Università
della
Svizzera
italiana

Faculty
of Informatics

USI Technical Report Series in Informatics

Providing Scalability and Low Latency in State Machine Replication

Carlos Eduardo Bezerra[1,2], Fernando Pedone[1], Robbert van Renesse[3], Cláudio Geyer[2]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul, Brazil
[3] Computer Science Department, Cornell University, USA

Abstract

State machine replication (SMR) is a well-known replication technique that ensures strong consistency (linearizability) for distributed services. Even though SMR ensures strong consistency, it provides limited throughput scalability, since every replica executes every command. We propose Scalable SMR (S-SMR), a technique that provides throughput scalability by means of partitioning the service state and partially ordering commands across partitions, along with two optimizations: caching and state prefetching.

We also propose Fast-SSMR, which changes S-SMR by providing an extended interface between clients and servers. Fast-SSMR implements two state machines at each server replica: an *optimistic state machine* and a *conservative state machine*. Both state machines respond to each command, resulting in a fast *optimistic reply*, which is correct with a high probability, and a *conservative reply*, which is always correct and confirms the optimistic reply. This way, Fast-SSMR does not require the service to be able to undo operations.

We have implemented both S-SMR and Fast-SSMR as a Java library, along with a collection of representative applications. Our experiments show that both S-SMR and Fast-SSMR achieve throughput scalability with the number of partitions, and that the optimistic replies are significantly faster than the conservative replies.

Report Info

Published
December 2015
Number
USI-INF-TR-2015/06
Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
Online Access
www.inf.usi.ch/techreports

1 Introduction

Many current online services have stringent availability and performance requirements. Providing availability involves tolerating failures: if one server goes down, the service stays available through server replicas, which should be consistent with one another. Performance translates into *scalable throughput* and *low latency*, that is, the ability to accommodate an ever-increasing number of clients and to provide a good user experience by means of low response times. State machine replication (SMR) [1, 2] is a well-established replication approach to fault-tolerance, while also ensuring strong consistency (i.e., linearizability). With strong consistency, clients cannot distinguish the behavior of the replicated system from that of a single-server system [3]. SMR ensures that all server replicas execute the same sequence of commands, one by one. Assuming that each command execution is deterministic, every server reaches the same state after executing the same command in the sequence. Although this ensures strong consistency and fault tolerance, SMR was not designed to scale throughput with an increasing number of replicas. Adding more of them increases the number of failures tolerated by the service, but likely not the maximum throughput of the system.

To provide scalability, many distributed systems rely on partitioning (e.g., [4–7]). If different requests can be handled simultaneously by servers of different partitions, then augmenting the number of partitions results in an overall increase in system throughput. In the context of SMR, however, exploiting partitioned state is challenging for several reasons. First, as we want the system to scale, we should avoid totally ordering commands. However, imposing a partial order on commands allows some executions that violate linearizability. Second, with SMR, all the service designer has to do is provide a sequential implementation of each command. This is relatively simple in SMR because every replica contains all state that may be accessed by the commands. With partitioned state, it may be necessary for replicas to exchange portions of the state when executing commands that access multiple partitions. Finally, when executing commands that span across multiple partitions, the coordination between partitions is likely to significantly increase the response time for that command.

This paper presents Scalable State Machine Replication (S-SMR), an approach that achieves scalable throughput and strong consistency (i.e., linearizability) without constraining service commands or adding additional complexity to their implementation. S-SMR partitions the service state and relies on atomic multicast [8], a class of protocols that ensure partial, yet consistent ordering of messages sent to groups of processes. Since simply ordering commands consistently across partitions is not enough to ensure linearizability, S-SMR implements *execution atomicity*, a property that prevents command interleavings that violate our consistency criterion.

To reduce latency when executing multi-partition commands, we also propose a few optimizations: conservative caching, speculative caching and state prefetching. Conservative caching avoids unnecessary exchange of state between servers when executing multi-partition commands: if a cached item is still up-to-date, there is no need to send the value again, as a cache validation message would suffice. Speculative caching speeds up the execution of such commands by having the replicas assume that their cache is correct and verify after the execution. Finally, with state prefetching, servers of different partitions proactively propagate the state needed to execute multi-partition commands before the state is needed, thus reducing execution time (i.e., when the execution of a command starts in a server, the server will likely have all state needed to execute the command).

To further reduce the response time of commands, we also propose Fast-SSMR, which is a replication approach that extends S-SMR by providing a different interface between clients and servers and ensuring slightly different properties. Fast-SSMR brings lower latency for both single- and multi-partition commands, at the cost of potentially exposing temporary inconsistencies to clients. The idea of Fast-SSMR is to provide a fast, *optimistic reply*, which is potentially inconsistent, while the client waits for the final, *conservative reply*. Conservative replies can be used by clients to verify optimistic replies. Fast-SSMR requires optimistic atomic multicast [9, 10], which delivers messages to replicas twice: once optimistically and once conservatively. The optimistic delivery is fast but may violate order across replicas; the conservative delivery is slower but guarantees order.

Fast-SSMR implements two state machines at each server replica: an *optimistic state machine*, which executes commands as they are optimistically delivered, and a *conservative state machine*, which waits for the conservative delivery to execute each command. Previous optimistic replicated designs (e.g., [11–14]) require replicas to recover from commands executed in the wrong order, when the optimistic and the conservative orders do not match. Fast-SSMR does not have this constraint: in case of order mismatches, a correct state is copied from the conservative state machine to the optimistic one. Moreover, our technique generalizes optimistic execution to partitioned-state systems, dealing with problems not faced in fully replicated optimistic approaches, such as handling dependencies between states of different partitions and the exchange of optimistic state.

The optimistic state machines can operate in two modes: with and without optimistic state exchange. Without exchanging optimistic state, the optimistic state machines responsible for different state partitions only exchange conservative state when executing commands, significantly simplifying the optimistic execution of multi-partition commands. On the other hand, when an optimistic state machine receives optimistic state of a remote partition, it must account for the possibility of the received state being invalid. This may happen even if the local optimistic delivery order of optimistic atomic multicast was correct, making the verification of the optimistic execution significantly more complex.

Applications that involve user interaction (e.g., social networks) are well-suited for Fast-SSMR: before the conservative execution finishes, the client can receive the likely result of the computation, based on the optimistic reply. Although Fast-SSMR can be configured to provide consistent replies only, by executing commands less aggressively (i.e., servers send an optimistic reply only if the optimistic delivery order is

confirmed, in which case it is not be necessary to wait for the conservative execution), we opted for a more aggressive approach that provides lower latency, at the cost of exposing optimism to the client. We base this decision on the importance of response time for user experience and business revenues [15–17].

The dual state machine approach used by Fast-SSMR is service-independent and does not require an environment that supports recoverable actions (e.g., a database) or the implementation of service-specific recovery. Running two state machines, however, requires additional processing and memory resources. We judge that this cost in resources is offset by the flexibility of the approach and the resulting simplification it brings in the design of applications. Moreover, one could argue that running two state machines as separate threads makes better use of multi-core architectures than SMR or S-SMR, which typically have only a single execution thread per replica. As for memory overhead, widely deployed techniques, such as copy-on-write [18], can mitigate the effects of maintaining two state machines. Finally, to minimize the cost of copying the conservative state onto the optimistic one, Fast-SSMR copies only the divergent portion of the state.

To assess the performance of S-SMR and Fast-SSMR, we developed Eyrie, a Java library that allows developers to implement partitioned-state services transparently, abstracting partitioning details. All communication between partitions is handled internally by Eyrie, including remote object transfers. We also developed Volery, a service that implements the API of Zookeeper [19], and Chirper, which is a scalable Twitter-like social-network application.

In the experiments we conducted with Volery and Chirper, throughput scaled with the number of partitions, in some cases linearly. In some deployments, Volery reached over 250 thousand commands per second, significantly outperforming Zookeeper, which served 45 thousand commands per second under the same workload. With Chirper, we demonstrated that Fast-SSMR provides a significant decrease of response time, at the cost of a low number of inconsistent optimistic replies.

The remainder of this paper is organized as follows. Section 2 describes our system model and provides some definitions. Section 3 presents state machine replication and the motivation for this work. Section 4 introduces S-SMR; describing in detail our scalable variant of SMR. Section 5 describes Fast-SSMR, including both execution modes. Section 6 details the implementation of Eyrie, Volery and Chirper. Section 7 presents and comments on the results found with the experiments. Section 8 surveys related work and Section 9 concludes the paper.

2 System model and definitions

We consider a distributed system consisting of an unbounded set of client processes $\mathcal{C} = \{c_1, c_2, ...\}$ and a bounded set of n server processes (replicas) $\mathcal{S} = \{s_1, ..., s_n\}$. Set $\mathcal{S}$ is divided into k disjoint groups of servers, $\mathcal{S}_1, ..., \mathcal{S}_k$. Processes are either *correct*, if they never fail, or *faulty*, otherwise. In either case, processes do not experience arbitrary behavior (i.e., no Byzantine failures).

Processes communicate by message passing, using either one-to-one or one-to-many communication. The system is asynchronous: there is no bound on message delay or on relative process speed. One-to-one communication uses primitives $send(p, m)$ and $receive(m)$, where m is a message and p is the process m is addressed to. If sender and receiver are correct, then every message sent is eventually received. One-to-many communication relies on reliable multicast, atomic multicast and optimistic atomic multicast, respectively defined in Sections 2.1, 2.2 and 2.3.[1]

2.1 Reliable multicast

Reliable multicast allows messages to be sent to a set of groups. To reliably multicast a message m to a set of groups γ, processes use primitive reliable-multicast(γ, m). Message m is delivered at the destinations with reliable-deliver(m). Reliable multicast has the following properties:

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

[1] Solving atomic multicast requires additional assumptions [20, 21]. In the following, we simply assume the existence of an atomic multicast oracle.

2.2 Atomic multicast

S-SMR relies on atomic multicast [8]. Like reliable multicast, atomic multicast also allows messages to be sent to a set γ of groups. It is defined by the primitives atomic-multicast(γ, m) and atomic-deliver(m). Let relation $\prec$ be defined such that $m \prec m'$ iff there is a destination that delivers m before m'. Atomic multicast guarantees the following properties:

- If a correct process multicasts m, then every correct process in γ delivers m *(validity)*.
- If a process delivers m, then every correct process in γ delivers m *(uniform agreement)*.
- For any message m, every process p in γ delivers m at most once, and only if some process has multicast m previously *(integrity)*.
- The order relation $\prec$ is acyclic *(atomic order)*.

Atomic broadcast is a special case of atomic multicast in which there is a single process group. Atomic multicast provides uniform agreement, whereas our choice of reliable multicast ensures only non-uniform agreement. Although there are implementations of reliable multicast that provide uniformity, those implementations require two communication steps for messages to be delivered, while messages can be delivered within a single communication step if uniformity is not enforced [22].

2.3 Optimistic atomic multicast

Fast-SSMR relies on optimistic atomic multicast [9, 10], which performs two deliveries for each multicast message m: an *optimistic delivery* and a *conservative delivery*. It is defined by primitives opt-amcast(γ, m), opt-deliver(m) and cons-deliver(m).

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

- If the given optimistic assumptions hold,[2] the order relation $\prec_{opt}$ does not contradict the relation $\prec$ *(optimistic order)*.

Finally, optimistic atomic multicast ensures the following:

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

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

We developed a simple benchmark to assess latency, throughput and the rate of mistakes of optimistic atomic multicast. We consider two groups, each group consisting of three Paxos [23] acceptor processes and one server process. Clients multicast 100-byte messages to those servers, which reply when opt-delivering and when cons-delivering each message. Even when the system was saturated, the optimistic latency was roughly half the conservative one, as shown in Figure 1 (top). Also, the rate of mistakes was fairly low, as the number of messages optimistically delivered in the correct order was over 93% at the tipping point (150 clients) of the throughput graph, and above 84%, even under peak load, which can be seen in Figure 1 (bottom). The environment used is described in Section 7 and a detailed analysis of the optimistic atomic multicast protocol used here can be found in [10].

[2] Which exactly are the optimistic assumptions depends on the implementation of optimistic atomic multicast. In the case of [9] and [10], the optimistic assumption is that each process can accurately measure the latency from every other process in the system.

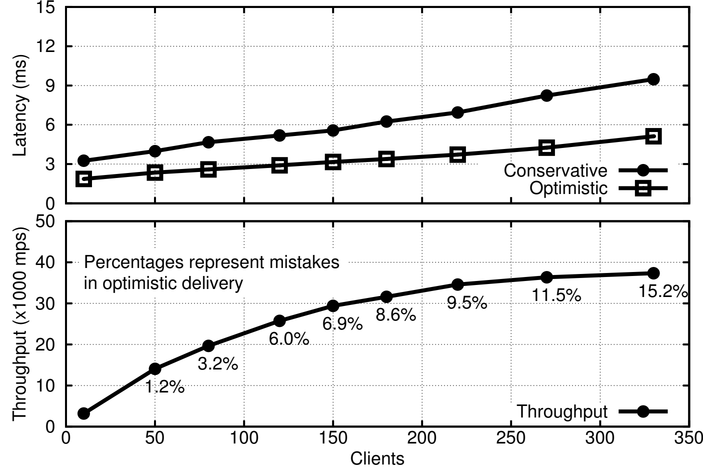


Figure 1: Latency (in milliseconds), throughput (in messages per second, or mps) and percentage of mistakes of optimistic atomic multicast.

3 Background and motivation

State machine replication is a fundamental approach to implementing a fault-tolerant service by replicating servers and coordinating the execution of client commands against server replicas [1,2]. The service is defined by a state machine, which consists of a set of *state variables* $\mathcal{V} = \{v_1, \ldots, v_m\}$ and a set of *commands* that may read and modify state variables, and produce a response for the command. Each command is implemented by a deterministic program. State machine replication can be implemented with atomic broadcast: commands are atomically broadcast to all servers, and all correct servers deliver and execute the same sequence of commands.

We are interested in implementations of state machine replication that ensure linearizability. Linearizability is defined with respect to a sequential specification. The *sequential specification* of a service consists of a set of commands and a set of *legal sequences of commands*, which define the behavior of the service when it is accessed sequentially. In a legal sequence of commands, every response to the invocation of a command immediately follows its invocation, with no other invocations or responses in between them. For example, a sequence of operations for a read-write variable v is legal if every read command returns the value of the most recent write command that precedes the read, if there is one, or the initial value otherwise. An execution $\mathcal{E}$ is linearizable if there is some permutation of the commands executed in $\mathcal{E}$ that respects (i) the service's sequential specification and (ii) the real-time precedence of commands. Command C_1 precedes command C_2 if the response of C_1 occurs before the invocation of C_2 [24].

In classical state machine replication, throughput does not scale with the number of replicas: each command must be ordered among replicas and executed and replied to by every (correct) replica. Some simple optimizations to the traditional scheme can provide improved performance but not throughput scalability. For example, although update commands must be ordered and executed by every replica, only one replica can respond to the client, saving resources at the other replicas. Commands that only read the state must be ordered with respect to other commands, but can be executed by a single replica, the replica that will respond to the client.

Lacking throughput scalability is a fundamental limitation of SMR: while some optimizations may increase throughput by adding servers, the improvements are limited since fundamentally, the technique does not scale. In the next sections, we describe two extensions to SMR: S-SMR and Fast-SSMR. Under certain workloads, both extensions allow throughput to grow proportionally to the number of replicas. Fast-SSMR manages to reduce the response time of S-SMR by using optimistic execution.

4 Scalable State Machine Replication

In this section, we introduce Scalable State Machine Replication (S-SMR), discuss performance optimizations, and argue about S-SMR's correctness. In Section 4.1, we present an overview of S-SMR. In Section 4.2,

we detail its algorithm. In Section 4.3, we propose several performance optimizations. In Section 4.4, we argue about S-SMR's correctness.

4.1 General idea

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

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

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

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

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

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

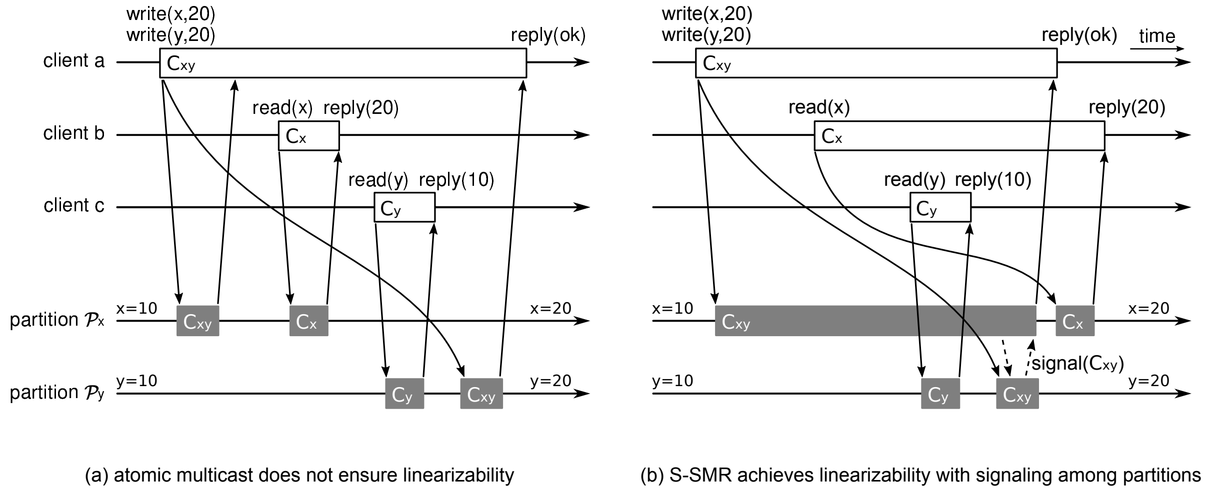
ii) <u>op is a $write(v, val)$ operation.</u>
If $\mathcal{P}_i \in part(v)$, s updates the value of v with val; if $\mathcal{P}_i \notin part(v)$, s executes op, creating a local copy of v, which will be up-to-date at least until the end of C's execution (caching is explained in Section 4.3.1).

iii) <u>op is a computation operation.</u>
In this case, s executes op.

As we now show, the procedure above is not enough to ensure linearizability. Consider the execution depicted in Figure 2 (a), where state variables x and y have initial value of 10. Command C_x reads the value of x, C_y reads the value of y, and C_{xy} sets x and y to value 20. Consequently, C_x is multicast to partition $\mathcal{P}_x$, C_y is multicast to $\mathcal{P}_y$, and C_{xy} is multicast to both $\mathcal{P}_x$ and $\mathcal{P}_y$. Servers in $\mathcal{P}_y$ deliver C_y and then C_{xy}, while servers in $\mathcal{P}_x$ deliver C_{xy} and then C_x, which is consistent with atomic order. In this execution, the only possible legal permutation for the commands is: C_y, C_{xy}, then C_x. However, this violates the real-time precedence of the commands, since C_x precedes C_y in real-time (C_y is invoked after the reply for C_x is received).

Intuitively, the problem with the execution in Figure 2 (a) is that commands C_x and C_y execute "in between" the execution of C_{xy} at partitions $\mathcal{P}_x$ and $\mathcal{P}_y$. In S-SMR, we avoid such cases by ensuring that the execution of every command is atomic. Command C is *execution atomic* if, for each server s that executes C, there exists at least one server r in every other partition in $part(C)$ such that the execution of C at s finishes after r delivers C. More precisely, let $delivery(C, s)$ and $end(C, s)$ be, respectively, the time when s delivers command C and the time when s completes C's execution. Execution atomicity ensures that, for every server s in partition $\mathcal{P}$ that executes C, there is a server r in every $\mathcal{P}' \in part(C)$ such that $delivery(C, r) < end(C, s)$. Intuitively, this condition guarantees that the execution of C at $\mathcal{P}$ and $\mathcal{P}'$ overlap in time.

Replicas can ensure execution atomicity by coordinating the execution of commands. After delivering command C, servers in each partition send a $signal(C)$ message to servers in the other partitions in $part(C)$. Before finishing the execution of C, each server must receive a $signal(C)$ message from at least one server



(a) atomic multicast does not ensure linearizability

(b) S-SMR achieves linearizability with signaling among partitions

Figure 2: Atomic multicast and S-SMR. (To simplify the figure, we show a single replica per partition.)

in every other partition that executes C. Moreover, if server s in partition $\mathcal{P}$ receives the value of a variable from server r in another partition $\mathcal{P}'$, as part of the execution of C, then s does not need to receive a $signal(C)$ message from servers in $\mathcal{P}'$. This way, to tolerate f failures, each partition requires $f+1$ servers; if all servers in a partition fail, service progress is not guaranteed.

Figure 2 (b) shows an execution of S-SMR. In the example, servers in P_x wait for a signal from P_y, therefore delaying C_{xy}'s execution in P_x and also postponing the execution of C_x. Note that the outcome of each command execution is the same as in case (a), but the executions of C_x, C_y and C_{xy}, as seen by clients, now overlap in time with one another. Hence, there is no real-time precedence among them and linearizability is not violated.

4.2 Detailed algorithm

The basic operation of S-SMR is shown in Algorithm 1. To submit a command C, the client queries an oracle to get set $C.dests$ (line 5), which is a superset of $part(C)$ used by the client as destination set for C (line 6).

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

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

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

4.3 Performance optimizations

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

Algorithm 1 Scalable State Machine Replication (S-SMR)

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

Algorithm variables:

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

$C.id$: unique identifier of command C

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

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

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

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

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

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

- Even though all replicas in all partitions in $part(C)$ execute C, a reply from a single replica (from a single partition) suffices for the client to finish the command.
- Servers can keep a cache of variables that belong to remote partitions; when multi-partition commands are executed and remote variables are received, this cache is verified and possibly updated.
- The exchange of objects between partitions serves the purpose of signaling. Therefore, if server s sends variable v's value to server r in another partition, r does not need to receive a signal message from s's partition.
- It is not necessary to exchange each variable more than once per command since any change during the execution of the command will be deterministic and thus any changes to the variable can be applied to the cached value.
- Server s does not need to wait for the execution of command C to reach a $read(v)$ operation to only then multicast v to the other partitions in $part(C)$. If s knows that v will be read by C, s can send v's value to the other partitions as soon as s starts executing C.
- As soon as a command C that reads a variable v is delivered, if there are no pending commands to be executed before C that change v, the value of v can already be sent to other partitions that deliver C.

The last two optimization are similar, as variables are sent proactively from servers of a partition to servers of another partition. They differ in the sense that, in the last one, the local variable's value is retrieved from the local state even before the command that accesses it is executed (possibly with a number of other commands waiting to be executed before)—we call this *state prefetching*.

4.3.1 Caching

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

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

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

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

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

4.3.2 State prefetching

When handling a multi-partition command C, partitions may need to exchange data so that every partition concerned by C has the state necessary to execute C. This exchange of data among partitions slows down the execution of multi-partition commands. Moreover, since commands are handled in delivery order, multi-partition commands may also delay commands that will come after them, a phenomenon sometimes referred to as *convoy effect* [6].

We reduce the delaying effects of multi-partition commands with *state prefetching*, a technique similar to hardware prefetching, used to speed up the execution of a program by reducing wait states in a processor. Instead of propagating data to partition $\mathcal{P}$ on a per-command basis, with state prefetching servers deliver a stream of commands and send to $\mathcal{P}$ the data needed to execute all multi-partition commands in the stream in a single batch, before the commands in the stream are executed in $\mathcal{P}$. This optimization improves performance in two ways. First, communication is more efficient as a consequence of batching the data to be propagated. Second, when a server in $\mathcal{P}$ starts the execution of a command, it may already have received the data needed by the command.

State prefetching introduces a complication: Let C be a command delivered before command C'. If the state of variable v needed to execute C' is sent by s before C and C' are executed and C modifies the value of v, the value sent by s will be invalid since it does not contain C's modifications. We cope with this problem by only allowing the value of v to be prefetched if no command delivered before C' (but not yet executed) will modify v. As a consequence, our implementation of state prefetching requires to determine the variables read and written by commands.

4.4 Correctness

In this section, we argue that, if every command in execution $\mathcal{E}$ of S-SMR is delivered by atomic multicast and is execution atomic, then $\mathcal{E}$ is linearizable.

Theorem 1. *S-SMR ensures linearizability.*

Proof. As defined in Section 2.2, we denote the order given by atomic multicast by relation $\prec$. Given any two messages m_1 and m_2, "$m_1 \prec m_2$" means that both messages are delivered by the same processes and m_1 is delivered before m_2, or there is some message m' such that $m_1 \prec m'$ and $m' \prec m_2$, which can be written as $m_1 \prec m' \prec m_2$.

Suppose, by means of contradiction, that there exist two commands x and y, where x finishes before y starts, but $y \prec x$ in the execution. There are two possibilities to be considered: (i) x and y are delivered by the same process p, or (ii) no process delivers both x and y.

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

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

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

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

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

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

5 Fast Scalable State Machine Replication

Fast-SSMR extends S-SMR by allowing clients to receive replies from servers within a lower time by means of optimistic execution. In Section 5.1, we present baseline Fast-SSMR, a basic version of our optimistic technique. We provide a detailed algorithm in Section 5.2. In Section 5.3, we extend the baseline Fast-SSMR algorithm with a technique to speed up the execution of multi-partition commands, by allowing

partitions to exchange optimistic state when executing multi-partition commands. Section 5.4 argues for the correctness of the algorithm.

5.1 Baseline Fast-SSMR

The baseline Fast-SSMR is inspired by previous optimistic designs to reduce the latency of replication (e.g., [11–14]). Some replicated systems share a similar execution pattern to exploit optimistic message ordering: (a) commands are exposed to the service before their order is final (i.e., optimistically delivered); (b) commands are optimistically executed by the service while the ordering protocol determines the final order of the commands (i.e., conservative delivery); (c) if the optimistic and the conservative orders do not match, commands must be re-executed in the order established by the conservative delivery. Like other optimistic replicated systems, baseline Fast-SSMR reduces latency by overlapping the execution of commands with their ordering. Differently from previous techniques, Fast-SSMR generalizes optimistic execution to partitioned-state systems, while providing linearizability.

Optimistic replicated designs require the ability to recover from commands executed in the wrong order. Previous approaches have tackled this issue by relying on the recoverability of the environment in which the service is deployed (e.g., replicated databases [11, 12]) or by explicitly implementing a recovery procedure, either “physical recovery" (e.g., checkpoints [13]) or “logical recovery" (e.g., application-dependent recovery [14]). We propose a different technique, consisting of two state machines that run concurrently: a *conservative state machine*, which implements S-SMR and whose execution is always correct, and an *optimistic state machine*, which relies on optimistic delivery to speed up the execution. Repairing the state of the optimistic state machine, in case of commands delivered out-of-order, boils down to copying the conservative state (or part of it) over the optimistic state. This dual state machine approach is service-independent and does not require an environment that supports recoverable actions (e.g., a database) or the implementation of service-specific recovery, although it requires additional processing and memory resources. Since current servers are typically multi-processor, each state machine can be assigned to a different processor, lessening the processing overhead. The memory overhead is mitigated by widely deployed OS techniques like copy-on-write [18]. Finally, we note that Fast-SSMR only copies the divergent portion of the state when performing a repair.

Most of the complexity of the Fast-SSMR algorithm deals with repairing the optimistic state in case of order violations. When an order violation is detected, the optimistic state machine stops the execution of commands, but to repair the optimistic state, the conservative state machine needs to “catch up" with the optimistic state machine. From the equivalence property of optimistic atomic multicast (defined in Section 2.3, eventually every command delivered optimistically by a correct process will be delivered conservatively. Therefore, once the optimistic execution stops, eventually the conservative state machine will catch up. After the optimistic state is repaired, any optimistically delivered command that is already included in the copied state is discarded.

Besides the repair procedure, the two state machines differ in another important aspect. When executing a *read*(v) operation in a multi-partition command, the optimistic state machines in the concerned partitions exchange the conservative state of v, instead of its optimistic state. This procedure simplifies the optimistic state machine as only valid state is exchanged across partitions—we introduce optimistic state exchange in Section 5.3. Both the optimistic and the conservative state machines send replies to clients; the conservative reply for a command allows the client to confirm the optimistic reply, likely to have been received earlier.

5.2 Detailed baseline algorithm

Algorithm 2 is executed by the conservative state machine. It is essentially the same as that of S-SMR (Algorithm 1), except that it uses optimistic atomic multicast instead of atomic multicast, and it also keeps track of the conservative execution order. Algorithm 3 details the execution of the optimistic state machine, which shares variables of the conservative state machine's algorithm. For brevity, we say that a command is “opt-executed" and “cons-executed" meaning that it was executed by the optimistic and the conservative state machines, respectively.

The core idea behind Algorithm 2 and Algorithm 3 is to compare the sequence of commands executed by the optimistic state machine to the sequence of commands executed by the conservative state machine, one by one. Those sequences are kept in the ordered sets $opt_executed$ and $cons_executed$, respectively.

Algorithm 2 Conservative State Machine (Fast-SSMR)

```
1: Initialization:
2:   cons_executed ← empty ordered set
3:   ∀C ∈ K : rcvd_signals(C) ← ∅
4:   ∀C ∈ K : rcvd_variables(C) ← ∅
5: Command C is submitted by a client as follows:
6:   C.dests ← oracle(C)
7:   opt-amcast(C.dests, C)
8:   wait for reply
9: Server s of partition P executes command C as follows:
10:  when cons-deliver(C)
11:    reliable-multicast(C.dests, signal(C))
12:    for each operation op in C do
13:      if op is read(v) then
14:        if v ∈ P then
15:          reliable-multicast(C.dests, ⟨v, C.id⟩)
16:        else
17:          wait until v ∈ rcvd_variables(C)
18:          update v with the value in rcvd_variables(C)
19:      execute op
20:    wait until rcvd_signals(C) = C.dests
21:    send reply to client
22:    append C to cons_executed
23:  when reliable-deliver(signal(C)) from partition P′
24:    rcvd_signals(C) ← rcvd_signals(C) ∪ {P′}
25:  when reliable-deliver(⟨v, C.id⟩)
26:    rcvd_variables(C) ← rcvd_variables(C) ∪ {v}
```

Algorithm variables:

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

$C.id$: unique identifier of command C

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

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

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

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

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

$cons_executed$: commands executed, in order of execution

Commands are appended to those sets as they are executed by the different state machines, and are removed as the optimistic execution order is confirmed (or a mistake is detected and the optimistic state is repaired). If the optimistic execution order at a server was the same as the conservative one, the optimistic state is correct. This condition suffices to determine that the optimistic state at a server is correct because mistaken optimistic deliveries at a server cannot cause the optimistic state of other servers to be incorrect—baseline Fast-SSMR does not exchange optimistic state between servers when executing multi-partition commands; instead, it uses the conservative state exchanged by the conservative state machines. This way, when the first element of both $opt_executed$ and $cons_executed$ is the same, this means that both state machines reached the same state after executing that command, which is removed from both sets. On the other hand, if the first element of $opt_executed$ differs from that of $cons_executed$, this means that there was an ordering mistake and the *repairing* flag is set to $True$, signaling that the optimistic state machine is under repair and no command will be opt-executed before the repair is finished. To repair its state, the optimistic state machine first waits until the conservative state machine has caught up with the optimistic execution, that is, until $opt_executed = \emptyset$ (commands are removed from that set as they are executed by the conservative state machine). This is sure to eventually happen at any correct replica thanks to the equivalence property of optimistic atomic multicast: any command that is opt-delivered is also cons-delivered, and vice-versa. Once the command is cons-delivered, it is executed by the conservative state-machine.

Algorithm 3 Optimistic State Machine (Fast-SSMR) (part 1)

```
1: Initialization:
2:   opt_queue ← opt_executed ← empty ordered set
3:   repairing ← False, skip_opt ← ∅
4:   ∀C ∈ 𝒦 : rcvd_opt_signals(C) ← ∅
5: Server s of partition 𝒫 executes command C as follows:
6:   when opt-deliver(C)
7:     reliable-multicast(C.dests, opt_signal(C))
8:     append C to opt_queue
9:   when opt_queue ≠ ∅ ∧ not repairing
10:    remove first element C of opt_queue
11:    if C ∉ skip_opt then
12:      append C to opt_executed
13:      execute-opt(C)
14:  when opt_executed ≠ ∅ ∧ not repairing
15:    C ← first element of opt_executed
16:    if C is the first element of cons_executed then
17:      // C was correctly opt-executed
18:      remove C from opt_executed
19:      remove C from cons_executed
20:    else if cons_executed ≠ ∅ then
21:      repairing ← True
22:  when repairing ∧ ∃C : C ∈ opt_executed ∩ cons_executed
23:    remove C from opt_executed
24:    remove C from cons_executed
25:  when repairing ∧ opt_executed = ∅
26:    copy conservative state onto optimistic state
27:    skip_opt ← skip_opt ∪ cons_executed
28:    cons_executed ← ∅
29:    repairing ← False
30:  when reliable-deliver(opt_signal(C)) from partition 𝒫′
31:    rcvd_opt_signals(C) ← rcvd_opt_signals(C) ∪ {𝒫′}
```

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

```
34: Server s of partition P executes command C as follows:
35:   function execute-opt(C)
36:     for each operation op in C do
37:       if op is read(v) then
38:         if v ∉ P then
39:           wait until v ∈ rcvd_variables(C) ∨ repairing
40:           if repairing then
41:             exit function
42:           v_opt ← v : v ∈ rcvd_variables(C)
43:         execute read(v_opt)
44:       else if op is write(v, val) then
45:         execute write(v_opt, val)
46:       else
47:         execute op
48:     wait until rcvd_opt_signals(C) = C.dests
49:     send optimistic reply to client
```

Additional variables:

v_{opt}: the optimistic copy of variable v

repairing: tells whether the optimistic state is under repair

opt_queue: commands waiting to be opt-executed

opt_executed: commands whose optimistic execution has not been confirmed yet

opt_signal(C): signal exchanged to provide linearizability

rcvd_opt_signals(C): partitions that signaled $\mathcal{P}$ about C

skip_opt: commands to be skipped

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

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

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

In case (i), both state machines executed the same commands, although in different orders. The optimistic state machine then simply copies the conservative state and resumes execution with the next command opt-delivered. In case (ii), not only the order was different, but also the set of commands executed. To deal with this, those commands that were not opt-executed yet, but already cons-executed, are put in a set to be skipped by the optimistic state machine. If one of these commands changes the state of the service, the state copied from the conservative state machine already contains any such changes. Reexecuting them might cause the optimistic state to become incorrect. To illustrate how Algorithm 3 works, suppose the following events happen at a replica (the command execution sequences after each event are shown):

1. Command C_3 is opt-executed.

   $opt_executed = (C_3)$, $cons_executed = \emptyset$

2. Command C_1 is opt-executed.

   $opt_executed = (C_3, C_1)$, $cons_executed = \emptyset$

3. Command C_1 is cons-executed.

   $opt_executed = (C_3, C_1)$, $cons_executed = (C_1)$

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

4. Command C_2 is cons-executed ($opt_executed = (C_3), cons_executed = (C_2)$).

5. Command C_3 is cons-executed ($opt_executed = \emptyset, cons_executed = (C_2)$).

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

We now give a more detailed, "operational" description of baseline Fast-SSMR. In Algorithm 2, when a server opt-delivers a command (line 6) it appends the command to a queue (line 8) from which commands are later removed to be executed (lines 10–13). For every $read(v)$ operation during the execution of a command (line 36), the server checks whether v is local (line 38); if not, the server waits until the value of v is received from a remote conservative state machine. Once v has arrived, its value is used to update v_{opt}, the optimistic copy of v (line 42). When reading (or writing) a variable v, the optimistic state machine reads (or writes) v_{opt} instead, thus accessing only the optimistic state (lines 43–45). The optimistic reply is sent to the client that issued the command (line 49).

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

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

5.3 Optimistic state exchange

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

Figure 3 illustrates how exchanging optimistic state increases the complexity of the optimistic execution. The figure shows servers s_1, s_2 and s_3, respectively of partitions $\mathcal{P}_1$, $\mathcal{P}_2$ and $\mathcal{P}_3$, as they execute optimistically and conservatively commands C_1, C_2 and C_3. Server s_1 optimistically delivers and executes C_2 before C_1, but s_1 conservatively delivers and executes commands C_1 before C_2. Because of this, s_1 sends invalid state to s_2, which also executes C_2. Even though the optimistic and conservative delivery orders match at s_2, s_2 reaches an invalid state after executing C_2. Even commands that were optimistically delivered in the correct order in all replicas can be "contaminated" by remote invalid states. Consider, for example, command C_3, which involves s_2 and s_3. As we can see, the optimistic and conservative delivery orders match in both servers. The problem is that C_3 was executed by s_2 after C_2, which in turn was executed based on an invalid state received from s_1. This may cause the state read by C_3 from $\mathcal{P}_2$ (and sent to s_3) to be invalid.

For a server s to determine that the optimistic execution of a command C is correct, s must (i) optimistically deliver C in the correct order and (ii) receive a confirmation from each remote partition involved in the command that the state s received from the partition is valid. Moreover, to handle contaminated states, commands that precede C must be confirmed as well. For this reason, a server s only sends a confirmation regarding the local execution of command C to other partitions after the execution of every command C'

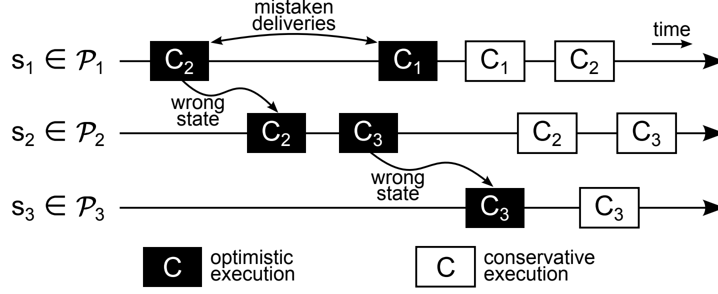


Figure 3: Invalid states propagated across optimistic state machines of different partitions.

executed by s before C is confirmed. If a server detects that the optimistic execution of a command was based on invalid state, the server switches to repairing mode and proceeds as described in Section 5.1.

Due to message delays and server failures, upon executing command C, a server s_1 of partition $\mathcal{P}_1$ may receive an optimistic state from s_2 in $\mathcal{P}_2$ and a confirmation from a different server s_3 in $\mathcal{P}_2$. Since the optimistic execution at s_2 and s_3 may diverge (e.g., if C's optimistic and conservative orders match at s_3 but they do not match at s_2), how can s_1 determine whether the state received from s_2 is valid? We handle this problem as follows. Since s_1 used the optimistic state sent by s_2, s_1 waits until it receives the confirmation from s_2, or until s_1 suspects that s_2 has failed. If s_1 suspects s_2's failure, s_1 assumes that the received state was incorrect and switches to repairing mode. Note that incorrectly suspecting a correct process has no implications on the correctness of the approach, only on its performance. Therefore, this mechanism can be implemented with a failure detector with *strong completeness* only, without *accuracy* [21].

5.4 Correctness

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

Theorem 2. *Fast-SSMR ensures linearizability.*

Proof. The conservative state machine guarantees linearizable executions, since it implements S-SMR. The optimistic state machine (Algorithm 3) ensures linearizability for optimistic replies, as long as the optimistic delivery order matches the conservative one. Both the conservative and optimistic state machines exchange signals in order to provide linearizability, as explained in detail in Section 4. □

Theorem 3. *Fast-SSMR is deadlock-free.*

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

Both optimistic and conservative deliveries have the agreement property (Section 2.1). Therefore, when executing a command C, all correct replicas of all partitions in $C.dests$ deliver C. Each replica will reliable-multicast a signal to all servers in $C.dests$ as soon as C is delivered. Therefore, from the validity property, every replica will reliable-deliver at least one signal from each partition in $C.dests$. This means that no replica will block forever waiting for a signal. This argument is valid for both the optimistic and the conservative state machines in case (i).

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

If the optimistic delivery order does not match the conservative delivery order, it is possible that an optimistic state machine s_{opt} (at server s) waits for a variable that will never be sent from a remote partition.

For example, say s_{opt}, due to a previous optimistic ordering mistake, has a variable *index* with value i, while its correct value is j, where $j \neq i$. Then, s_{opt} delivers C, which reads *index* and, then, reads v_{index}. Since *index* is i, s_{opt} tries to access v_i, which belongs to partition $\mathcal{P}_i$, thus waiting for its value to be received from a server in $\mathcal{P}_i$. However, this value will never be sent: no conservative state machine executes C with *index* equal to i. Therefore, s_{opt} never receives v_i. To circumvent this problem, Algorithm 3 stops waiting if an ordering mistake is detected (i.e., when *repairing* is set to $True$). When a mistake is detected, the optimistic state machine stops the execution of C (i.e., it exits the execute-opt function) and the optimistic state is repaired. This prevents deadlocks from happening in case (ii), even when optimistic delivery mistakes happen. In this example, s will not produce an optimistic reply for C, but only a conservative one. □

6 Implementation

In this section, we describe Eyrie, Volery and Chirper. Eyrie is a library that implements both S-SMR and Fast-SSMR. Volery is a service implemented with Eyrie that provides Zookeeper's API. Chirper is a scalable social network application also developed with Eyrie that exploits optimism to deliver faster replies to commands. Eyrie, Volery and Chirper were implemented in Java.

6.1 Eyrie

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

6.1.1 The PRObject class

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

Eyrie uses AspectJ[3] to intercept method calls for all subclasses of `PRObject`. Internally, the aspect related to such method invocations communicates with the `StateMachine` instance in order to (i) determine if the object is stored locally or remotely and (ii) ensure that the object is up-to-date when each command is executed. When executing Fast-SSMR, Eyrie also creates an optimistic copy of every object; when a command is optimistically delivered, it is executed against the optimistic copy of the objects, i.e., against the optimistic state.

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

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

[3] http://eclipse.org/aspectj

partitions to ensure linearizability, a write method that completely overwrites an object, regardless of its previous state, does not need an up-to-date version of the object before executing. Because of this, method `clear()` can be safely annotated as local, avoiding unnecessary communication between $\mathcal{P}_1$ and $\mathcal{P}_2$.

6.1.2 The StateMachine class

Linearizability is ensured by the `StateMachine` class: it executes commands one by one, in the order defined by atomic multicast, and implements the exchange of signals as described in Section 4. This class is abstract and must be extended by the application server class. To execute commands, the developer must provide an implementation for the method `executeCommand(Command c)`. The code for such a method is agnostic to the existence of partitions. In other words, it can be exactly the same as the code used to execute commands with classical state machine replication (i.e., full replication). Eyrie is responsible for handling all communication between partitions transparently. When executing Fast-SSMR there are two threads in Eyrie that call the `executeCommand` method: the conservative state machine thread, and the optimistic state machine thread. Again, this is transparent to the service designer, who simply writes one implementation of the method executeCommand, accessing objects as if there was a single copy of each of them. Internally, when Eyrie intercepts the method call to `PRObject` instances, it determines which thread (conservative or optimistic) made the call and redirects the method invocation to the proper (conservative or optimistic) copy of the object. This was implemented with the Java reflection API.

6.1.3 The PartitioningOracle class

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

Method `getDestinations(Command c)` is used by the clients to ask the oracle which partitions should receive each command. It returns a list of `Partition` objects. The developer can override this method, which will parse command `c` and return a list containing all partitions involved in the execution of `c`. The `getDestinations` method can encapsulate any kind of implementation, including one that involves communicating with servers, so its execution does not necessarily need to be local to clients. If the set of partitions involved in the execution of a command cannot be determined a priori, the oracle can communicate with servers to determine such set and then return it to the client, which then multicasts the command to the right partitions.

Another important method in `PartitioningOracle` is `getLocalObjects(Command c)`, which is used by servers to allow *state prefetching* before executing `c`. The method returns a list of local objects (i.e., objects in the server's partition) that will be accessed by `c`. It may not be possible to determine in advance which objects are accessed by every command, but the list of accessed objects does not need to be complete. Any kind of early knowledge about which objects need to be updated in other partitions helps decrease execution time, as the objects can potentially be sent as soon as the server delivers the command. The default implementation of this method returns an empty list, which means that objects are exchanged among partitions as their methods are invoked during execution (i.e., no state prefetching).

6.1.4 Other classes

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

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

To issue a command (i.e., a request), a client must create a `Command` object containing the command's parameters. The `Command` object is then multicast to the partitions determined by the partitioning oracle at the client. The `Command` class offers methods `addItems(Objects... objs)`, `getNext()`, `hasNext()`

and so on. How the server will process such parameters is application-dependent and determined by how the method `executeCommand` of the `StateMachine` class is implemented.

Eyrie uses atomic multicast to disseminate commands from clients and handle communication between partitions. This is done by configuring a `LocalReplica` object, which is created by parsing a configuration file provided by the developer, in both clients and servers. Eyrie is built on top of a multicast adaptor library,[4] being able to easily switch different implementations of atomic multicast, with and without optimistic deliveries. Thanks to Eyrie's flexibility, we deployed the applications with different atomic multicast implementations, namely Multi-Ring Paxos[5] and Ridge.[6] Fast-SSMR requires an optimistic atomic multicast protocol, which is implemented in Eyrie with Ridge.

6.2 Volery

We implemented the Volery service on top of Eyrie, providing an API similar to that of Zookeeper [19]. Zookeeper implements a hierarchical key-value store, where each value is stored in a *znode*, and each znode can have other znodes as children. The abstraction implemented by Zookeeper resembles a file system, where each path is a unique string (i.e., a key) that identifies a znode in the hierarchy. We implemented the following Volery client API:

- `create(String path, byte[] data)`: creates a znode with the given `path`, holding `data` as content, if there was no znode with that path previously and there is a znode with the parent path.
- `delete(String path)`: deletes the znode that has the given `path`, if there is one and it has no children.
- `exists(String path)`: returns True if there exists a znode with the given `path`, or False, otherwise.
- `getChildren(String path)`: returns the list of znodes that have `path` as their parent.
- `getData(String path)`: returns the data held by the znode identified by `path`.
- `setData(String path, byte[] data)`: sets the contents of the znode identified by `path` to `data`.

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

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

6.3 Chirper

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

- `post(long uid, String msg)`: user with id `uid` publishes message `msg`.
- `follow(long uid, long fid)`: user with id `uid` starts following user with id `fid`.

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

[5] With Multi-Ring Paxos (https://github.com/sambenz/URingPaxos), each message can be multicast to a single ring only. To map rings to partitions, each server in partition $\mathcal{P}_i$ is a learner in rings $\mathcal{R}_i$ and $\mathcal{R}_{all}$ (merge is deterministic); if message m is addressed only to $\mathcal{P}_i$, m is sent to $\mathcal{R}_i$, otherwise, to $\mathcal{R}_{all}$ (and discarded by non-addressee partitions).

[6] https://bitbucket.org/kdubezerra/ridge

- `unfollow(long uid, long fid)`: user with id `uid` stops following user with id `fid`.
- `getTimeline(long uid)`: user with id `uid` requests messages of all people the user follows.

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

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

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

We compared Chirper to Retwis,[7] which is another Twitter clone. Retwis relies on Redis,[8] a key-value store well-known for its performance. Redis offers asynchronous replication, which does not ensure linearizability. The Twitter API is implemented in the Retwis client as follows. The `post` commands update the list of all messages ever posted, the list of message ids of the user who is posting, and the materialized timeline (a list of message ids) of each follower of the poster. The `follow` and `unfollow` commands update the list of people followed by the user issuing the command, and the list of followers of another user. Finally, the `getTimeline` command accesses only the materialized timeline of the user who issued the command.

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

[7]http://retwis.antirez.com
[8]http://redis.io

7 Performance evaluation

In this section, we assess the performance of S-SMR and Fast-SSMR, in terms of throughput scalability and latency. For this purpose, we conducted experiments with Volery and Chirper.

With Volery, we evaluated S-SMR's throughput scalability by deploying the system with different numbers of partitions, message sizes, both with on-disk storage and with in-memory storage. Our goal was also to find the limits of S-SMR, showing when the technique does not perform as well, that is, when global commands are a significant part of the workload. We compare Volery results with Zookeeper and ZKsmr, which is an implementation of Zookeeper with traditional SMR.

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

7.1 Environment setup and configuration parameters

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

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

7.2 Volery

We compared Volery with the original Zookeeper and with ZKsmr, which is an implementation of the Zookeeper API using traditional state machine replication. Local commands consisted of calls to `setData`, while global commands were invocations to `create` and `delete`. "Message size" and "command size", in the next sections, refer to the size of the byte array passed to such commands. For the Zookeeper experiments, we used an ensemble of 3 servers. For the other approaches, we used Multi-Ring Paxos for atomic multicast, having 3 acceptors per ring: ZKsmr had 3 replicas that used one Multi-Ring Paxos ring to handle all communication, while Volery had 3 replicas per partition, with one ring per partition, plus one extra ring for commands that accessed multiple partitions. Since Zookeeper runs the service and the broadcast protocol (i.e., Zab [25]) in the same machines, each ZKsmr/Volery replica was co-located with a Paxos acceptor in the same node of the cluster. We had workloads with three different message sizes: 100, 1000 and 10000 bytes. Volery was run with 1, 2, 4 and 8 partitions. We conducted all experiments using disk for storage, then using memory (by means of a ramdisk). For on-disk experiments, we configured Multi-Ring Paxos with Δ of 40 ms [26], batching timeout of 50 ms and batch size threshold of 250 kilobytes; for in-memory experiments, these parameters were 5 ms, 50 ms and 30 kilobytes, respectively.

7.2.1 Experiments using on-disk storage

In Figure 4, we show results for local commands only. Each Paxos acceptor wrote its vote synchronously to disk before accepting each proposal. Zookeeper also persisted data to disk. In Figure 4 (top left), we can see the maximum throughput for each replication scheme and message size, normalized by the throughput

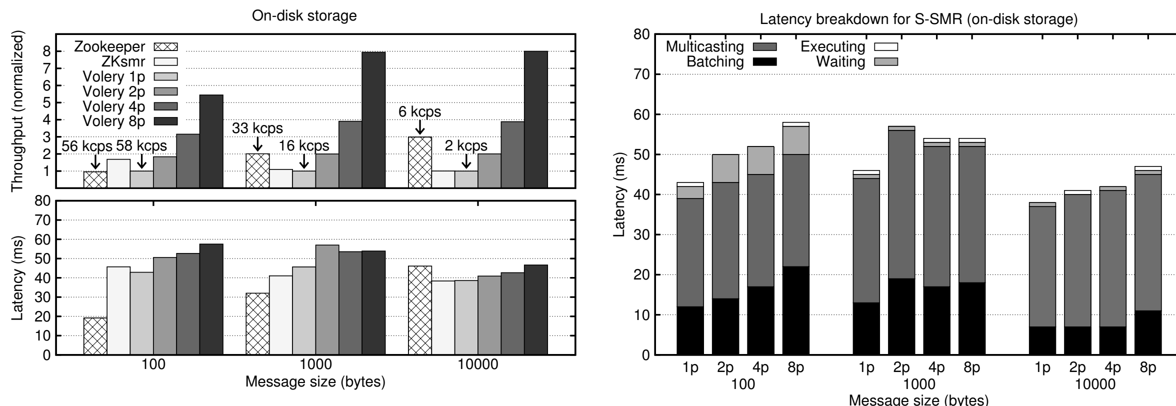


Figure 4: Results for Zookeeper, ZKsmr and Volery with 1, 2, 4 and 8 partitions, using on-disk storage. Throughput was normalized by that of Volery with a single partition (absolute values in kilocommands per second, or kcps, are shown). Latencies reported correspond to 75% of the maximum throughput.

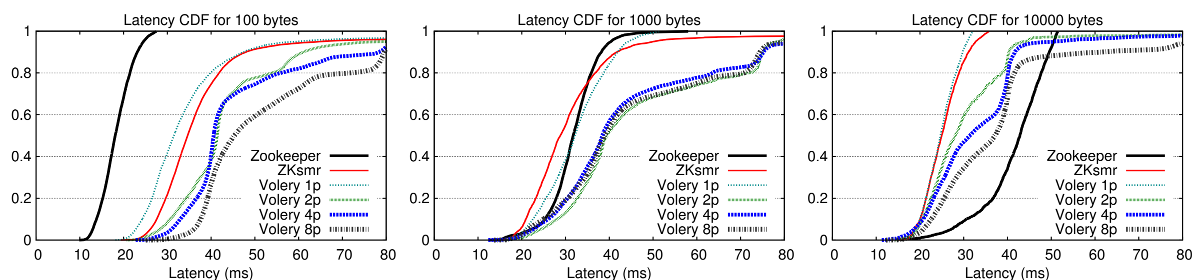


Figure 5: Cumulative distribution function (CDF) of latency for different command sizes (on-disk storage).

of Volery with a single partition. In all cases, the throughput of Volery scaled with the number of partitions and, for message sizes of 1000 and 10000 bytes, it scaled linearly (ideal case). For small messages (100 bytes), Zookeeper has similar performance to Volery with a single partition. As messages increase in size, Zookeeper's throughput improves with respect to Volery: with 1000-byte messages, Zookeeper's throughput is similar to Volery's throughput with two partitions. For large messages (10000 bytes), Zookeeper is outperformed by Volery with four partitions. Comparing S-SMR with traditional SMR, we can see that for small messages (100 bytes), ZKsmr performed better than Volery with one partition. This is due to the additional complexity added by Eyrie in order to ensure linearizability when data is partitioned. Such difference in throughput is less significant with bigger commands (1000 and 10000 bytes).

We can also see in Figure 4 (bottom left), the latency values for the different implementations tested. Latency values correspond to 75% of the maximum throughput. Zookeeper has the lowest latency for 100- and 1000-byte command sizes. For 10000-byte commands, Volery had similar or lower latency than Zookeeper. Such lower latency of Volery with 10000-byte commands is due to a shorter time spent with batching: as message sizes increase, the size threshold of the batch (250 kilobytes for on-disk storage) is reached faster, resulting in lower latency.

Figure 4 (right) shows the latency breakdown of commands executed by Volery. *Batching* is the time elapsed from the moment the client sends command C to the instant when C is proposed by the ring coordinator as part of a batch. *Multicasting* is the time from when the propose is executed until when the batch that contains C is delivered by a server replica. *Waiting* represents the time between the delivery of C and the moment when C starts executing. *Executing* measures the delay between the start of the execution of command C until the client receives C's response. We can see that more than half of the latency time is due to multicasting, which includes saving Multi-Ring Paxos instances synchronously to disk. There is also a significant amount of time spent with batching, done to reduce the number of disk operations and allow higher throughput: each Paxos proposal is saved to disk synchronously, so increasing the number of commands per proposal (i.e., per batch) reduces the number of times the disk is accessed. This allows performance to improve, but increases latency.

In Figure 5, we show the cumulative distribution functions (CDFs) of latency for all experiments where disk was used for storage. The results show that the latency distributions for ZKsmr and Volery with a single partition are similar, while latency had more variation for 2, 4 and 8 partitions. An important difference between deployments with a single and with multiple partitions is related to how Multi-Ring Paxos is used. In ZKsmr and in Volery with a single partition, there is only one Paxos ring, which orders all commands from all clients and delivers them to all replicas. When there are multiple partitions, each replica delivers messages from two rings: one ring that orders messages related to the replica's partition only, and another ring that orders messages addressed to more than one partition—each replica deterministically merges deliveries from multiple rings. As the time necessary to perform such deterministic merge is influenced by the level of synchrony of the rings, latency is expected to fluctuate more when merging is involved.

7.2.2 Experiments using in-memory storage

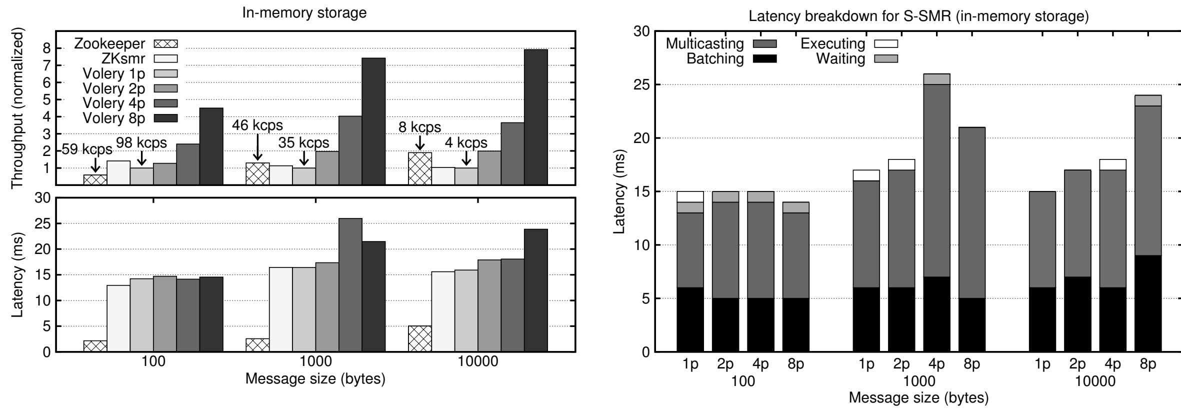


Figure 6: Results for Zookeeper, ZKsmr and Volery with 1, 2, 4 and 8 partitions, using in-memory storage. Throughput was normalized by that of Volery with a single partition (absolute values in kilocommands per second are shown). Latencies reported correspond to 75% of the maximum throughput.

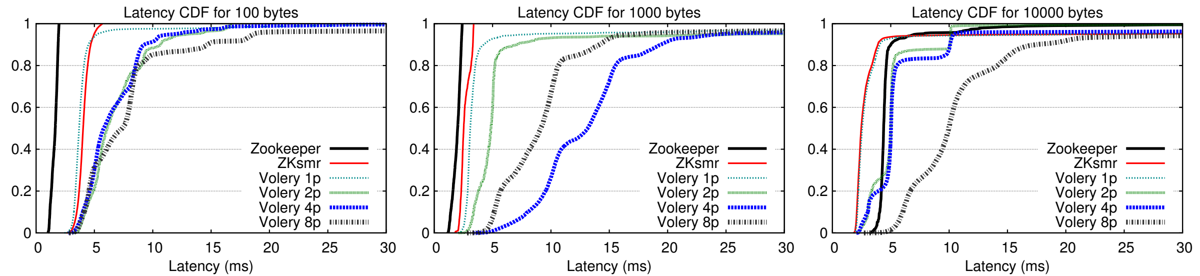


Figure 7: Cumulative distribution function (CDF) of latency for different command sizes (in-memory storage).

In Figure 6, we show the results for local commands when storing data in memory only. Volery's throughput scales with the number of partitions (Figure 6 (top left)), specially for large messages, in which case throughput grows linearly with the number of partitions (i.e., ideal case). We can also see that latency values for Volery and ZKsmr are less than half of what they are for on-disk storage (Figure 6 (bottom left)), while Zookeeper's latency decreased by an order of magnitude.

Figure 6 (right) shows the latency breakdown. Even though no data is saved to disk, multicasting is still responsible for most of the latency, followed by batching. Differently from the experiments described in Section 7.2.1, batching here had a size threshold of 30 kilobytes, which helps to explain why batching time is roughly the same for different message sizes. In these experiments, although there are no disk writes, batching is still used because it reduces the number of Paxos proposals and the number of messages sent through the network, which allows higher throughput. Figure 7 shows the latency CDFs for the in-memory experiments, where we can see that Volery with multiple partitions (i.e., deployments where Multi-Ring Paxos uses multiple rings) tends to have more variation in latency.

7.2.3 Experiments with global commands

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

We ran experiments with different rates of global commands (i.e., create and delete operations): 0%, 1%, 5% and 10% of all commands. We chose such rates for two reasons: (i) it is obvious that high rates of global commands will prevent the system from scaling, plus (ii) it is common for large scale services to have a high rate of read requests (which are local commands in Volery). An example of such a service is Facebook's TAO [27], which handles requests to a social graph; it allows, for instance, pages to be generated based on the user's connections in the social network. In Facebook's TAO, 99.8% of all requests are read-only [27].

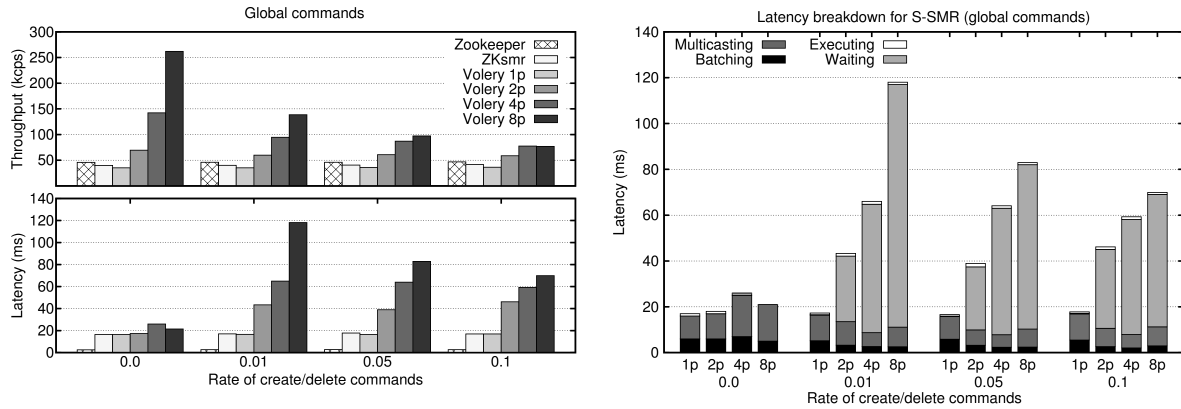


Figure 8: Throughput and latency versus rate of create/delete commands (in-memory storage, 1000-bytes commands). Throughput is shown in units of a thousand commands per second (kcps). Latencies shown corresponds to 75% of the maximum throughput.

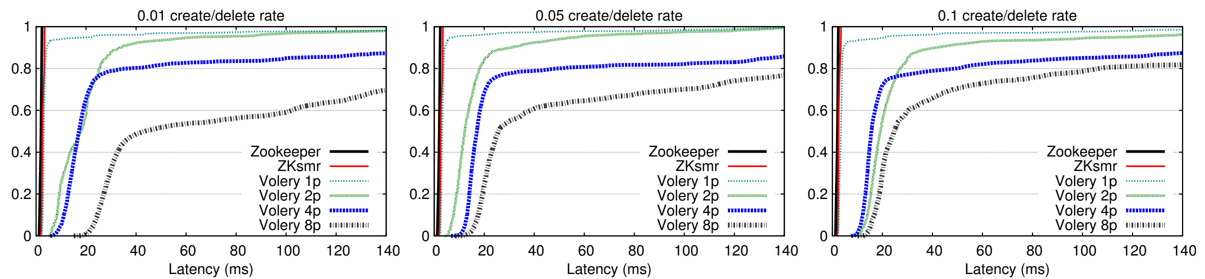


Figure 9: Cumulative distribution function (CDF) of latency for different rates of create/delete commands (in-memory storage, 1000-bytes commands).

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

Figure 9 shows the latency CDFs for the workloads with global commands. For experiments with more than one partition, the rate of messages with high latency is much higher than the rate of global commands. This happens due to a "convoy effect": local commands may be delivered after global commands, having to wait for the latter to finish.

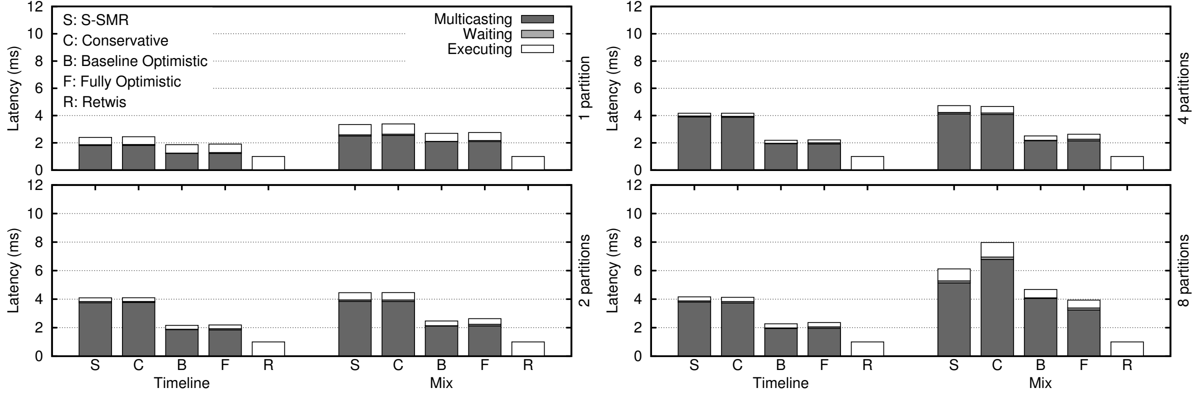


Figure 10: Latency components for different workloads, partitionings and levels of optimism.

7.3 Chirper

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

We deployed Retwis with a single Redis server, while Chirper was run with 1, 2, 4 and 8 partitions, with 3 replicas per partition. In our experiments with Chirper, we use Ridge [10] for multicast in experiments with both S-SMR and Fast-SSMR. Each Paxos ensemble in our Ridge deployment has 3 acceptors, with in-memory storage.

7.3.1 Latency improvement

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

Figure 10 shows the latency components for each algorithm tested. For each command C, the *Multicasting* component is the time between the moment when the client multicasts C and the moment when the server S that replies to C cons- or opt-delivers the command and puts it in an execution queue; *Waiting* is the time from when S delivers C until when S starts to execute it; *Executing* is the time from when S starts to execute C to when the client receives the reply. We can see that the multicast time of the optimistic approaches (i.e., Baseline and Fully Optimistic) is significantly lower (in proportion) than that of S-SMR and the conservative one of Fast-SSMR, which happens thanks to the lower latency of the optimistic atomic delivery. The Waiting component is nearly zero because the system is not saturated, so commands do not have to wait very long to be executed—when there are too many commands, they are put in a queue to be executed later. The execution time comprises the time it takes to execute the command itself, including any coordination across partitions. We can see that the Executing component is shorter for the Timeline workload, which happens because all requests are single-partition, requiring no cross-partition coordination. In the Mix workload, many requests are multi-partition, which require exchange of both signals and state between servers.

The optimistic state exchange optimization (Fully Optimistic) shows some latency improvement over the Baseline approach, although this is visible only as the number of partitions increases (i.e., eight partitions, in Figure 10, bottom right). In Figure 11, we can see the cumulative distribution functions (CDFs) of latency for each workload and partitioning. The latency difference between the different techniques becomes more evident, as both optimistic latencies are significantly lower than those of S-SMR and Conservative. As for the optimistic state exchange, it is expected to reduce latency only for the Mix workload with more than one partition, and we see improvements with eight partitions. Nevertheless, the Fully Optimistic latency is roughly the same as the Baseline Optimistic latency in the worst case.

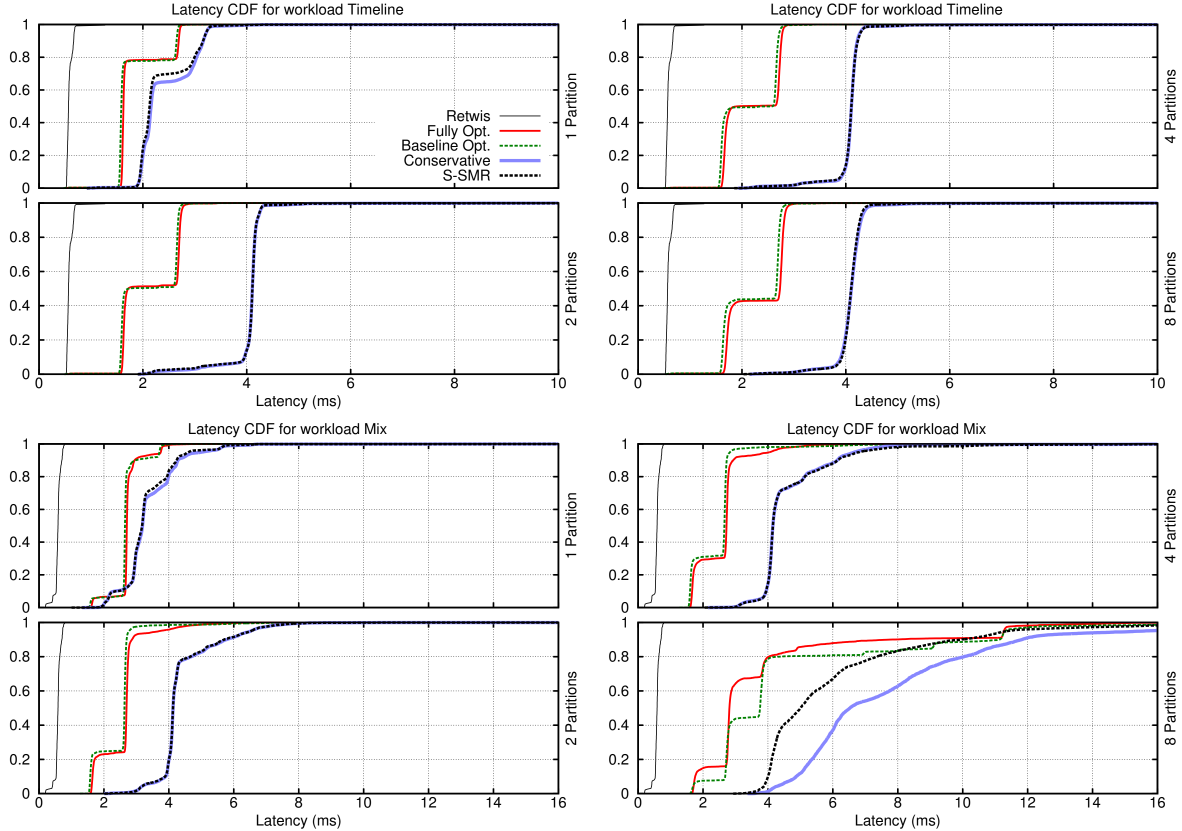


Figure 11: Cumulative distribution function (CDF) of latency for different workloads, partitionings and levels of optimism.

7.3.2 Throughput scalability

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

Figure 12 (left) shows throughput and latency results for Chirper when deployed with S-SMR. Throughput values for each workload are normalized by the throughput of Chirper with a single partition, whose absolute value (in kilocommands per second, or kcps) is shown in the graph. Figure 12 (right) shows the corresponding results for Chirper deployed with Fast-SSMR. In the case of Fast-SSMR, the optimistic and conservative latencies are both shown in Figure 12 (bottom right): the solid bottom of each bar represents the latency of the optimistic replies, while the hatched top represents the difference between the two latencies; the sum of the two is the conservative latency.

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

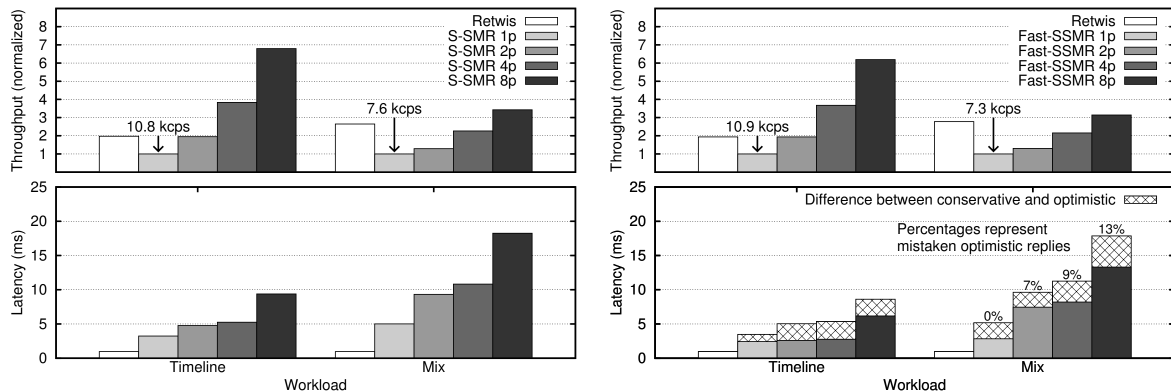


Figure 12: Scalability results for Chirper, implemented with S-SMR and Fast-SSMR, with 1, 2, 4 and 8 partitions. Retwis results are also shown. For each workload, the throughput was normalized by that of Chirper with a single partition (absolute values in kilocommands per second are shown). Latencies reported correspond to 75% of the maximum throughput.

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

The rate of mistakes is calculated by dividing the number of optimistic replies that differed from the conservative replies by the total number of optimistic replies received. The corresponding percentages are not shown for the Timeline workload: getTimeline is a read-only request and, thus, an execution composed only of getTimeline requests would never have incorrect replies sent to the client—even if the servers deliver the commands out-of-order. For an incorrect reply to be received, the workload must contain both requests that read and requests that update the service state, as in the Mix workload. We can see that, in all experiments, the percentage of mistaken replies was fairly low. The percentage of mistakes can be controlled by setting more or less aggressive parameters for the optimistic atomic multicast algorithm, with a trade-off between latency improvement and number of correct replies. For instance, waiting a bit longer before delivering a message optimistically may be enough for a process to receive a late message that would otherwise be optimistically delivered in the incorrect order [9].

8 Related Work

State machine replication is a well-known approach to replication and has been extensively studied (e.g., [1,2,28–30]). State machine replication requires replicas to execute commands deterministically, which implies sequential execution. Even though increasing the performance of state machine replication is non-trivial, different techniques have been proposed for achieving scalable systems, such as optimizing the propagation and ordering of commands (i.e., the underlying atomic broadcast algorithm). In [31], the authors propose to have clients sending their requests to multiple computer clusters, where each such cluster executes the ordering protocol only for the requests it received, and then forwards this partial order to every server replica. The server replicas, then, must deterministically merge all different partial orders received from the ordering clusters. In [32], Paxos [23] is used to order commands, but it is implemented in a way such that the task of ordering messages is evenly distributed among replicas, as opposed to having a leader process that performs more work than the others and may eventually become a bottleneck.

State machine replication seems at first to prevent multi-threaded execution since it may lead to non-determinism. However, some works have proposed multi-threaded implementations of state machine replication, circumventing the non-determinism caused by concurrency in some way. In [30], for instance, the authors propose organizing each replica in multiple modules that perform different tasks concurrently,

such as receiving messages, batching, and dispatching commands to be executed. The execution of commands is still sequential, but the replica performs all other tasks in parallel. We also implemented such kinds of parallelism in Eyrie.

Some works have proposed to parallelize the execution of commands in SMR. In [28], application semantics is used to determine which commands can be executed concurrently without reducing determinism (e.g., read-only commands can be executed in any order relative to one another). Upon delivery, commands are directed to a parallelizer thread that uses application-supplied rules to schedule multi-threaded execution. Another way of dealing with non-determinism is proposed in [29], where commands are speculatively executed concurrently. After a batch of commands is executed, replicas verify whether they reached a consistent state; if not, commands are rolled back and re-executed sequentially. Both [28] and [29] assume a Byzantine failure model and in both cases, a single thread is responsible for receiving and scheduling commands to be executed. In the Byzantine failure model, command execution typically includes signature handling, which can result in expensive commands. Under benign failures, command execution is less expensive and the thread responsible for command reception and scheduling may become a performance bottleneck.

Many database replication schemes also aim at improving the system throughput, although commonly they do not ensure strong consistency as we define it here (i.e., as linearizability). Many works (e.g., [7, 33–35]) are based on the deferred-update replication scheme, in which replicas commit read-only transactions immediately, not necessarily synchronizing with each other. This provides a significant improvement in performance, but allows non-linearizable executions to take place. The consistency criteria usually ensured by database systems are serializability [36] or snapshot isolation [37]. Those criteria can be considered weaker than linearizability, in the sense that they do not take into account real-time precedence of different commands among different clients. For some applications, this kind of consistency is good enough, allowing the system to scale better, but services that require linearizability cannot be implemented with such techniques.

Other works have tried to make linearizable systems scalable [14, 38, 39]. In [38], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In [14], a partitioned variant of SMR is proposed, supporting single-partition updates and multi-partition read operations. It relies on total order: all commands have to be ordered by a single sequencer (e.g., a Paxos group of acceptors), so that linearizability is ensured. The replication scheme proposed in [14] does not allow multi-partition update commands. Spanner [39] uses a separate Paxos group per partition. To ensure strong consistency across partitions, it assumes that clocks are synchronized within a certain bound that may change over time. The authors say that Spanner works well with GPS and atomic clocks.

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

Fast-SSMR proposes to use optimism to provide faster replies to client requests. Other works have used optimistic execution before. In [14, 40], optimism is implemented in the communication layer, which allows replicas to deliver messages early. The approach proposed in [41] uses speculative execution in the context of fully replicated state machines. It assumes that commands are independent (i.e., they do not access the same data item) to accelerate execution; when this assumption does not hold, the execution is rolled back and the command has to be multicast for execution again. An optimistic atomic broadcast protocol [42] is used in [11, 12] to certify update transactions. In [12], the system ensures snapshot isolation [37] in a partitioned database. Out-of-order optimistic deliveries may cause transactions to abort, but only if the out-of-order transactions conflict and one of them already started executing.

In Zyzzyva [13], each command is sent to a primary replica, which forwards it to a number of replicas. Those replicas immediately execute the command and send a reply to the client, along with an execution history. If the client receives enough replies with the same history, the order is guaranteed to be correct; otherwise, the client informs the replicas about the inconsistency, forcing them to execute a conservative total order protocol. MDCC [43] makes uses of Generalized Paxos [44] to reduce the number of round-trips between data centers when executing transactions. Generalized Paxos extends Fast Paxos [45], and both are optimistic consensus protocols that may achieve consensus in fewer communication rounds than Paxos. MDCC performs well with commutative operations and provides read-committed isolation [46]. The authors argue that the protocol could be extended to provide stronger consistency levels, up to serializability.

There are three main differences between those previous works that use speculative execution and what we propose here. First, Fast-SSMR accommodates any kind of (deterministic) multi-partition commands,

which requires a fairly sophisticated repair procedure in case of mistakes of the optimistic delivery. Second, in our dual state machine approach, the replica never stops executing commands or sending replies to clients; only the optimistic state machine may stop and not reply to some commands in case of an ordering mistake. Third, no rollback is necessary when repairing the optimistic state: it is simply copied from the conservative one; most complexity lies in determining which commands the optimistic state machine will execute after the repair is done.

9 Conclusion

This work introduces S-SMR, a variant of the well-known state machine replication technique that differs from previous related works in that it allows throughput to scale with the number of partitions without weakening consistency. We also introduce general principles of how to implement an optimistic execution in a partitioned-state system, while ensuring linearizability. The algorithm presented, Fast-SSMR, extends S-SMR and differs from previous works because (i) it does not require the application to be able to rollback executions, (ii) it allows any kind of workload (composed of deterministic commands) and any kind of partitioning of the state, and (iii) when a mistake is detected, the conservative state machine keeps handling commands and sending replies. The cost of this technique is to have two state machines running in parallel at each replica: a conservative one and an optimistic one. To evaluate both S-SMR and Fast-SSMR, we developed the Eyrie library and implemented Volery (a Zookeeper clone) and Chirper (a Twitter clone) with Eyrie. Our experiments demonstrate that in deployments with 8 partitions (the largest configuration we can deploy in our infrastructure) and under certain workloads, throughput experienced an 8-time improvement, resulting in ideal scalability. Moreover, Volery's throughput proved to be significantly higher than Zookeeper's. Chirper's optimistic replies have significantly lower latency than the conservative replies. This was also achieved thanks to the optimizations proposed here, which allow multi-partition commands to execute efficiently, even when servers have to exchange state.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, pp. 558–565, July 1978.

[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, pp. 299–319, Dec. 1990.

[3] A. D. Fekete and K. Ramamritham, "Replication," ch. Consistency Models for Replicated Data, pp. 1–17, Berlin, Heidelberg: Springer-Verlag, 2010.

[4] D. Sacca and G. Wiederhold, "Database partitioning in a cluster of processors," *ACM Transactions on Database Systems*, vol. 10, pp. 29–56, Mar. 1985.

[5] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-store: A high-performance, distributed main memory transaction processing system," *Proceedings of the VLDB Endowment*, vol. 1, pp. 1496–1499, Aug. 2008.

[6] N. Schiper, P. Sutra, and F. Pedone, "P-store: Genuine partial replication in wide area networks," in *Proceedings of the 2010 29th IEEE Symposium on Reliable Distributed Systems*, SRDS '10, (Washington, DC, USA), pp. 214–224, IEEE Computer Society, 2010.

[7] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[8] X. Défago, A. Schiper, and P. Urbán, "Total order broadcast and multicast algorithms: Taxonomy and survey," *ACM Computing Surveys*, vol. 36, pp. 372–421, Dec. 2004.

[9] C. E. Bezerra, F. Pedone, B. Garbinato, and C. Geyer, "Optimistic atomic multicast," in *Proceedings of the 2013 33rd IEEE International Conference on Distributed Computing Systems*, ICDCS '13, (Washington, DC, USA), pp. 380–389, IEEE Computer Society, 2013.

[10] C. E. Bezerra, D. Cason, and F. Pedone, "Ridge: high throughput, low-latency atomic multicast," in *Proceedings of the 2015 34th IEEE Symposium on Reliable Distributed Systems*, SRDS '15, (Washington, DC, USA), IEEE Computer Society, 2015.

[11] B. Kemme, F. Pedone, G. Alonso, and A. Schiper, "Processing transactions over optimistic atomic broadcast protocols," in *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, ICDCS '99, (Washington, DC, USA), pp. 424–, IEEE Computer Society, 1999.

[12] R. Jiménez-Peris, M. P. no Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters," in *Proceedings of the 22nd International Conference on Distributed Computing Systems*, ICDCS '02, (Washington, DC, USA), pp. 477–, IEEE Computer Society, 2002.

[13] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong, "Zyzzyva: Speculative byzantine fault tolerance," in *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, (New York, NY, USA), pp. 45–58, ACM, 2007.

[14] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *Proceedings of the 2011 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '11, pp. 454–465, June 2011.

[15] N. Bhatti, A. Bouch, and A. Kuchinsky, "Integrating user-perceived quality into web server design," *Computer Networks*, vol. 33, pp. 1–16, June 2000.

[16] A. Bouch, A. Kuchinsky, and N. Bhatti, "Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, (New York, NY, USA), pp. 297–304, ACM, 2000.

[17] X. S. Wang, A. Balasubramanian, A. Krishnamurthy, and D. Wetherall, "Demystifying page load performance with WProf," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, NSDI '13, (Berkeley, CA, USA), pp. 473–486, USENIX Association, 2013.

[18] J. M. Smith and G. Q. Maguire Jr., "Effects of copy-on-write memory management on the response time of unix fork operations," *Computing Systems*, vol. 1, no. 3, pp. 255–278, 1988. QC 20111107 NR 20140804.

[19] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale systems," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, (Berkeley, CA, USA), pp. 11–11, USENIX Association, 2010.

[20] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM*, vol. 32, pp. 374–382, Apr. 1985.

[21] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, pp. 225–267, Mar. 1996.

[22] N. Schiper, *On multicast primitives in large networks and partial replication protocols.* PhD thesis, Università della Svizzera italiana, 2009.

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

[24] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems*, vol. 12, pp. 463–492, July 1990.

[25] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proceedings of the 2011 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '11, (Washington, DC, USA), pp. 245–256, IEEE Computer Society, 2011.

[26] P. J. Marandi, M. Primi, and F. Pedone, "Multi-ring paxos," in *Proceedings of the 2012 42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '12, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2012.

[27] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani, "Tao: Facebook's distributed data store for the social graph," in *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, (Berkeley, CA, USA), pp. 49–60, USENIX Association, 2013.

[28] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, (Washington, DC, USA), pp. 575–, IEEE Computer Society, 2004.

[29] M. Kapritsos, Y. Wang, V. QuÃ©ma, A. Clement, L. Alvisi, and M. Dahlin, "Eve: Execute-Verify Replication for Multi-Core Servers," in *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, oct 2012.

[30] N. Santos and A. Schiper, "Achieving high-throughput state machine replication in multi-core systems," in *Proceedings of the 2013 IEEE 33rd International Conference on Distributed Computing Systems*, ICDCS '13, (Washington, DC, USA), pp. 266–275, IEEE Computer Society, 2013.

[31] M. Kapritsos and F. P. Junqueira, "Scalable agreement: Toward ordering as a service," in *Proceedings of the Sixth International Conference on Hot Topics in System Dependability*, HotDep '10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[32] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication," in *Proceedings of the 2012 31st IEEE Symposium on Reliable Distributed Systems*, SRDS '12, (Washington, DC, USA), pp. 111–120, IEEE Computer Society, 2012.

[33] P. Chundi, D. J. Rosenkrantz, and S. S. Ravi, "Deferred updates and data placement in distributed databases," in *Proceedings of the Twelfth International Conference on Data Engineering*, ICDE '96, (Washington, DC, USA), pp. 469–476, IEEE Computer Society, 1996.

[34] A. Sousa, R. Oliveira, F. Moura, and F. Pedone, "Partial replication in the database state machine," in *Proceedings of the 1st IEEE International Symposium on Network Computing and Applications*, NCA '01, (Washington, DC, USA), pp. 298–, IEEE Computer Society, 2001.

[35] T. Kobus, M. Kokocinski, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proceedings of the 2013 33rd IEEE International Conference on Distributed Computing Systems*, ICDCS '13, (Washington, DC, USA), pp. 286–296, IEEE Computer Society, 2013.

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

[37] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño Martínez, and J. E. Armendáriz-Iñigo, "Snapshot isolation and integrity constraints in replicated databases," *ACM Transactions on Database Systems*, vol. 34, pp. 11:1–11:49, July 2009.

[38] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Scalable consistency in scatter," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 15–28, ACM, 2011.

[39] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, pp. 8:1–8:22, Aug. 2013.

[40] P. Felber and A. Schiper, "Optimistic active replication," in *Proceedings of the The 21st International Conference on Distributed Computing Systems*, ICDCS '01, (Washington, DC, USA), pp. 333–, IEEE Computer Society, 2001.

[41] P. J. Marandi and F. Pedone, "Optimistic parallel state-machine replication," in *Proceedings of the 2014 33rd IEEE International Symposium on Reliable Distributed Systems*, SRDS '14, (Washington, DC, USA), pp. 57–66, IEEE Computer Society, 2014.

[42] F. Pedone and A. Schiper, "Optimistic atomic broadcast," in *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, (London, UK), pp. 318–332, Springer-Verlag, 1998.

[43] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete, "Mdcc: Multi-data center consistency," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, (New York, NY, USA), pp. 113–126, ACM, 2013.

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

[45] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006.

[46] A. Adya, B. Liskov, and P. O'Neil, "Generalized isolation level definitions," in *Data Engineering, 2000. Proceedings. 16th International Conference on*, pp. 67–78, 2000.