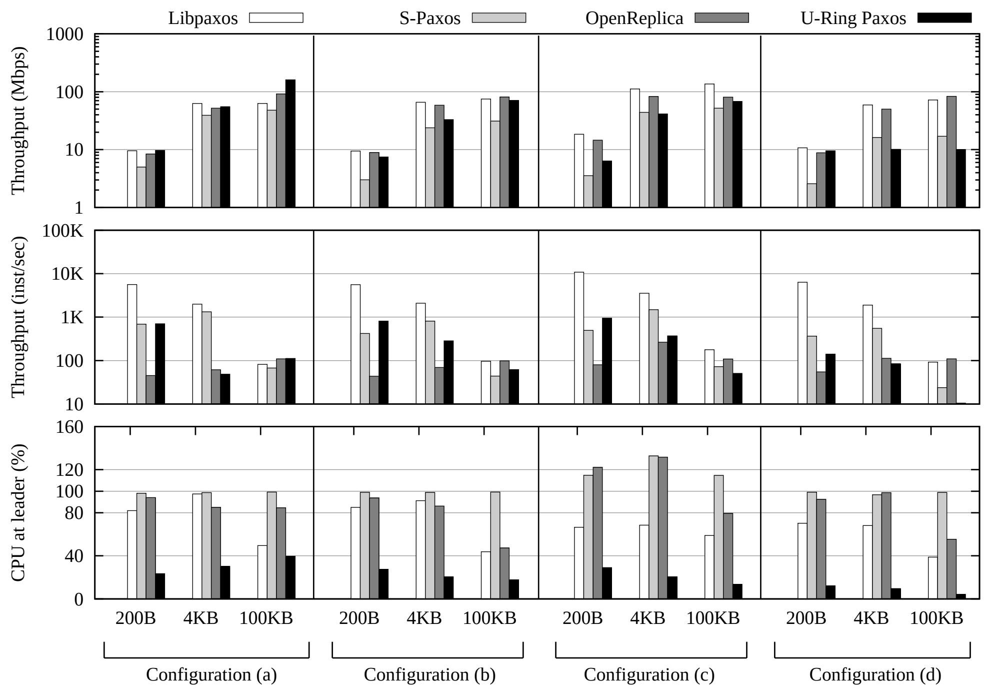


Figure 7.2. Peak performance of Libpaxos, S-Paxos, U-Ring Paxos, and OpenReplica in four configurations (see Table 7.1); the y-axis in the two top-most graphs is in log scale.

Library	[Conf. ,Size]
S-Paxos	[(b),4] [(d),4]
OpenReplica	[(c),100] [(d),4]
U-Ring Paxos	[(b),100] [(d),4]
Libpaxos	[(b),4] [(d),4]

Table 7.2. Configurations in which we evaluate the flow control mechanism of the open-source libraries.

7.3.2 Methodology

The goal of our performance assessment is twofold: First, we measure the peak performance of S-Paxos, OpenReplica, U-Ring Paxos, and Libpaxos in a set of configurations as illustrated in Table 7.1 (Section 7.3.3). Second, we select a subset of these configurations to take a closer look at the flow control mechanisms of the libraries (Sections 7.3.4, 7.3.5, 7.3.6, 7.3.7). Since libraries are different in their implementation and communication strategies, the configurations we choose vary across libraries. Table 7.2 enlists the set of the chosen configurations.

7.3.3 Peak performance

Figure 7.2 displays the results for peak performance. The graphs in this figure measure the following performance metrics from top to bottom: delivery throughput in megabits per second, delivery throughput in number of decided instances per second, and CPU usage at the leader. The number of requests delivered per second is directly proportional to the delivery throughput in megabits per second. Each experiment is performed for a period of 100 seconds and the results from the first and last 10 seconds are discarded. S-Paxos, U-Ring Paxos, and OpenReplica are multithreaded and therefore in some scenarios (configuration (c)) the CPU usage at the leader is higher than 100%. When comparing performance, unless stated otherwise, the values of throughput in Mbps are considered (top-most graph). The following patterns can be discerned from Figure 7.2.

- For all implementations and configurations, throughput improves as the request size increases, although the improvement is more noticeable from small to medium messages.

- In most of the configurations, OpenReplica and Libpaxos show similar performance, better than S-Paxos and U-Ring Paxos's performance.

- When we replaced one of the small acceptors in configuration (a) with a slower acceptor (see configuration (b) in Table 7.1), performance of Libpaxos and OpenReplica does not change between configurations (a) and (b), regardless of request size since their execution is driven by the fastest majority-quorum and in both configurations there is a majority-quorum that contains two small acceptors. The throughput of S-Paxos and U-Ring Paxos in configuration (b) is lower than in configuration (a) since S-Paxos and U-Ring Paxos adapt their performance to the speed of the slowest member.

- When we placed the leader in configuration (b) in a more powerful node, configuration (c), we observed that in all protocols except U-Ring Paxos the throughput increased, regardless of the request size. The leader of U-Ring Paxos is not CPU-bound in configuration (b) and therefore replacing the leader by a stronger machine had no effect in performance.

- To understand the effects of geographical deployments on the performance, compare the results in configurations (a) and (d). The performance of Libpaxos and OpenReplica do not change between the two configurations: in both cases there is a majority-quorum in the vicinity of the proposer (leader) in the two libraries. Hence, Libpaxos and OpenReplica are not limited by the slow links between the proposer and the acceptor located in a remote region (US-East). Performance of S-Paxos and U-Ring Paxos on the other hand is dictated by the slowest link.

- In most of the configurations and with small requests the leader in Libpaxos and OpenReplica is CPU-bound. Except for configuration (c), S-Paxos is always CPU-intensive. This happens because threads constantly spin while waiting for events (e.g., a message to arrive). U-Ring Paxos is never CPU-bound.

- Although all the implementations achieve more or less comparable peak throughput (in Mbps), the number of instances decided per second varies across them. We attribute this to differences in their batch sizes and also to the way batching takes place.

7.3.4 S-Paxos under failures

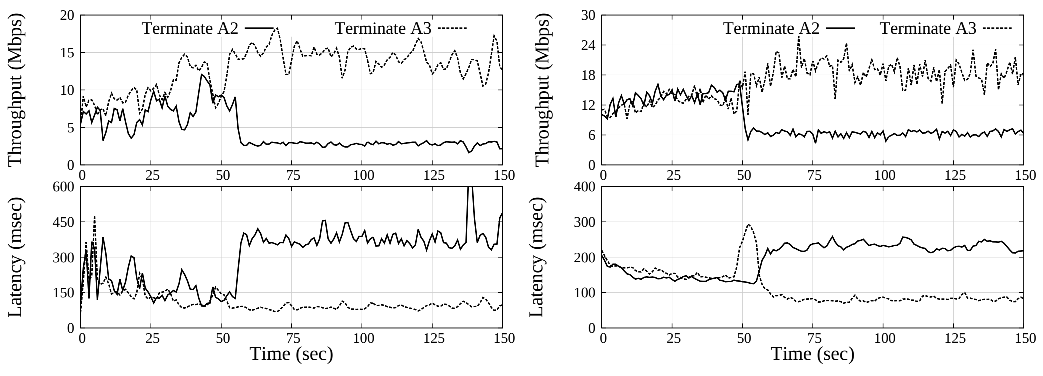


Figure 7.3. Performance of S-Paxos in configuration (b) with 4 KByte requests at 70% of peak throughput (left-most graphs); and in configuration (d) with 4 KByte requests at peak performance (right-most graphs). In each configuration, two experiments are performed; at each experiment one acceptor (A2 or A3) is terminated after 50 seconds.

Figure 7.3 shows the performance of S-Paxos in configuration (b) with 4 KByte requests at 70% of peak throughput and also in configuration (d) with 4 KByte requests over a period of 150 seconds and at peak throughput. The top graphs show the delivery throughput in megabits per second and the bottom graphs show the corresponding latency in milliseconds. In the experiments, after 50 seconds of the execution one acceptor is terminated.

In S-Paxos the load is distributed among acceptors and the execution proceeds at the pace of the slowest or the most distant acceptor. Thus, in both configurations after the termination of acceptor A3, throughput increases and latency decreases. This happens because performance is no longer limited by the slow acceptor. However, after the termination of acceptor A2, throughput decreases and latency increases. This is a consequence of the fact that acceptor A2 no longer contributes its share to the performance.

7.3.5 OpenReplica under failures

The left-most graphs of Figure 7.4 shows the performance of OpenReplica at configuration (c) with 100 KByte requests for a duration of 350 seconds at peak performance. In this execution, throughput varies between two thresholds. During intervals in which all the acceptors are responsive, throughput is higher.

Throughput is lower when the leader needs to devote a fraction of its processing power to retransmit messages to the slow acceptor (see also Section 7.2). During this period, only the faster acceptors contribute to performance. We terminated acceptor A2 after 150 seconds of the execution when the throughput was at its lower value. As it is seen in the figure, performance suffers a small reduction at this point. This is due to the fact that a majority-quorum is not immediately available, preventing the Paxos protocol from deciding new values until the quorum is restored.

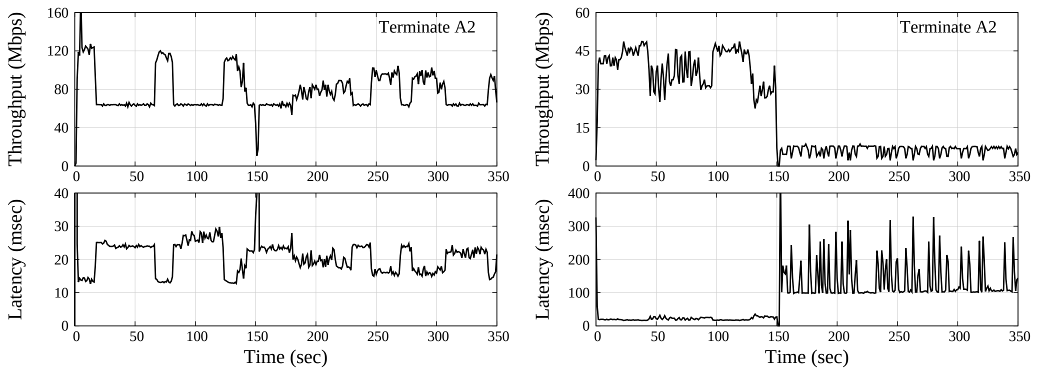


Figure 7.4. Performance of OpenReplica in configuration (c) with 100 KByte requests at peak performance (left-most graphs); and in configuration (d) with 4 KByte requests at 70 % of peak performance (right-most graphs); In both configurations, acceptor A2 is terminated after 150 seconds.

In the right-most graphs of Figure 7.4, we executed OpenReplica in configuration (d) with 4 KByte requests at 70% of peak performance. When acceptor A2 is terminated, after 150 seconds, throughput drops and latency increases since every majority quorum includes acceptors in different regions. With both acceptors A1 and A2 operational, though, we can see that throughput oscillates. We also observed that while OpenReplica has bursty behavior under high load (i.e., peak performance in a LAN with medium and large values, and 70% of peak performance in a WAN), it presents respectively more stable performance under moderate load.

7.3.6 U-Ring Paxos under failures

Figure 7.5 shows the performance of U-Ring Paxos in configurations (b) (left-most graphs) and configuration (d) (right-most graphs) over a period of 350 seconds. In all the experiments, after 150 seconds one of the acceptors (A2 or

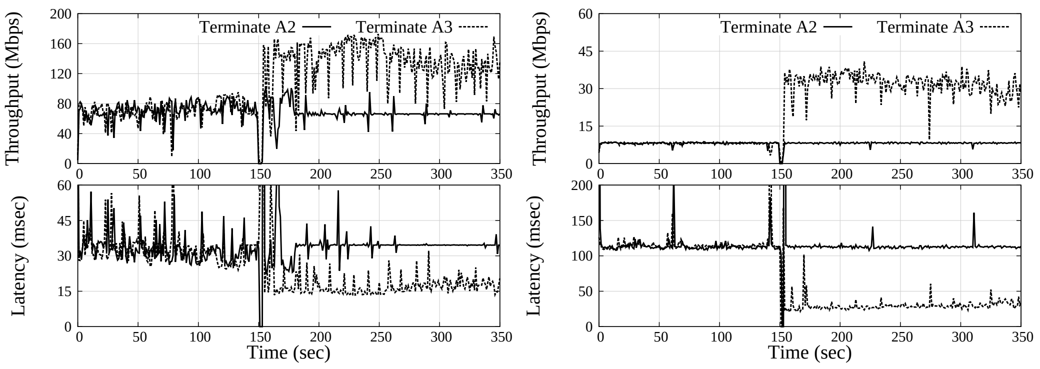


Figure 7.5. Performance of U-Ring Paxos in configuration (b) with 100 KByte requests at 100% of the peak performance (left-most) graphs; and in configuration (d) with 4 KByte requests at 70% of the peak performance (right-most graphs). In each configuration two experiments are performed; in each experiment one acceptor (A2 or A3) is terminated after 150 seconds.

A3) is terminated. After an acceptor is terminated, the performance drops to zero for a period of 2 to 3 seconds during which the ring is reconfigured. T he left-most graphs show the performance of U-Ring Paxos in configuration (b) with 100 KByte requests when the system is operating at its peak performance. When acceptor A3 is terminated, after 150 seconds, throughput increases. This is because U-Ring Paxos operates at the speed of the slowest acceptor (acceptor A3 in this experiment) and as soon as it leaves the ring, the protocol is no longer limited to its speed. This is also the reason why after terminating acceptor A2 the performance is not affected.

The right-most graphs of Figure 7.5 show the performance of U-Ring Paxos in configuration (d) with 4 KByte messages when the system is operating at 70% of the peak performance. When acceptor A3, in the east coast is terminated (after 150 seconds), throughput increases and latency decreases. This is because U-Ring Paxos is no longer bound by the slow communication links of acceptor A3. Similarly to configuration (b) in configuration (d) after terminating acceptor A2 performance does not improve.

7.3.7 Libpaxos and Libpaxos$^+$under failures

In this section, we consider the performance of Libpaxos and Libpaxos$^+$ in configurations (b) (Figure 7.6) and (d) (Figure 7.7) over a period of 150 seconds. In these experiments, the request size is 4 KBytes and the system operates at 70%

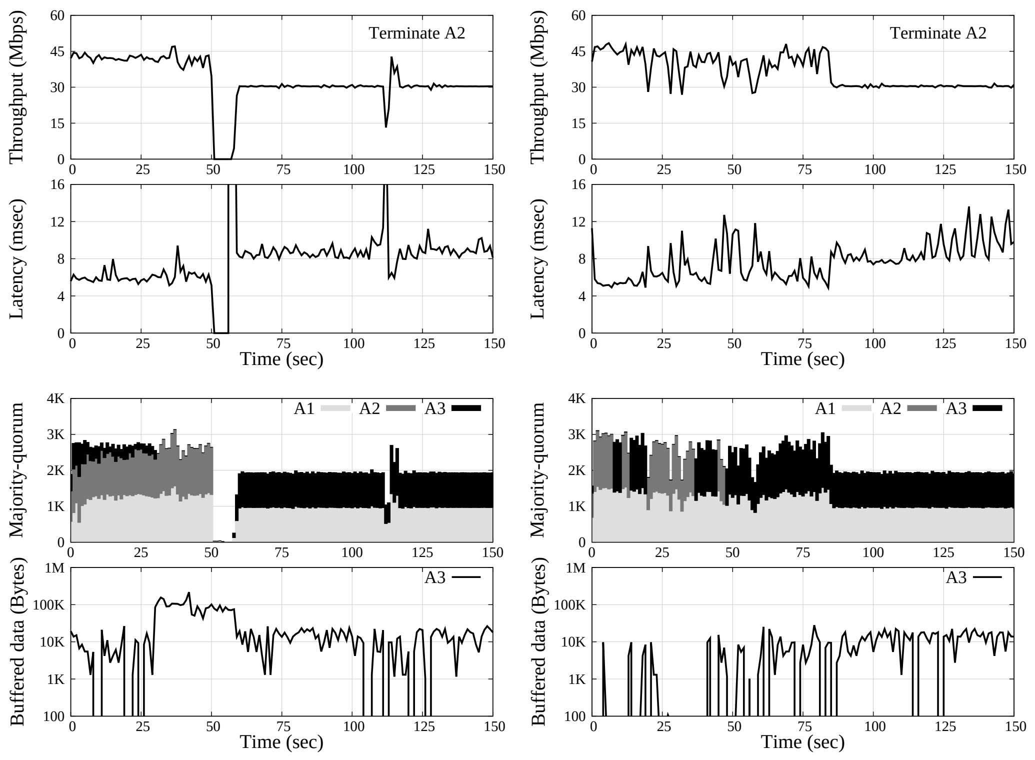


Figure 7.6. Performance of Libpaxos (left graphs) and Libpaxos$^+$ (right graphs) with 4 KByte requests at configuration (b) at 70% of peak throughput (see Table 7.1); acceptor A2 is terminated after 50 seconds. Majority-quorum for each acceptor measures the number of instances for which that acceptor is included in the first majority-quorum; The y-axis of the bottom-most graphs is in log scale.

of the peak throughput. Acceptor A2 is terminated after 50 seconds of the execution. In configuration (b), A2 is a fast acceptor and in configuration (d) A2 is an acceptor located in the same region as A1. In both cases, the termination of A2 forces slow acceptor A3 to be part of a majority-quorum. We report the following results in the graphs, from top to bottom: the delivery throughput in megabits per second, the latency as measured by the clients in milliseconds, the number of instances for which an acceptor's Phase 2B is included in that instance's first majority-quorum, and the amount of outgoing data buffered in the operating system at acceptor A3. We recall that acceptors in Libpaxos forward values and Phase 2B messages to the learners and proposers. Before the termination of acceptor A2 in the experiments of Libpaxos (left-side graphs in Figure 7.6), it is mostly acceptors A1 and A2 that participate in the majority-quorums. It can be

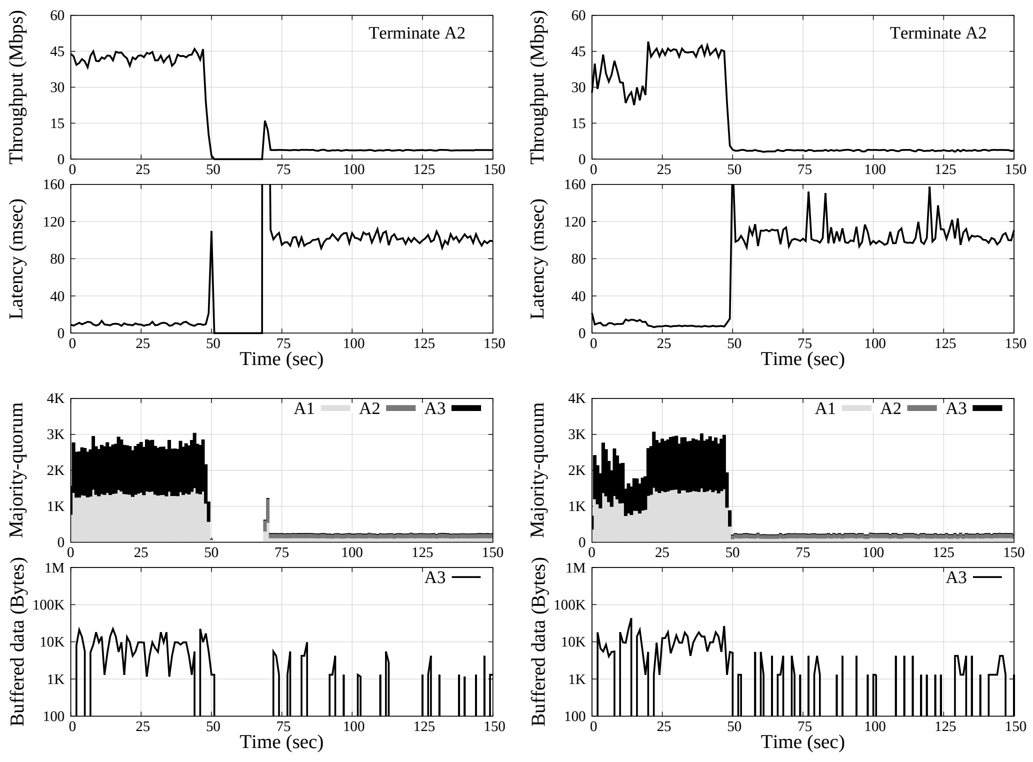


Figure 7.7. Performance of Libpaxos (left graphs) and Libpaxos$^+$ (right graphs) with 4 KByte requests at configuration (d) at 70% of peak throughput (see Table 7.1); acceptor A2 is terminated after 50 seconds. Majority-quorum for each acceptor measures the number of instances for which that acceptor is included in first majority-quorum. The y-axis of the bottom-most graphs is in log scale.

seen in the third graph from the top that in the first 30 seconds of the execution, A3 participates in a few majority-quorums but then it becomes overwhelmed and data in its send buffers accumulates (see bottom graph). When acceptor A2 is terminated (after 50 seconds), the delivery throughput drops to zero for a duration of 4 seconds, the time it takes for slow acceptor A3 to process its backlog of previous instances. After acceptor A3 empties its buffers and can participate in the majority-quorum, the system becomes responsive. With acceptors A1 and A3, throughput is lower and latency is higher than in the beginning of the execution. In Libpaxos$^+$ (graphs on the right), after termination of A2, the shift to a new majority-quorum happens smoothly since A3 was never overwhelmed with requests. The proposer detects the slower acceptor A3 and spares it. Thus, when acceptor A2 is terminated, a majority-quorum is available immediately and the execution continues smoothly. Notice the sustainable throughput in Libpaxos$^+$ before and after the termination of A2 is similar to Libpaxos.

In Figure 7.7 we investigate the behavior of Libpaxos and Libpaxos$^+$ in a wide-area deployment (configuration (d) in Table 7.1) with 4 KByte requests. The behavior of Libpaxos after the termination of faster acceptor A2 in configuration (d) is similar to configuration (b), except that in a wide-area deployment it takes longer (18 seconds) for acceptor A3 to catch up with A1. After the execution resumes (at time 68), there is an important reduction in throughput and increase in latency when compared to the execution before the termination of A2. This is due to the large round-trip time between US-West and US-East regions. Similarly to Figure 7.6, the throughput of Libpaxos$^+$ does not drop to zero after the termination of acceptor A2.

In both Figures 7.6 and 7.7, there is a peak in latency when normal operation resumes. This happens as the requests that the clients sent immediately before the crash are only decided after acceptor A3 catches up.

7.4 Main lessons from the experiments

In this section we share the main lessons we have learnt from our experiments with the four open-source Paxos libraries. Unlike Libpaxos and OpenReplica, S-Paxos and U-Ring Paxos allow clients to send their requests to any processes, which in turn disseminate the requests to other processes. The advantages of this scheme is that the protocol is not limited by the network and CPU resources of only one process (mostly the leader). Processes that receive client requests directly can batch the requests to make the distribution of requests to other participants more efficient. The downside of this strategy is that in global deployments (e.g., configuration (d)) the clients might select a process that is "far away". One way to enhance these libraries is to include policies for clients to select the process to which they transmit their requests (e.g., considering the delay between client and process). S-Paxos and U-Ring Paxos operate at the speed of the slowest participant. Therefore, in heterogeneous configurations (e.g., configurations (b) and (c)) or when participants are distributed across multiple data centers (e.g., configuration (d)), U-Ring Paxos and S-Paxos are likely outperformed by the other libraries, which do not require all acceptors to be equally powerful as the leader or in the vicinity of the leader. If heterogeneity affects a majority of the participants, however, all the libraries will operate at the speed of the slowest member. Notice that leaving out a slow acceptor during failure-free scenarios has both advantages and disadvantages. The advantage is that in the absence of failures, the protocol operates at the speed of the fastest available majority. Libpaxos and OpenReplica benefit from this. The disadvantage is that during failures, the protocol might be stuck as it happens with Libpaxos but not with U-Ring Paxos and S-Paxos. (U-Ring Paxos has to reconfigure the ring, but the reduction in performance is not due to the backlog of messages that gather at the slowest acceptor.) Ideally, a protocol would operate at the speed of the fastest members in failure-free scenarios and would not be penalized in case of failures, something we observed with Libpaxos$^+$.

While it may be tempting for systems running on a tight budget to deploy Paxos with a majority of fast acceptors, adding a few slower ones purely as backups, our study shows that this strategy may seriously impact failover. Despite this, the implications of a heterogeneous quorum is often neglected in practice. We hope this study will bring to the foreground the fact that performance differences in acceptors can have a significant impact on overall performance, especially when failures occur. We further suspect that the lessons we learnt apply to other quorum-based protocols, such as ABD [92] and the initial Isis protocol [93] since they all rely on a majority quorum for progress.

7.5 Conclusion

Paxos is one of the dominant protocols in building fault-tolerant systems and its performance has a significant impact on the overall efficiency of the systems built on top of it. Consequently, it is very important that Paxos implementations achieve steady performance and that they deal well with the load and scheduling variations common in modern cloud computing settings. Our experiments reveal that this property is achievable, but not without effort, and illustrate the surprisingly large variations in performance that out-of-the-box Paxos implementations may exhibit under even mild stress. In particular, we showed that without taking actions to stabilize the protocol, widely used Paxos libraries can exhibit sudden and rather long periods with no protocol decisions occurring at all. Our experiments also reveal surprising variability in the rate of decisions: some versions of Paxos are extremely bursty. Bursty throughput can cascade to create bursty and hence inefficient application-level performance. Finally, focusing on one Paxos implementation (Libpaxos), we showed how one can modify the protocol to preserve correctness and yet reduce the degree to which such problems arise.

Chapter 8

Conclusion

State-machine replication is a popular approach to high availability. In this thesis, we studied state-machine replication from a performance perspective and proposed new solutions to improve its efficiency. State-machine replication preserves strong consistency among independent replicas so that the users of a replicated service remain oblivious to the existence of multiple copies. However, if the replication library is not implemented efficiently, the users of a replicated service will perceive a performance lower than that of a single-copy non-replicated service. With this in mind, we designed, implemented, and extensively evaluated several solutions to reduce the negative effects of replication on performance. Our solutions proved promising not only at preserving the performance of a single-copy service but also at increasing it beyond the performance of a non-replicated service.

The flow of our study was shaped by two main objectives: Our first goal was to study the implications of request ordering on the performance of a service built on state-machine replication. To this end we studied and identified several fundamental issues in the design of atomic broadcast protocols and devised solutions to overcome them. Our observations and findings are mostly general and applicable to other protocols that seek high performance. As our second goal, we questioned sequential execution of requests on replicas in state-machine replication and employed parallelism in two different forms, state-partitioning and multithreading, to enhance performance. We iterated between these two goals as our findings in one direction led to new insights in the other. In the next section we briefly overview our findings and overview the important lessons we have learnt throughout this study.

8.1 Summary of our findings and lessons learnt

In the following we discuss and review some of the important lessons that we have learnt during this study.

1. To develop an efficient distributed system, effective communication patterns and networking requirements of the target environment must be identified and considered in the design, since communication is one of the most important and essential elements of any distributed system. We particularly emphasized the importance of communication patterns in Chapter 3 where we designed Ring Paxos, composed of two efficient atomic broadcast protocols: M-Ring Paxos and U-Ring Paxos. These protocols can be used interchangeably except that M-Ring Paxos needs ip-multicast support from the environment. Ring Paxos distributes its processes on a ring overlay and builds on several practical observations to achieve wire-speed efficiency. Our findings on communication patterns are general and applicable to the design of similar systems.

2. In classic state-machine replication, ordering and execution of requests follow a pipeline structure: a request is executed only after its order is determined. The existence of an ordering stage between clients and servers adds extra overhead to the response time. An approach to mitigating the effect of ordering on response time is to replace the pipelined structure with a parallel model. Optimistic delivery is a solution for realizing this parallelism. The idea is for the replicas to execute requests while their order is being determined. This strategy is deemed optimistic as replicas assume that the final order will abide by the execution order. In the case of a mismatch, execution must be rolled back and the system's state prior to the execution must be restored. For optimistic delivery to prove efficient, the optimistic assumption must be based on some realistic evidence. If not, high frequency of roll backs will impose extra overheads on the performance, such that contrary to its initial motivation, optimistic delivery will increase the response time. In Chapter 4 we implemented speculative delivery with M-Ring Paxos to reduce latency. Depending on the efficiency of the ordering protocol, the amount of advantage gained by optimistic delivery will vary. M-Ring Paxos is an efficient protocol and obtaining any gains in latency is challenging. Improvement on latency also depends on the service and duration of a request's execution. If execution time is much higher than ordering time, then the advantage of optimistic delivery will be negligible.

3. Throughput of a replicated service is limited either by the number of requests that replicas can execute or by the number of requests an atomic broadcast protocol can order. For the former case, partitioning the state is a well-known technique to speedup request execution. Partitioning is effective in that it introduces parallelism in the execution: different sets of replicas deliver different streams of ordered requests and in parallel to each other execute their own stream of requests sequentially. In Chapter 4 we used state partitioning in combination with M-Ring Paxos to improve throughput. We saw that state partitioning was extremely useful in increasing throughput of our examined application where the execution was limiting the performance.

4. If throughput of a replicated service is restricted by the number of requests that can be ordered, state partitioning will be ineffective in improving performance. Adding new participants to atomic broadcast protocol can not help either, as more processes will only enhance the availability and fault-tolerance of the protocol but not its performance. Moreover, if the participants of an atomic broadcast protocol assume uneven roles, throughput will be restrained from increasing once a process with a heavier role reaches its maximum performance. We observed this phenomenon with M-Ring Paxos protocol in Chapter 5. These problems are not specific to Ring Paxos and can be found in similar atomic broadcast protocols. In Chapter 5 we studied the scalability of Ring Paxos's throughput and proposed Multi-Ring Paxos: a group of Ring Paxos instances that are coordinated via several pre-configured parameters to collectively order the requests with a high throughput. The ideas used in designing Multi-Ring Paxos can be generalized and applied to other atomic broadcast protocols. Multi-Ring Paxos does not demand partitioning but can be used with partitioned services as well. Since partitioning is not a must, services whose states can not be partitioned can also benefit from the high performance and scalability of Multi-Ring Paxos.

5. Partitioning the state of a service, as an approach to enhancing performance of request execution, is only useful for applications whose state can be partitioned. An alternative technique is to make multiple threads concurrently execute the requests. In Chapter 6 we studied this approach and observed that sequentiality of state-machine replication can be altered to support the replication of multithreaded services without compromising consistency. We studied parallel replication models in Chapter 6 and realized that despite parallelism in executing requests, they suffer from a se-

quential delivery mechanism. With multi-core processors and multi-queue network interfaces available as commodity, a parallelized system must not possess intrinsic design properties that restrict its capability at fully benefiting from these technologies. Based on this observation we built on top of Multi-Ring Paxos and proposed P-SMR, a new model for parallelizing request execution and request delivery in state-machine replication. Our experimental evaluations showed significant performance improvement over the state-of-the-art approaches. We also saw that parallelized models lose performance to the sequential execution model if the workload demands heavy synchronization among concurrent threads.

6. When evaluating Paxos and its variants often a homogenous and highly controlled environment is assumed, in which processes operate at comparable speeds and exclusively own the machine on which they are deployed. In the emerging cloud-based deployments, however, this is not the case. We evaluated several open-source libraries of Paxos on Amazon' EC2 infrastructure and demonstrated that the performance of even highly efficient implementations of Paxos is subject to the heterogeneity of the environment and can suffer from down times. We concluded that a Paxos implementation should take proper actions to protect its performance's stability in such environments, and proposed a solution to be integrated with one of the evaluated libraries.

8.2 Future directions

In this section we conclude the dissertation by suggesting some directions for future research:

Local versus geographic replication. A service can be replicated locally within a data center or geographically across multiple data centers. In this work we have studied the performance of state-machine replication in local deployments. Geographic replication has several advantages over local replication. First, the replicated service is no longer vulnerable to the failures that affect an entire site such as data center-wide power outages or natural disasters that affect a specific area (e.g., earthquake). Second, the latency observed by clients is reduced as a consequence of data locality (i.e., to issue their requests clients communicate with a replica in their vicinity rather than with some replica in a remote location, assuming that the clients of a service are spread world wide). In addition to differences in fault tolerance and co-locality, local and geographic deployments differ also in the quality and characteristics of their networking infrastructure. While network links within a data center have low latency and high bandwidth, network links among data centers are subject to high, unpredictable, and variable latency and many suffer from low bandwidth and high congestion. As a consequence, systems designed to operate in geographic scale should avoid inter-data center communications to a large extent and mask the negative effects of long-distance communication from the user. Because of these differences, replication techniques developed for local deployments are not efficient when used in geographic distributions. As one of the directions for expanding the subject of this study, it is worthwhile to investigate the applicability and adaptability of our techniques to geographic replication and also to propose new and specific solutions that meet the requirements of these environments.

Sensitivity of performance to workload specifics. In Chapter 6 we proposed a new model for parallelizing request execution on the replicas of state-machine replication. We observed that the scalability and efficiency of our model and its competitors is affected by the characteristics of the workload. In our case, this effect stems from the need to synchronize concurrent threads. In some scenarios, the effect of synchronization is such that the parallelized models have lower performance than the sequential model. Clearly, for parallelized models to be favored and widely used, their performance should be higher from and at worst equal to the performance of sequential execution model. To achieve this, synchronization strategies in parallel replication models should be optimized. The literature on synchronization techniques is vast. While some techniques offer optimizations over well-known synchronization strategies (e.g., locking) others aim at avoiding the inefficiency of locking algorithms all together. As the second direction for improving this work, it is worthwhile to study synchronization techniques and possibly propose new and efficient synchronization mechanisms that can be integrated with parallel replication models.

Appendices

Proofs of Correctness

Ring Paxos

In this section we provide a proof sketch of the correctness of M-Ring Paxos and U-Ring Paxos protocols. We focus on properties II and III of consensus (see Chapter 2). Property (I) holds trivially from the algorithms.

Proposition 1 *(II) Uniform agreement: No two processes decide different values.*

Let v and v' be two decided values, and $v\text{-}id$ and $v'\text{-}id$ their unique identifiers. We prove that $v\text{-}id = v'\text{-}id$.

M-Ring Paxos: Let r (r') be the round in which some coordinator c (c') ip-multicast a decision message with $v\text{-}id$ ($v'\text{-}id$). In M-Ring Paxos, c ip-multicasts a decision message with $v\text{-}id$ after:

(a) c receives $f+1$ messages of the form (Phase 1B, r, *, *);

(b) c selects the value $v_{val} = v$ with the highest round number v_{rnd} among the set M_{1B} of phase 1B messages received, or picks a value v if $v_{rnd} = 0$;

(c) c ip-multicasts (Phase 2A, r, v, $v\text{-}id$); and

(d) c receives (Phase2B, r, $v\text{-}id$) from the second last process in the ring, q. When c receives this message from q, it is equivalent to c receiving $f+1$ (Phase 2B, r, $v\text{-}id$) messages directly because the ring is composed of $f+1$ acceptors. Let M_{2B} be the set of $f+1$ phase 2B messages.

Now consider that coordinator c received the set of messages M_{1B} and M_{2B} in a system where all processes ran Paxos on value identifiers. In this case, c would send a decide message with $v\text{-}id$ as well. Since the same reasoning can be applied to coordinator c', and Paxos implements consensus, $v\text{-}id = v'\text{-}id$. □

U-Ring Paxos: Let r (r') be the round in which the last acceptor a_l (a'_l) sends a decision message m_D with $v\text{-}id$ ($v'\text{-}id$) along the ring. The proof for U-Ring Paxos is similar to the proof for M-Ring Paxos since U-Ring Paxos only differs in the way m_D is propagated to the learners and in the identity of the process who first sends m_D. In contrast to M-Ring Paxos where it is the coordinator that sends m_D, in U-Ring Paxos, it is the last acceptor in the ring a_l that initiates the propagation of m_D along the ring. Despite this difference, a_l sends m_D when a_l receives, $f+1$ (Phase 2B, r, v, $v\text{-}id$) messages, just as in M-Ring Paxos. Since M-Ring Paxos guarantees that $v\text{-}id = v'\text{-}id$, the same holds in U-Ring Paxos. □

Proposition 2 *(III) Uniform termination: If one (or more) correct process proposes a value then eventually some value is decided by all correct processes.*

The proof is almost identical for M-Ring Paxos and U-Ring Paxos. We note the differences when necessary. After GST, processes eventually select a correct coordinator c. c considers a ring (c-ring) composed entirely of correct acceptors, for M-Ring Paxos, and a ring (c-ring) composed entirely of correct proposers, acceptors, and learners, for U-Ring Paxos. Coordinator c sends a message of the form (Phase 1A, *, c-ring) to the acceptors in c-ring. Because after GST, all processes are correct and all messages exchanged between correct processes are received, all correct processes eventually decide some value. □

Linearizability of state partitioning

In this section we argue that our replicated and partitioned B-tree from Chapter 4 is linearizable (see Chapter 2 for the definition of linearizability).

Assumptions. We make the following assumptions:

- Assume a B-tree whose state is divided into partitions $\Pi = \{P_1, P_2, ...\}$.
- A command C is composed of one or more sub-commands $C(k)$, one for each partition P_k it addresses. In particular, C can insert, delete or query items in the B-tree.
- Each B-Tree partition is replicated and implemented as a series of consensus executions such that the i-th consensus instance decides on the i-th sub-command of partition P_k. Sub-commands in a partition are executed in the order in which they are decided, that is, the i-th sub-command only starts after the $(i-1)$-th sub-command has finished.
- $G = (V, E)$ is a directed graph where V contains all commands C_x in the execution and E contains a directed edge $C_x \rightarrow C_y$ iff a sub-command of C_x is executed before a sub-command of C_y in some partition P_k. State partitioning ordering states that G is acyclic.

Proof Sketch. In order to show that any execution of the B-tree implemented using state-machine replication and state partitioning is linearizable, we must show that there is a way to reorder the commands in a sequence S such that (i) S respects the order of non-overlapping commands across all clients, and (ii) S respects the semantics of the commands, as defined in their sequential specifications.

(i) *We first show that there is a sequence S that respects the order of non-overlapping commands across all clients.* To do so, we consider two conditions:

(a) If C_x precedes C_y in G, then C_x precedes C_y in S.

(b) If C_x finishes before C_y starts (i.e., they are non-overlappping), then we order C_x before C_y in S.

We claim that conditions (a) and (b) can always be accommodated. To see why, assume for a contradiction that C_x precedes C_y in G and C_y finishes before C_x starts. From the fact that C_x precedes C_y in G, both C_x and C_y access some partition P_k and $C_x(k)$ is executed before $C_y(k)$ at P_k. Thus, $C_x(k)$ is delivered before $C_y(k)$, and it follows that C_y cannot finish before C_x starts.

(ii) *We next show that sequence S respects the semantics of B-tree commands.* We must show that any command in S takes into account all commands that precede it, and in the order in which they appear in S. Let C_x be a command in S. For every sub-command $C_x(k)$ of C_x, only commands on P_k can affect $C_x(k)$, thus, we can focus on sub-commands $C_y(k)$ only, that is, sub-commands of some command C_y on the same partition P_k. Since C_x and C_y are composed of sub-commands on a common partition, from the definition of G and the fact that it is acyclic, we can totally order them. Thus, sub-commands $C_x(k)$ and $C_y(k)$ will be executed on partition P_k according to the order of C_x and C_y in S. Moreover, from the implementation of each partition, a sub-command in a partition is only executed after the sub-command that precedes it is completed. Thus, sub-commands take into account their preceding sub-commands, in the order they are executed.

Multi-Ring Paxos

In this section, we argue that Multi-Ring Paxos ensures uniform agreement, uniform partial order, and validity (for the definition of atomic multicast see Chapter 2).

Proposition 3 *(III) Uniform agreement: If a process delivers message m multicast to g_i, then all correct processes in g_i deliver m.*

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

Proposition 4 *(IV) Uniform partial order. If processes p and q deliver messages m and m', then they deliver them in the same order.*

Two cases must be considered:

(a) m and m' were multicast to the same group g;

(b) m and m' were multicast to groups g_i and g_j, respectively, where $i < j$.

In case (a), it is simple to see from Algorithm 1 in Chapter 5 that both messages are delivered in the same order by all processes. Partial order also holds for case (b) from the fact that processes order groups in the same way and first deliver M messages from one group and then deliver M from the other. Assume m is delivered in consensus instance k_i and m' in consensus instance k_j. If $k_i \leq k_j$, then both p and q will deliver m first and then m'. If $k_i > k_j$, then both processes will deliver m' first and then m. □

Proposition 5 *(I) Validity. If a correct process multicasts a message m to g, then all correct processes in g will eventually deliver m.*

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

P-SMR

In this section we show that P-SMR is linearizable and deadlock-free.

P-SMR is linearizable. From the definition of linearizability (see Chapter 2), we show that there is a permutation π of commands in $\mathcal{E}$ that respects (i) the real-time ordering of commands across all clients, and (ii) the semantics of the commands. Let C_x and C_y be two commands in $\mathcal{E}$ submitted by clients c_x and c_y, respectively.

Two cases must be considered:

(a) C_x and C_y are independent. Thus, either C_x and C_y access disjoint sets of variables or only read variables commonly accessed. Consequently, the execution of one command does not affect the execution of the other and they can be placed in any relative order in π. We arrange C_x and C_y in π so that their relative order respects their real-time dependencies, if any.

(b) C_x and C_y are dependent. Assume C_x and C_y are multicast to groups in γ_x and γ_y, respectively. From the fact that C_x and C_y depend on each other, $\gamma_{xy} = \gamma_x \cap \gamma_y \neq \emptyset$. In every correct server s, C_x (resp. C_y) is delivered by all threads in groups in γ_x (resp. γ_y) and executed by one thread, say t_x (resp. t_y). From the order property of atomic multicast, every thread in groups in γ_{xy} delivers C_x and C_y in the same relative order. Without lack of generality, assume C_x is delivered before C_y.

We first claim that t_x executes C_x before t_y executes C_y and the execution satisfies the sequential semantics of the commands. To see why, notice that t_x only executes C_x after t_i delivers C_x and every other thread in groups in γ_x delivers C_x and signal t_i. Every thread $t \neq t_i$ in a group in γ_x then waits until t_i executes C_i to proceed with the next command. Thus, t_y will only receive a signal from threads in groups in γ_{xy} and execute C_y after t_x has executed C_x. Consequently, the two commands execute in sequence, which satisfies their semantics.

We now claim that the delivery order satisfies any real-time constraints among C_x and C_y. Without lack of generality, assume C_x finishes before C_y starts, that is, C_x precedes C_y in real time. Thus, before C_y is multicast by a client, C_x has completed (i.e,. its client has received C_x's response). The claim follows from the fact that before C_x is executed, it must be multicast, and thus C_x is delivered before C_y. From the claims above, we can arrange C_x and C_y in π according to their delivery order so that the execution of each command satisfies its semantics.

P-SMR is deadlock-free. For a contradiction, assume a deadlock where thread x_1 waits for $x_2, ..., x_l$ waits for x_1. Let $p(x)$ (resp. $n(x)$) be the thread that precedes (resp. succeeds) x in the deadlock chain. Thread x waits for $n(x)$ if (1) there is a command $C_{x,n(x)}$ multicast to groups that contain x and $n(x)$; (2) x delivered $C_{x,n(x)}$; and (3) x needs a signal from $n(x)$ (a) before x executes $C_{x,n(x)}$ or (b) after $n(x)$ executes $C_{x,n(x)}$.

We now claim that x delivers $C_{x,n(x)}$ before $C_{p(x),x}$, that is, $C_{x,n(x)} < C_{p(x),x}$. To see why, assume x delivers $C_{p(x),x}$ before $C_{x,n(x)}$. From the algorithm, when x delivers $C_{x,n(x)}$ it has

(a) sent a signal to $p(x)$, if $p(x)$ was to execute $C_{p(x),x}$ or

(b) received a signal from $p(x)$, if x was to execute $C_{p(x),x}$.

In both cases, $p(x)$ cannot wait for x, as assumed in our deadlock chain. From the claim above, $C_{x_l,x_1} < C_{x_{l-1},x_l} < ... < C_{x_l,x_1}$, which contradicts the atomic multicast order property.

Bibliography

[1] S. Shankland, "Google spotlights data center inner workings."

[2] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *ACM SIGMOD Record*, vol. 25, pp. 173–182, ACM, 1996.

[3] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[4] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, May 1998.

[5] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty processor," *Journal of the ACM*, vol. 32, no. 2, pp. 374–382, 1985.

[6] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," *Journal of the ACM*, vol. 35, no. 2, pp. 288–323, 1988.

[7] F. B. Schneider, "What good are models and what models are good?," in *Distributed Systems* (S. Mullender, ed.), ch. 2, Addison-Wesley, 2nd ed., 1993.

[8] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *J. ACM*, vol. 43, no. 2, pp. 225–267, 1996.

[9] M. P. Herlihy and J. M. Wing, "Linearizability: a correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, pp. 463–492, July 1990.

[10] J. Kirsch and Y. Amir, "Paxos for system builders: An overview," in *LADIS*, 2008.

[11] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock, "Dr. multicast: Rx for data center communication scalability," EuroSys, 2010.

[12] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, pp. 5:1–5:32, July 2010.

[13] R. Ekwall, A. Schiper, and P. Urbán, "Token-based atomic broadcast using unreliable failure detectors," in *SRDS*, 2004.

[14] L. Lamport and M. Massa, "Cheap Paxos," in *DSN*, 2004.

[15] N. Santos and A. Schiper, "Tuning paxos for high-throughput with batching and pipelining," in *ICDCN*, 2012.

[16] T. Chandra, R. Griesemer, and J. Redstone, "Paxos made live: An engineering perspective," in *PODC*, 2007.

[17] R. van Renesse, "Paxos made moderately complex," tech. rep., Cornell University, March 2011.

[18] W. J. Bolosky, D. Bradshaw, R. B. Haagens, N. P. Kusters, and P. Li, "Paxos replicated state machines as the basis of a high-performance data store," in *NSDI*, 2011.

[19] M. Burrows, "The Chubby Lock Service for loosely-coupled distributed systems," in *OSDI*, 2006.

[20] J. Baker, C. Bond, J. Corbett, J. J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, "Megastore: Providing scalable, highly available storage for interactive services," in *CIDR*, 2011.

[21] K. P. Birman and T. A. Joseph, "Reliable communication in the presence of failures," *ACM Trans. Comput. Syst.*, vol. 5, pp. 47–76, Jan. 1987.

[22] K. P. Birman, A. Schiper, and P. Stephenson, "Lightweight causal and atomic group multicast," *ACM Trans. Computer Systems*, vol. 9, pp. 272–314, Aug. 1991.

[23] M. F. Kaashoek and A. S. Tanenbaum, "Group communication in the Amoeba distributed operating system," in *ICDCS*, 1991.

[24] J.-M. Chang and N. Maxemchuk, “Reliable broadcast protocols,” *ACM Trans. Comput. Syst.*, vol. 2, no. 3, pp. 251–273, 1984.

[25] F. Cristian and S. Mishra, “The Pinwheel asynchronous atomic broadcast protocols,” in *International Symposium on Autonomous Decentralized Systems (ISADS)*, (Phoenix, Arizona, USA), 1995.

[26] J. Kim and C. Kim, “A total ordering protocol using a dynamic token-passing scheme,” *Distributed Systems Engineering*, vol. 4, no. 2, pp. 87–95, 1997.

[27] Y. Mao, F. P. Junqueira, and K. Marzullo, “Mencius: building efficient replicated state machines for wans,” in *OSDI*, 2008.

[28] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal, “Fault-tolerant total order multicast to asynchronous groups,” in *SRDS*, 1998.

[29] L. Lamport, “The implementation of reliable distributed multiprocess systems,” *Computer Networks*, vol. 2, pp. 95–114, 1978.

[30] T. Ng, “Ordered broadcasts for large applications,” in *Symposium on Reliable Distributed Systems (SRDS)*, pp. 188–197, 1991.

[31] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella, “The Totem single-ring membership protocol,” *ACM Trans. Computer Systems*, vol. 13, no. 4, pp. 311–342, 1995.

[32] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, “S-paxos: Offloading the leader for high throughput state machine replication,” *Reliable Distributed Systems, IEEE Symposium on*, vol. 0, pp. 111–120, 2012.

[33] Y. Amir, C. Danilov, M. Miskin-Amir, J. Schultz, and J. Stanton, “The Spread toolkit: Architecture and performance,” tech. rep., Johns Hopkins University, 2004. CNDS-2004-1.

[34] http://libpaxos.sourceforge.net.

[35] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso, and B. Kemme, “Are quorums an alternative for data replication?,” *ACM Transactions on Database Systems*, vol. 28, no. 3, pp. 257–294, 2003.

[36] R. Jiménez-Peris, M. Patiño Martínez, B. Kemme, and G. Alonso, “Improving the scalability of fault-tolerant database clusters,” in *ICDCS*, 2002.

[37] B. Kemme, F. Pedone, G. Alonso, and A. Schiper, "Processing transactions over optimistic atomic broadcast protocols," in *ICDCS*, 1999.

[38] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[39] N. Schiper, *On Multicast Primitives in Large Networks and Partial Replication Protocols*. PhD thesis, University of Lugano, 2009.

[40] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: speculative byzantine fault tolerance," in *PSOSP*, 2007.

[41] B. Wester, J. Cowling, E. B. Nightingale, P. M. Chen, J. Flinn, and B. Liskov, "Tolerating latency in replicated state machines through client speculation," in *NSDI*, 2009.

[42] C. Coulon, E. Pacitti, and P. Valduriez, "Consistency management for partial replication in a high performance database cluster," ICPADS, 2005.

[43] A. de Sousa, R. C. Oliveira, F. Moura, and F. Pedone, "Partial replication in the database state machine," NCA, pp. 298–309, IEEE Computer Society, 2001.

[44] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware," in *Proc. of USENIX Annual Technical Conference, Freenix track*, 2004.

[45] N. Schiper, R. Schmidt, and F. Pedone, "Optimistic algorithms for partial database replication," OPODIS, 2006.

[46] U. Fritzke and P. Ingels, "Transactions on partially replicated data based on reliable and atomic multicasts," in *ICDCS*, 2001.

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

[48] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme, "Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation," in *PRDC*, IEEE Computer Society, 2007.

[49] D. Serrano, M. Patiño-Martínez, R. Jiménez-Peris, and B. Kemme, "An autonomic approach for replication of internet-based services," in *SRDS*, 2008.

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

[51] S. Elnikety, F. Pedone, and W. Zwaenepoel, “Database replication using generalized snapshot isolation,” in *SRDS*, (Orlando, USA), 2005.

[52] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. E. Armendáriz-Iñigo, “Snapshot isolation and integrity constraints in replicated databases,” *ACM Trans. Database Syst.*, vol. 34, no. 2, 2009.

[53] X. Défago, A. Schiper, and P. Urbán, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Comput. Surv.*, vol. 36, no. 4, pp. 372–421, 2004.

[54] M. K. Aguilera, W. M. Golab, and M. A. Shah, “A practical scalable distributed B-tree,” *PVLDB*, vol. 1, no. 1, pp. 598–609, 2008.

[55] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis, “Sinfonia: a new paradigm for building scalable distributed systems,” in *SOSP*, 2007.

[56] M. Primi. http://libmarco.googlecode.com/svn/trunk/.

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

[58] http://libpaxos.sourceforge.net/ringpaxos.

[59] C. Curino, E. Jones, Y. Zhang, and S. Madden, “Schism: a workload-driven approach to database replication and partitioning,” *Proc. VLDB Endow.*, vol. 3, pp. 48–57, 2010.

[60] M. Adler, Z. Ge, J. F. Kurose, D. F. Towsley, and S. Zabele, “Channelization problem in large scale data dissemination,” ICNP, pp. 100–109, 2001.

[61] K. P. Birman and T. A. Joseph, “Reliable communication in the presence of failures,” *ACM Transactions on Computer Systems (TOCS)*, vol. 5, pp. 47–76, Feb. 1987.

[62] R. Guerraoui and A. Schiper, “Genuine atomic multicast in asynchronous distributed systems,” *Theor. Comput. Sci.*, vol. 254, no. 1-2, pp. 297–316, 2001.

[63] N. Schiper and F. Pedone, “On the inherent cost of atomic broadcast and multicast in wide area networks,” ICDCN, 2008.

[64] J. Fritzke, U., P. Ingels, A. Mostefaoui, and M. Raynal, “Fault-tolerant total order multicast to asynchronous groups,” SRDS, 1998.

[65] L. Rodrigues, R. Guerraoui, and A. Schiper, “Scalable atomic multicast,” ICCCN, 1998.

[66] C. Delporte-Gallet and H. Fauconnier, “Fault-tolerant genuine atomic multicast to multiple groups,” OPODIS, 2000.

[67] M. K. Aguilera and R. E. Strom, “Efficient atomic broadcast using deterministic merge,” PODC, 2000.

[68] F. B. Schneider, “Implementing fault-tolerant services using the state machine approach: A tutorial,” *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.

[69] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, “Eve: Execute-verify replication for multi-core servers,” in *OSDI*, 2012.

[70] R. Kotla and M. Dahlin, “High throughput byzantine fault tolerance,” in *DSN*, 2004.

[71] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, 1984.

[72] A. S. Tanenbaum, *Distributed operating systems*. Pearson Education India, 1995.

[73] N. Santos and A. Schiper, “Achieving high-throughput state machine replication in multi-core systems,” tech. rep., EPFL, 2011.

[74] A. Aviram, S.-C. Weng, S. Hu, and B. Ford, “Efficient system-enforced deterministic parallelism,” in *OSDI*, 2010.

[75] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble, “Deterministic process groups in dos,” in *OSDI*, 2010.

[76] J. Devietti, B. Lucia, L. Ceze, and M. Oskin, “DMP: deterministic shared memory multiprocessing,” in *ASPLOS*, 2009.

[77] A. Thomson and D. J. Abadi, “The case for determinism in database systems,” *Proc. VLDB Endow.*, vol. 3, pp. 70–80, Sept. 2010.

[78] G. Altekar and I. Stoica, “ODR: output-deterministic replay for multicore debugging,” in *SOSP*, 2009.

[79] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, “Execution replay of multiprocessor virtual machines,” in *VEE*, 2008.

[80] P. Montesinos, L. Ceze, and J. Torrellas, “Delorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently,” in *ISCA*, 2008.

[81] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu, “PRES: probabilistic replay with execution sketching on multiprocessors,” in *SOSP*, 2009.

[82] M. Ronsse and K. De Bosschere, “Recplay: a fully integrated practical record/replay system,” *ACM Trans. Comput. Syst.*, vol. 17, pp. 133–152, May 1999.

[83] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy, “DoublePlay: parallelizing sequential logging and replay,” *SIGPLAN Not.*, vol. 47, pp. 15–26, Mar. 2011.

[84] M. Xu, R. Bodik, and M. D. Hill, “A “flight data recorder” for enabling full-system multiprocessor deterministic replay,” in *ISCA*, 2003.

[85] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg, “Thrifty generic broadcast,” in *DISC*, 2000.

[86] F. Pedone and A. Schiper, “Generic broadcast,” in *DISC*, 1999.

[87] L. Lamport, “Generalized consensus and paxos,” Tech. Rep. MSR-TR-2005-33, Microsoft Research (MSR), Mar. 2005.

[88] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg, “The primary-backup approach,” in *Distributed systems (2nd Ed.)* (S. Mullender, ed.), New York, NY: ACM Press/Addison-Wesley Publishing Co., 1993.

[89] F. P. Junqueira, B. C. Reed, and M. Serafini, “Zab: High-performance broadcast for primary-backup systems,” in *DSN*, 2011.

[90] R. van Renesse, “Paxos made moderately complex,” tech. rep., Cornell University, 2011.

[91] D. Altinbuken and E. G. Sirer, “Commodifying replicated state machines with openreplica.” Avaible at http://openreplica.org/static/papers/OpenReplica.pdf, 2012.

[92] H. Attiya, A. Bar-Noy, and D. Dolev, “Sharing memory robustly in message-passing systems,” *Journal of the ACM (JACM)*, vol. 42, no. 1, pp. 124–142, 1995.

[93] K. Birman and R. Cooper, “The ISIS project: Real experience with a fault tolerant programming system,” in *SIGOPS European workshop*, 1990.