2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Scalable State-Machine Replication

Carlos Eduardo Bezerra*‡, Fernando Pedone*, Robbert van Renesse†
*University of Lugano, Switzerland
†Cornell University, USA
‡Universidade Federal do Rio Grande do Sul, Brazil

Abstract—**State machine replication (SMR) is a well-known technique able to provide fault-tolerance. SMR consists of sequencing client requests and executing them against replicas in the same order; thanks to deterministic execution, every replica will reach the same state after the execution of each request. However, SMR is not scalable since any replica added to the system will execute all requests, and so throughput does not increase with the number of replicas. Scalable SMR (S-SMR) addresses this issue in two ways: (i) by partitioning the application state, while allowing every command to access any combination of partitions, and (ii) by using a caching algorithm to reduce the communication across partitions. We describe Eyrie, a library in Java that implements S-SMR, and Volery, an application that implements Zookeeper's API. We assess the performance of Volery and compare the results against Zookeeper. Our experiments show that Volery scales throughput with the number of partitions.**

I. Introduction

Many current online services have stringent availability and performance requirements. High availability entails tolerating component failures and is typically accomplished with replication. For many online services, "high performance" essentially means the ability to serve an arbitrarily large load, something that can be achieved if the service can scale throughput with the number of replicas. State machine replication (SMR) [1], [2] is a well-known technique to provide fault tolerance without sacrificing strong consistency (i.e., linearizability). SMR regulates how client commands are propagated to and executed by the replicas: every non-faulty replica must receive and execute every command in the same order. Moreover, command execution must be deterministic. SMR provides configurable availability, by setting the number of replicas, but limited scalability: every replica added to the system must execute all requests; hence, throughput does not increase as replicas join the system.

Distributed systems usually rely on state partitioning to scale (e.g., [3], [4]). If requests can be served simultaneously by multiple partitions, then augmenting the number of partitions results in an overall increase in system throughput. However, exploiting state partitioning in SMR is challenging: First, ensuring linearizability efficiently when state is partitioned is tricky. To see why, note that the system must be able to order multiple streams of commands simultaneously (e.g., one stream per partition) since totally ordering all commands cannot scale. But with independent streams of ordered commands, how to handle commands that address multiple partitions? Second, SMR hides from the service designer much of the complexity involved in replication; all the service designer must provide is a sequential implementation of each service command. If state is partitioned, then some commands may need data from multiple partitions. Should the service designer introduce additional logic in the implementation to handle such cases? Should the service be limited to commands that access a single partition?

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 an atomic multicast primitive to consistently order commands within and across partitions. We show in the paper that simply ordering commands consistently across partitions is not enough to ensure strong consistency in partitioned state machine replication. S-SMR implements *execution atomicity*, a property that prevents command interleaves that violate strong consistency. To assess the performance of S-SMR, we developed Eyrie, a Java library that allows developers to implement partitioned-state services transparently, abstracting partitioning details, and Volery, a service that implements Zookeeper's API [5]. All communication between partitions is handled internally by Eyrie, including remote object transfers. In the experiments we conducted with Volery, throughput scaled with the number of partitions, in some cases linearly. In some deployments, Volery reached over 250 thousand commands per second, largely outperforming Zookeeper, which served 45 thousand commands per second under the same workload.

The paper makes the following contributions: (1) It introduces S-SMR and discusses several performance optimizations, including caching techniques. (2) It details Eyrie, a library to simplify the design of services based on S-SMR. (3) It describes Volery to demonstrate how Eyrie can be used to implement a service that provides Zookeeper's API. (4) It presents a detailed experimental evaluation of Volery and compares its performance to Zookeeper.

The remainder of this paper is organized as follows. Section II describes our system model. Section III presents state-machine replication and the motivation for this work. Section IV introduces S-SMR; we explain the technique in detail and argue about its correctness. Section V details the implementation of Eyrie and Volery. Section VI reports on the performance of the Volery service. Section VII surveys related work and Section VIII concludes the paper.

978-1-4799-2233-8/14 $31.00 © 2014 IEEE
DOI 10.1109/DSN.2014.41

IEEE computer society

II. 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 server processes $\mathcal{S} = \{s_1, ..., s_n\}$. Set $\mathcal{S}$ is divided into P disjoint groups of servers, $\mathcal{S}_1, ..., \mathcal{S}_P$. 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, as defined next. 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 atomic multicast, defined by the primitives *multicast*(γ, m) and *deliver*(m), where γ is a set of server groups. Let relation $\prec$ be defined such that $m \prec m'$ iff there is a server that delivers m before m'. Atomic multicast ensures that (i) if a server delivers m, then all correct servers in γ deliver m *(agreement)*; (ii) if a correct process multicasts m to groups in γ, then all correct servers in every group in γ deliver m *(validity)*; and (iii) relation $\prec$ is acyclic *(order)*.[1] The order property implies that if s and r deliver messages m and m', then they deliver them in the same order. Atomic broadcast is a special case of atomic multicast in which there is a single group with all servers.

III. 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* $\mathcal{V} = \{v_1, ..., v_m\}$ 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 invocation or response in between them. For example, a sequence of operations for a read-write variable v is legal if every read command returns the value of the most recent write command that precedes the read, if there is one, or the initial value otherwise. 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.

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 by every (non-faulty) replica. Some simple optimizations to the traditional scheme can provide improved performance but not 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.

This is a fundamental limitation: while some optimizations may increase throughput by adding servers, the improvements are limited since fundamentally, the technique does not scale. In the next section, we describe an extension to SMR that under certain workloads allows performance to grow proportionally to the number of replicas.

IV. Scalable State-Machine Replication

In this section, we introduce S-SMR, discuss performance optimizations, and argue about S-SMR's correctness.

A. General idea

S-SMR divides the application state $\mathcal{V}$ (i.e., state variables) into P partitions $\mathcal{P}_1, ..., \mathcal{P}_P$, where for each $\mathcal{P}_i$, $\mathcal{P}_i \subseteq \mathcal{V}$. Moreover, we require each variable v in $\mathcal{V}$ to be assigned to at least one partition and define $part(v)$ as the partitions that hold 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 $\mathcal{S}_i$.

To execute command C, the client multicasts C to all partitions that hold a variable read or updated by C. Consequently, the client must be able to determine the partitions accessed by C, denoted by $part(C)$. Note that this assumption does not imply that the client must know all variables accessed by C, but it must know the partitions these variables belong to. If the client cannot accurately estimate which partitions are accessed by C, it must determine 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

[1] Solving atomic multicast requires additional assumptions [6], [7]. In the following, we simply assume the existence of an atomic multicast 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 other 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) *op* is a $read(v)$ operation.
If $\mathcal{P}_i \in part(v)$, then s retrieves the value of v and sends it to every partition $\mathcal{P}_j$ that delivers C and does not hold v. If $\mathcal{P}_i \notin part(v)$, then s waits for v to be received from a server in a partition in $part(v)$.

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

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

As we now show, the procedure above does not ensure linearizability. Consider the execution depicted in Figure 1 (a), where state variables x and y have initial value of 10. Command C_x reads the value of x, C_y reads the value of y, and C_{xy} sets x and y to value 20. Consequently, C_x is multicast to partition $\mathcal{P}_x$, C_y is multicast to $\mathcal{P}_y$, and C_{xy} is multicast to both $\mathcal{P}_x$ and $\mathcal{P}_y$. Servers in $\mathcal{P}_y$ deliver C_y and then C_{xy}, while servers in $\mathcal{P}_x$ deliver C_{xy} and then C_x, which is consistent with atomic order. In this execution, the only possible legal permutation for the commands is C_y, C_{xy}, and C_x, which violates the real-time precedence of the commands, since C_x precedes C_y in real-time.

Intuitively, the problem with the execution in Figure 1 (a) is that commands C_x and C_y execute "in between" the execution of C_{xy} at partitions $\mathcal{P}_x$ and $\mathcal{P}_y$. In S-SMR, we avoid such cases by ensuring that the execution of every command is atomic. Command C is *execution atomic* if, for each server s that executes C, there exists at least one server r in every other partition in $part(C)$ such that the execution of C at s finishes 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 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 1 (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 moving the execution of C_x ahead in time. 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.

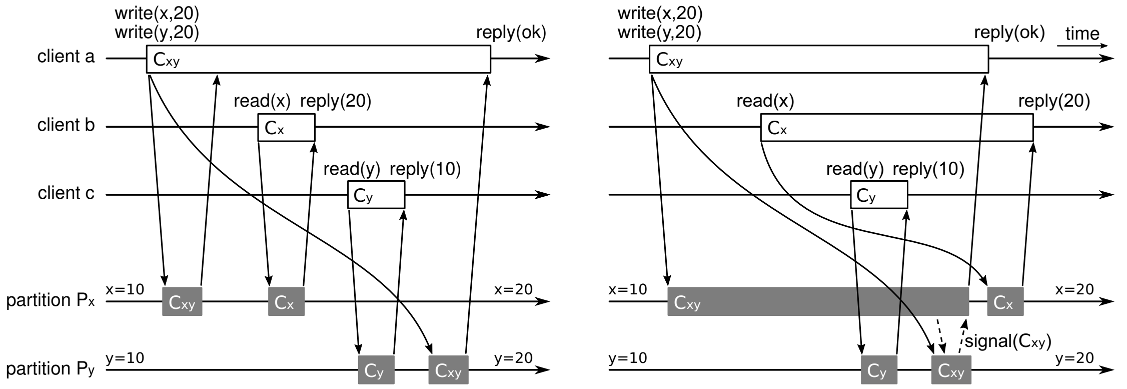
B. Detailed algorithm

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

Upon delivering C, 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 the sake of simplicity, Algorithm 1 simply 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,



(a) atomic multicast does not ensure linearizability

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

Figure 1. Atomic multicast and S-SMR. (To simplify the figure, we show a single replica per partition.)

only then, sends the reply back to the client (line 21). This ensures that C will be execution atomic.

C. Performance optimizations

Algorithm 1 can be optimized in many ways. In this section, we briefly mention some of these optimizations and then detail caching.

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

Server s in partition $\mathcal{P}$ can cache variables that belong to other partitions. There are different ways for s to maintain cached variables; here we define two techniques: conservative caching and speculative caching. In both cases, the basic operation is the following: When s executes a command that reads variable x from some other partition $\mathcal{P}_x$, after retrieving the value of x from a server in $\mathcal{P}_x$, s stores x's value in its cache and uses the cached value in future read operations. If a command writes x, s updates (or creates) x's local value. Server s will have a valid cache of x until (i) s discards the entry due to memory constraints, or (ii) some command not multicast to $\mathcal{P}$ changes the value of x. Since servers in $\mathcal{P}_x$ deliver all commands that access x, these servers know when any possible cached value of x is stale. How servers use cached entries distinguishes conservative from speculative caching.

Servers in $\mathcal{P}_x$ can determine which of its variables have a stale value cached in other partitions. This can be done by checking if there was any command that updated a variable x in $\mathcal{P}_x$, where such command was not multicast to some other partition $\mathcal{P}$ that had a cache of x. Say servers in $\mathcal{P}_x$ deliver command C, which reads x, and say the last command that updated the value of x was C_w. Since $x \in \mathcal{P}_x$, servers in $\mathcal{P}_x$ delivered C_w. One way for servers in $\mathcal{P}_x$ to determine which partitions need to update their cache of x is by checking which destinations of C did not receive C_w. This can be further optimized: even if servers in $\mathcal{P}$ did not deliver C_w, but delivered some other command C_r that reads x and C_r was ordered by multicast after C_w, then $\mathcal{P}$ already received an up-to-date value of x (sent by servers in $\mathcal{P}_x$ during the execution of C_r). If servers in $\mathcal{P}$ discarded the cache of x (e.g., due to limited memory), they will have to send a request for its value.

Conservative caching: Once s has a cached value of x, before it executes a $read(x)$ operation, it waits for a cache-validation message from a server in $\mathcal{P}_x$. The cache validation message contains a set of pairs (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, $\perp$, telling s that its cached value is up to date. If s discarded its cached copy, it sends a request for x to $\mathcal{P}_x$. If it is possible to determine which variables are accessed by C before C's execution, all such messages can be sent upon delivery of the command, reducing waiting time; messages concerning variables that could not be determined a-priori are sent later, during the execution of C, as variables are determined.

Algorithm 1 Scalable State-Machine Replication (S-SMR)

```
Initialization:
    $\forall C \in \mathcal{K} : rcvd\_signals(C) \leftarrow \emptyset$
    $\forall C \in \mathcal{K} : rcvd\_variables(C) \leftarrow \emptyset$

Command C is submitted by a client as follows:
    $dests \leftarrow oracle(C)$
    multicast($dests, C$)
    wait for response from one server

Command C is executed by a server in partition $\mathcal{P}$ as follows:
    upon deliver($C$)
        $others \leftarrow dests \setminus \{\mathcal{P}\}$
        multicast($others, signal(C)$)
        for each operation op in $C$ do
            if op is $read(v)$ then
                if $v \in \mathcal{P}$ then
                    multicast($others, \{v, C.id\}$)
                else
                    wait until $v \in rcvd\_variables(C)$
                    update $v$ with the value in $rcvd\_variables(C)$
            execute op
        wait until $rcvd\_signals(C) = others$
        send reply to client

    upon deliver($signal(C)$) from partition $\mathcal{P}'$
        $rcvd\_signals(C) \leftarrow rcvd\_signals(C) \cup \{\mathcal{P}'\}$

    upon deliver($\{v, C.id\}$)
        $rcvd\_variables(C) \leftarrow rcvd\_variables(C) \cup \{v\}$
```

Algorithm variables:

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

$C.id$: unique identifier of command C

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

$dests$: 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

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.

D. Correctness

In this proof, we denote the order given by atomic multicast with "$\prec$". Given any two messages m_1 and m_2, "$m_1 \prec m_2$" means that both messages are delivered by the same group 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 $m_1 \prec m' \prec m_2$.

We argue that, if every command in execution $\mathcal{E}$ of S-SMR is execution atomic, then $\mathcal{E}$ is linearizable. 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 for this: (i) x and y access some variable in common v, or (ii) x and y access no variable in common.

In case (i), at least one partition P_v (which contains v) delivers both x and y. As x finishes before y starts, then P_v delivers x, then y. From the properties of atomic multicast, and since each partition is mapped to a multicast group, no partition delivers y, then x. Moreover, atomic multicast ensures acyclic order, so there are no commands $z_1, ..., z_n$ such that their atomic order is $y \prec z_1 \prec \cdots \prec z_n \prec x$. So, we reached a contradiction in this case.

In case (ii), if there were no other command 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 $z_0, ..., z_n$ such that their atomic order is $y \prec z_0 \prec \cdots \prec z_n \prec x$. As $y \prec z_0$, then some partition P_y delivers y, then z_0. Also, since $z_0 \prec z_1$, then some partition P_1 delivers z_0, then z_1, and so on: partition P_i delivers z_{i-1}, then z_i, where $i \in \{1, ..., n\}$. Finally, partition P_x delivers z_n, then x.

We now claim that for every $R_x \in P_x$, R_x finishes executing x only after some $R_0 \in P_0$ delivered z_0. We prove our claim by induction.

Induction basis: As $z_n \prec x$, every $R_x \in Px$ executes command x only after the execution of z_n at R_x finished. As z_n is execution atomic, for every $R_x \in P_x$, there is a server $R_n \in P_n$ that delivered z_n before R_x finished executing z_n, which was before R_x even started executing command x. Therefore, every $R_x \in P_x$ finishes executing x only after some $R_n \in P_n$ delivered z_n.

Induction step: Assume that every $R_x \in P_x$ finishes executing x only after some $R_i \in P_i$ delivered z_i. As z_i is execution atomic, there is some $R_{i-1} \in P_{i-1}$ that delivered z_i before R_i finished executing z_i. As $z_{i-1} \prec z_i$ and R_{i-1} delivers both, then R_{i-1} delivers z_{i-1} before executing z_i. Thus, $R_x \in P_x$ finishes executing x only after some R_{i-1}

has delivered z_{i-1}, for every $R_x \in P_x$. This proves our claim.

Finally, for every $R_0 \in P_0$, there is a server $R_y \in P_y$ that delivers z_0 before R_0 finishes executing z_0. Since every R_y delivers and executes y, then z_0, and from the claim above, for every $R_x \in P_x$, there is an $R_y \in P_y$ that delivers y before R_x finishes executing x. This contradicts the assumption that x precedes y in real-time, i.e., that the reply to x is received before y is invoked by a client.

V. Implementation

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

A. Eyrie

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

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 user informs to Eyrie which object classes are partially replicated by extending the `PRObject` class. Each object of such class 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 user.

Eyrie uses AspectJ[2] to intercept method calls for all subclasses of `PRObject`. Internally, the aspect related to such method invocations communicates with the `StateMachine` instance in order to (i) determine if the object is stored locally or remotely and (ii) ensure that the object is up-to-date when each command is executed.

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

To avoid unnecessary communication, the user 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 user 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 P_1 and P_2. C completely overwrites objects O_1 and O_2, by calling `O1.clear()` and `O2.clear()`. Although C has to be delivered by both partitions to ensure linearizability, a write method that completely overwrites an object, regardless of its previous state, does not need an up-to-date version of the object before executing. Because of this, method `clear()` can be safely annotated as local, avoiding unnecessary communication between P_1 and P_2.

2) The `StateMachine` class: This class must be extended by the user's application server class. To execute commands, the user must provide an implementation for the method `executeCommand(Command c)`. The code for such a method is agnostic to the existence of partitions. In other words, it can be exactly the same as the code used to execute commands with classical state-machine replication (i.e., full replication). Eyrie is responsible for handling all communication between partitions transparently. To start the server, method `runStateMachine()` is called.

`StateMachine` ensures linearizability by making sure that each command is execution atomic (as defined in Section IV-A). As soon as each command C is delivered, `StateMachine` sends $signal(C)$ to all remote partitions that deliver C, in order to reduce waiting time. A command can only conclude its execution after it has received a signal from at least one server in every other partition that delivered the command—remote object updates received from other partitions count as signals for linearizability purposes.

In order to reduce communication, it is sufficient that a single replica in each partition sends object updates and signal messages to other partitions. If the designated replica fails, the other replicas in the same partition will suspect the failure and one of the operational replicas will retransmit the information.

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 user must provide an oracle that tells, for each command, which partitions are affected by

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

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 user to extend `PartitioningOracle` and override its methods.

Method `getDestinations(Command c)` is used by the oracle to tell what partitions should receive each command. It returns a list of `Partition` objects. The user 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 before executing `c`. The method returns a list of objects in the partition of the server that will be accessed by `c`. This list does not need to be complete, but any kind of early knowledge about what objects need to be updated in other partitions helps decrease execution time, as the objects can be sent as soon as the server starts executing 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. Depending on the application, the user may provide an implementation for this method.

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.

When sending commands, the client must multicast a `Command` object, which is serialized and sent to the partitions determined by the client's oracle. To the user, a command object is simply a container of objects, which are typically parameters for the command. 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 the user's implementation of `StateMachine`.

Eyrie uses atomic multicast to disseminate commands from clients and handle communication between partitions. Internally, it uses an implementation[3] of Multi-Ring Paxos [8]. 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).

B. Volery

We implemented the Volery service on top of Eyrie, providing an API similar to that of Zookeeper [5]. 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, Zookeeper is able to scale for workloads composed mainly of read-only requests. Volery ensures linearizability for every execution, regardless of what kind of commands are issued. In order to be scalable, Volery makes use of partitioning, done with Eyrie.

Distributing Volery's znodes among partitions was done based on each znode's path: a function $f(path)$ returned the id of the partition responsible for holding the znode at $path$. Function f is used by Volery's oracle to help clients determine which partitions must receive each command. 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 surely has its contents up-to-date.

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

VI. Performance Evaluation

In this section, we assess the performance of Volery with on-disk and in-memory deployments, and local and global commands.

A. Environment setup and configuration parameters

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

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

We compared Volery with the original Zookeeper and with ZKsmr, which is an implementation of the Zookeeper API using traditional state-machine replication. For the Zookeeper experiments, we used an ensemble of 3 servers. For the other approaches, we used Multi-Ring Paxos for atomic multicast, having 3 acceptors per ring: ZKsmr had 3 replicas that used one Paxos ring to handle all communication, while Volery had 3 replicas per partition, with one Paxos ring per partition, plus one extra ring for commands that accessed multiple partitions. Since Zookeeper runs the service and the broadcast protocol (i.e., Zab [9]) in the same machines, each ZKsmr/Volery replica was colocated with a Paxos acceptor in the same node of the cluster. We had workloads with three different message sizes: 100, 1000 and 10000 bytes. Volery was run with 1, 2, 4 and 8 partitions. We conducted all experiments using disk for storage, then using memory (by means of a ramdisk). For on-disk experiments, we configured Multi-Ring Paxos with Δ [8] of 40 ms, 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.

B. Experiments using on-disk storage

In Figure 2, we show results for local commands only. Each Paxos acceptor wrote its vote synchronously to disk before accepting each proposal. Zookeeper also persisted data to disk. In Figure 2 (top left), we can see the maximum throughput for each replication scheme and message size, normalized by the throughput of Volery with a single partition. 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 2 (bottom left), the latency values for the different implementations tested. Latency values correspond to 75% of the maximum throughput. Zookeeper has the lowest latency for 100- and 1000-byte command sizes. For 10000-byte commands, Volery had similar or lower latency than Zookeeper. Such lower latency of Volery with 10000-byte commands is due to a shorter time spent with batching: as message sizes increase, the size threshold of the batch (250 kilobytes for on-disk storage) is reached faster, resulting in lower latency.

Figure 2 (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 between the propose is executed until the batch that contains C is delivered by a server replica. *Waiting* represents the time between the delivery and the moment when C finally 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

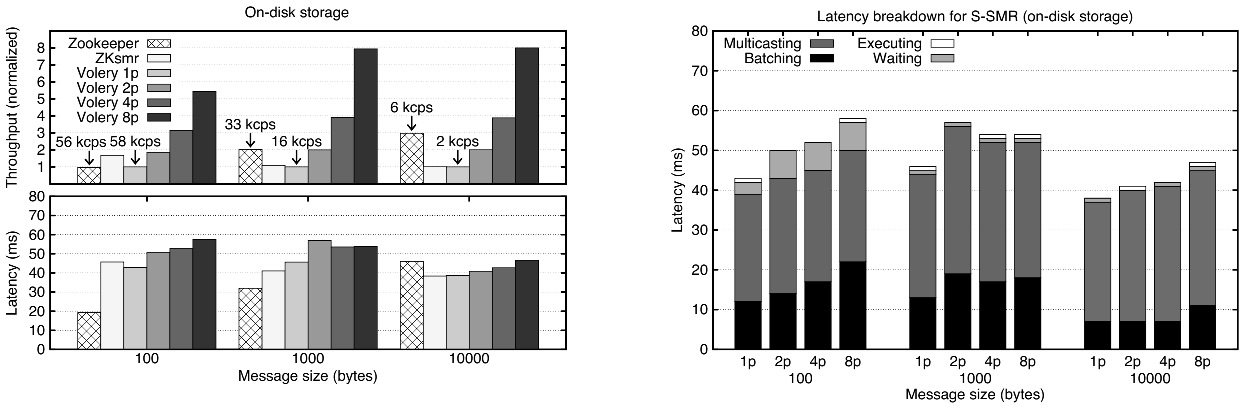


Figure 2. Results for Zookeeper, ZKsmr and Volery (with 1, 2, 4 and 8 partitions) using disk. Throughput was normalized by that of Volery with a single partition (absolute values in kilocommands per second are shown). Latencies reported correspond to 75% of the maximum throughput.

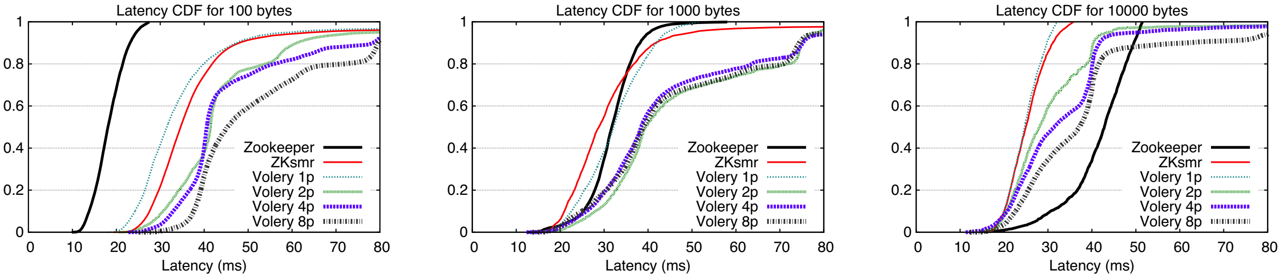


Figure 3. Cumulative distribution function (CDF) of latency for different command sizes (on-disk storage).

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

C. Experiments using in-memory storage

In Figure 4, we show the results for local commands when storing data in memory only. Volery's throughput scales with the number of partitions (Figure 4 (top left)), specially for large messages, in which case the scalability is linear 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 4 (bottom left)), while Zookeeper's latency decreased by an order of magnitude. These results suggest that further improvements should be achievable in the in-memory Volery configuration with additional optimizations and finer tuning of the atomic multicast parameters.

Figure 4 (right) shows the latency breakdown. Even though no data is saved to disk, multicasting is still responsible for most of the latency, followed by batching. Differently from the experiments described in Section VI-B, 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 5 shows the latency CDFs for the in-memory experiments, where we can see that Volery with multiple partitions (i.e., deployments where Multi-Ring Paxos uses multiple rings) tends to have more variation in latency.

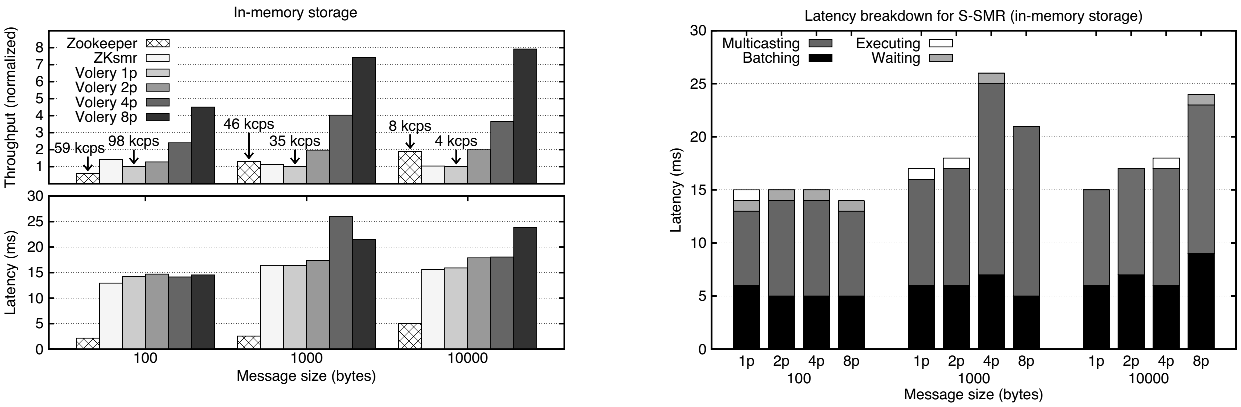


Figure 4. Results for Zookeeper, ZKsmr and Volery (with 1, 2, 4 and 8 partitions) using memory. Throughput was normalized by that of Volery with a single partition (absolute values in kilocommands per second are shown). Latencies reported correspond to 75% of the maximum throughput.

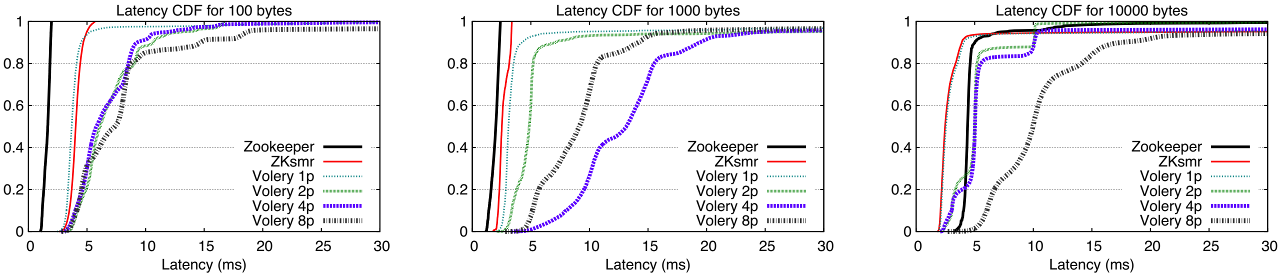


Figure 5. Cumulative distribution function (CDF) of latency for different command sizes (in-memory storage).

D. 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 reduces 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 [3], 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 [3].

We can see in Figure 6 (top left) that Volery scales throughput with the number of partitions for all configurations but the exceptional case of 10% of global commands when augmenting the number of partitions from 4 to 8. Moreover, Volery with two partitions outperforms the 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 6 (bottom left), as global commands are added to the workload: as the number of partitions increases, so does the average latency. As we can see in Figure 6 (right), this extra latency comes from the servers waiting for signals from other partitions.

Figure 7 shows the latency CDFs for the workloads with global commands. For experiments with more than one partition, the rate of messages with high latency is much higher than the rate of global commands. This happens due to a "convoy effect": local commands may be delivered after global commands, having to wait for the latter to finish.

VII. Related work

State-machine replication is a well-known approach to replication and has been extensively studied (e.g., [1], [2], [10], [11], [12]). 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

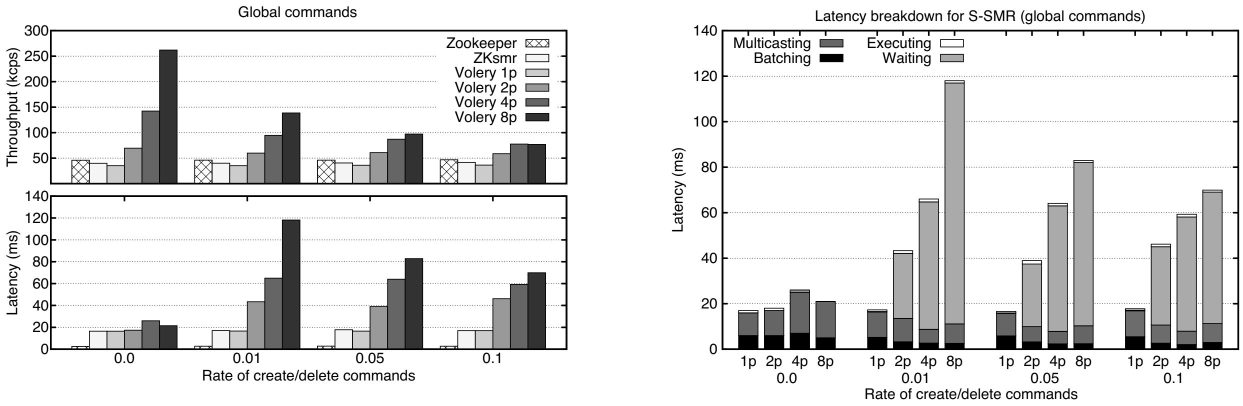


Figure 6. Throughput and latency versus rate of create/delete commands (in-memory storage, 1000-bytes commands). Throughput is shown in units of a thousand commands per second (kcps). Latencies shown corresponds to 75% of the maximum throughput.

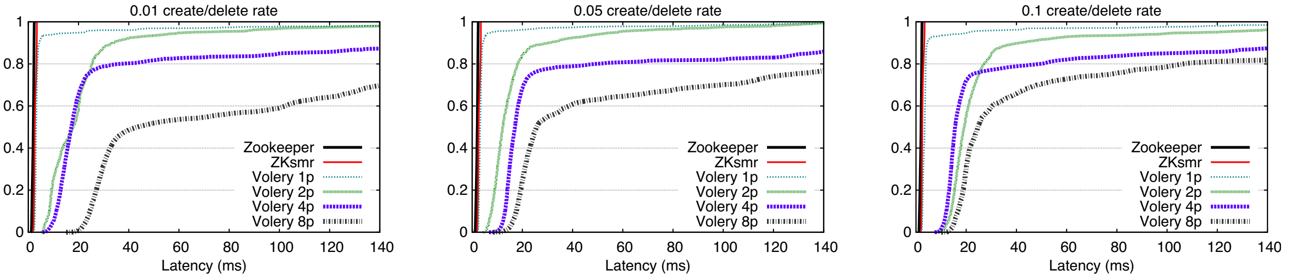


Figure 7. Cumulative distribution function (CDF) of latency for different rates of create/delete commands (in-memory storage, 1000-bytes commands).

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 [13], the authors propose to have clients sending their requests to multiple clusters, where each of such clusters 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 [14], Paxos [15] 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 [10], for instance, the authors propose organizing the 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 done sequentially, by a single thread, but the replica performs all other tasks in parallel. We also implemented such kind of parallelism in Eyrie.

Some works have proposed to parallelize the execution of commands in SMR. In [11], 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 [12], 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 [11] and [12] 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.

In [16], the authors propose to partition the service state and use atomic broadcast to totally order commands submitted by the clients. To ensure that linearizability holds for read-only commands that span multiple partitions, there is a single sequencer that ensures acyclic order; therefore, the approach cannot scale with the number of partitions, as the sequencer eventually becomes a bottleneck. Also, the approach can only handle single-partition update commands.

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., [4], [17], [18], [19]) 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 [20] and snapshot isolation [21]. 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.

VIII. Conclusion

This work introduces S-SMR, a scalable variant of the well-known state-machine replication technique. S-SMR differs from previous related works in that it allows throughput to scale with the number of partitions without weakening consistency. To evaluate S-SMR, we developed the Eyrie library and implemented Volery, a Zookeeper clone, with Eyrie. Our experiments demonstrate that in deployments with 8 partitions (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.

Acknowledgements

This work was supported in part by the Swiss National Science Foundation under grant number 146404. Robbert van Renesse is supported in part by grants from AFOSR, NSF, DARPA, ARPA-e, MDCN/iAd, Microsoft Corporation, Facebook Inc., and Amazon.com.

References

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

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

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

[4] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *DSN*, 2012.

[5] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *USENIX ATC*, 2010.

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

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

[8] P. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," in *DSN*, 2012.

[9] F. Junqueira, B. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *DSN*, 2011.

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

[11] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.

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

[13] M. Kapritsos and F. P. Junqueira, "Scalable agreement: Toward ordering as a service," in *HotDep*, 2010.

[14] M. Biely, Z. Milosevic, N. Santos, and A. Schiper, "S-paxos: Offloading the leader for high throughput state machine replication," in *SRDS*, 2012.

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

[16] P. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *DSN*, 2011.

[17] A. de Sousa, R. C. Oliveira, F. Moura, and F. Pedone, "Partial replication in the database state machine," in *NCA*, 2001.

[18] P. Chundi, D. Rosenkrantz, and S. Ravi, "Deferred updates and data placement in distributed databases," in *IDCE*, 1996.

[19] T. Kobus, M. Kokocinski, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *ICDCS*, 2013.

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

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