Strong Consistency at Scale

Carlos Eduardo Bezerra
University of Lugano (USI)
Switzerland

Le Long Hoang
University of Lugano (USI)
Switzerland

Fernando Pedone
University of Lugano (USI)
Switzerland

Abstract

Today's online services must meet strict availability and performance requirements. State machine replication, one of the most fundamental approaches to increasing the availability of services without sacrificing strong consistency, provides configurable availability but limited performance scalability. Scalable State Machine Replication (S-SMR) achieves scalable performance by partitioning the service state and coordinating the ordering and execution of commands. While S-SMR scales the performance of single-partition commands with the number of deployed partitions, replica coordination needed by multi-partition commands introduces an overhead in the execution of multi-partition commands. In the paper, we review Scalable State Machine Replication and quantify the overhead due to replica coordination in different scenarios. In brief, we show that performance overhead is affected by the number of partitions involved in multi-partition commands and data locality.

1 Introduction

In order to meet strict availability and performance requirements, today's online services must be replicated. Managing replication without giving up strong consistency, however, is a daunting task. State machine replication, one of the most fundamental approaches to replication [1, 2], achieves strong consistency (i.e., linearizability) by regulating 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.

State machine replication yields configurable availability but limited scalability. Since every replica added to the system must execute all requests, increasing the number of replicas results in bounded improvements in performance. Scalable performance can be achieved with state partitioning (also known as *sharding*). The idea is to divide the state of a service in multiple partitions so that most commands access one partition only and are equally distributed among partitions. Unfortunately, most services cannot be "perfectly partitioned", that is, the service state cannot be divided in a way that commands access one partition only. As a consequence, partitioned systems must cope with multi-partition commands. Scalable State Machine Replication (S-SMR) [3] divides the service state and replicates each partition. S-SMR relies on an atomic multicast primitive to consistently order commands within and across partitions. Commands that access a single partition are multicast to the concerned partition and executed as in classical SMR; commands that involve multiple partitions are multicast to and executed at all concerned partitions.

Atomic multicast ensures that commands are consistently ordered within and across partitions—in brief, no two replicas deliver the same commands in different orders. However, simply ordering commands consistently is not enough to ensure strong consistency in Scalable State Machine Replication since the execution of commands can interleave in ways that violate strong consistency. In order to avoid consistency violations, S-SMR implements execution atomicity. With execution atomicity, replicas coordinate the execution of multi-partition commands. Although the execution of any two replicas in the same partition does not need coordination, a replica in one partition must coordinate its execution with a replica in every other partition involved in a multi-partition command.

Execution atomicity captures real-time dependencies between commands, typical of strong consistency criteria such as linearizability and strict serializability. In both linearizability and strict serializability, if one operation precedes another in real time (i.e., the first operation finishes before the second operation starts), then this dependency must be reflected in the way the two operations are ordered and executed. Respecting real-time dependencies leads to replicated systems that truly behave like single-copy systems, and thus are easier to program. Serializability does not need execution atomicity but may lead to non-intuitive behavior. We show in the paper that execution atomicity is not as expensive to implement in a partitioned system as one might expect.

S-SMR has proved to provide scalable performance, in some cases involving single-partition commands, with improvements in throughput that grow linearly with the number of partitions [3]. Forcing replicas in different partitions to coordinate to ensure strong consistency may slow down the execution of multi-partition commands, since each replica cannot execute at its own pace (i.e., a fast replica in a partition may need to wait for a slower replica in a different partition). This paper reviews the Scalable State Machine Replication approach and takes a close look at replica coordination in the execution of multi-partition commands. For multi-partition commands that require replicas in different partitions to exchange data as part of the execution of the command, execution atomicity does not affect performance. For multi-partition commands that do not require data exchange, execution atomicity has an impact on performance that depends on the number of partitions involved in the command and on data locality. For example, in workloads that do not experience locality, the overhead in throughput introduced by execution atomicity in commands that span two partitions is around 32% in executions with 16 partitions; in workloads with data locality, this overhead is around 27%.

The remainder of the paper is structured as follows. Section 2 presents the system model and definitions. Sections 3 and 4 recall classical state machine replication and scalable state machine replication. Section 5 describes our experimental evaluation. Section 6 reviews related work and Section 7 concludes the paper.

2 System model and definitions

2.1 Processes and communication

We consider a distributed system consisting of an unbounded set of client processes $C = \{c_1, c_2, ...\}$ and a bounded set of server processes (replicas) $\mathcal{S} = \{s_1, ..., s_n\}$. Set $\mathcal{S}$ is divided into 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). Each server group S_i contains at least $f + 1$ correct processes, where f is the number of faulty processes.

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 and atomic multicast,[1] defined in Sections 2.2 and 2.3, respectively.

[1]Solving atomic multicast requires additional assumptions [4, 5]. In the following, we simply assume the existence of an atomic multicast oracle.

2.2 Reliable multicast

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 reliable-multicasts m, then every correct process in γ reliable-delivers m *(validity).*

– If a correct process reliable-delivers m, then every correct process in γ reliable-delivers m *(agreement).*

– For any message m, every process p in γ reliable-delivers m at most once, and only if some process has reliable-multicast m to γ previously *(integrity).*

2.3 Atomic multicast

To atomically multicast a message m to a set of groups γ, processes use primitive atomic-multicast(γ, m). Message m is delivered at the destinations with atomic-deliver(m). Atomic multicast ensures the following properties:

– If a correct process atomic-multicasts m, then every correct process in γ atomic-delivers m *(validity).*

– If a process atomic-delivers m, then every correct process in γ atomic-delivers m *(uniform agreement).*

– For any message m, every process p in γ atomic-delivers m at most once, and only if some process has atomic-multicast m to γ previously *(integrity).*

– No two processes p and q in both γ and γ' atomic-deliver m and m' in different orders; also, the delivery order is acyclic *(atomic order).*

Atomic broadcast is a special case of atomic multicast in which there is a single group of processes.

3 State machine replication

State machine replication is a fundamental approach to implementing a fault-tolerant service by replicating servers and coordinating the execution of client commands against server replicas [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 consider 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 in real-time 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.

4 Scalable State Machine Replication

In this section, we describe an extension to SMR that under certain workloads allows performance to grow proportionally to the number of replicas [3]. We first recall S-SMR and then discuss some of its performance optimizations.

4.1 General idea

S-SMR divides the application state $\mathcal{V}$ (i.e., state variables) into k partitions $\mathcal{P}_1, ..., \mathcal{P}_k$, 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 that may be accessed by C. Note that this assumption does not imply that the client must know all variables accessed by C, nor even the exact set of partitions. If the client cannot determine a priori which partitions will be accessed by C, it must define a superset of these partitions, in the worst case assuming all partitions. For performance, however, clients must strive to provide a close approximation to the command's actually accessed partitions. We assume the existence of an oracle that tells the client which partitions should receive each command.

Upon delivering command C, if server s does not contain all variables read by C, s must communicate with servers in 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.

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.

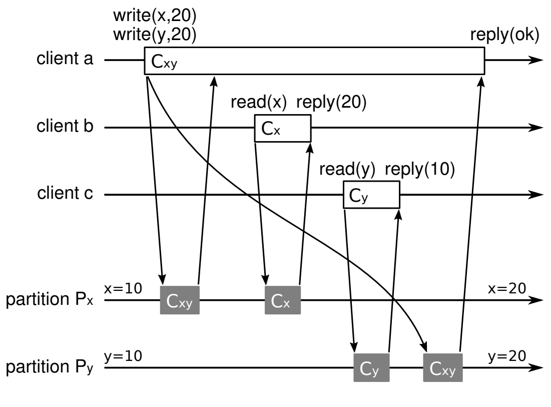
Server s in partition $\mathcal{P}_i$ executes op as follows.

i) $\underline{op \text{ is a } read(v) \text{ operation.}}$
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 $part(v)$.

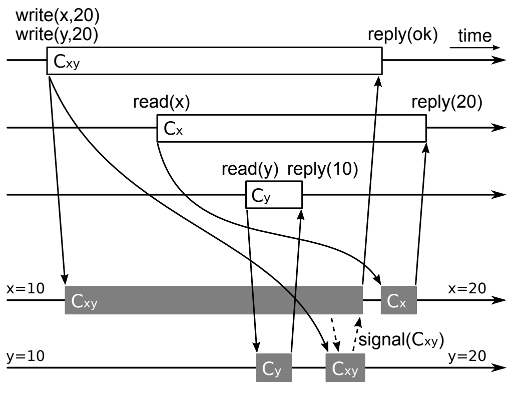
ii) $\underline{op \text{ is a } write(v, val) \text{ operation.}}$
If $\mathcal{P}_i \in part(v)$, s updates the value of v with val; if $\mathcal{P}_i \notin part(v)$, s executes op, creating a local copy of v, which will be up-to-date at least until the end of C's execution.

iii) $\underline{op \text{ is a computation operation.}}$
In this case, s executes op.

It turns out that atomically ordering commands and following the procedure above is not enough to ensure linearizability [3]. 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



(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.)

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 only after r starts executing C. More precisely, let $start(C, p)$ and $end(C, p)$ be, respectively, the time when server p starts executing command C and the time when p finishes C's execution. Execution atomicity ensures that, for every server s in partition $\mathcal{P}$ that executes C, there is a server r in every $\mathcal{P}' \in part(C)$ such that $start(C, r) < end(C, s)$. Intuitively, this condition guarantees that the execution of C at s and r overlap in time.

Replicas can ensure execution atomicity by coordinating the execution of commands. After starting the execution of command C, servers in each partition send a *signal*(C) message to servers in the other partitions in $part(C)$. Before finishing the execution of C and sending a reply to the client that issued C, each server must receive a *signal*(C) message from at least one server in every other partition that executes C. Because of this scheme, each partition is required to have at least $f + 1$ correct servers, where f is the maximum number of tolerated failures per partition; 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 $\mathcal{P}_x$ wait for a signal from $\mathcal{P}_y$, therefore ensuring that the servers of both partitions are synchronized during the execution of C_{xy}. Note that the outcome of each command execution is the same as in case (a), but the execution of C_x, C_y and C_{xy}, as seen by clients, now overlap in time with one another. Hence, there is no real-time precedence among them and linearizability is not violated.

4.2 Performance optimizations

The scheme described in the previous section 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.

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.

5 Performance evaluation

In this section, we present the results found with Chirper, a scalable social network application based on S-SMR. We compare the performance impact of the S-SMR signaling mechanism, by running experiments with, and experiments without signaling turned on. This was done for different workloads and numbers of partitions. In Section 5.1, we describe the implementation of Chirper. In Section 5.2, we describe the environment where we conducted our experiments. In Section 5.3, we report the results.

5.1 Chirper

We implemented Chirper, a social network application similar to Twitter, in order to evaluate the performance of S-SMR. Twitter is an online social networking service in which users can post 140-character messages and read posted messages of other users. The API of Chirper includes: post (user publishes a message), follow (user starts following another user), unfollow (user stops following someone), and getTimeline (user requests messages of all people whom the user follows).

Chirper partitions the state based on user id. A function $f(uid)$ 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 his or her timeline, all messages should be available in the partition that stores that user's data, in the form of a materialized timeline (similarly to a materialized view in a database). To make this possible, whenever a post request 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 request accesses only one partition, follow and unfollow requests access objects on at most two partitions, and post requests access up to all partitions.

One detail about the post request is that it needs access to all users that follow the user issuing the post. Since the Chirper client cannot know for sure who follows the user, it keeps a cache of followers. The client cache can become stale if a different user starts following the poster. To ensure linearizability when executing a post request, the Chirper server checks if the command is sent to the proper set of partitions. If this is the case, the request is executed. Otherwise, the server sends a $retry(\gamma)$ message, where γ is the complete set of additional partitions the command must be multicast to. Upon receiving the $retry(\gamma)$ message, the Chirper client multicasts the command again, now with the destination that includes all partitions in γ. 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 request to all partitions of the system.

Moreover, in order to observe the impact of the signaling mechanism described above, we also introduced these two commands: Follow-noop and Unfollow-noop, which demonstrate pure signal exchange, since they do not change the structure of the social network (i.e., do not change the list of followers and followed users).

5.2 Environment setup and configuration parameters

All experiments were conducted on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve 2920-48G gigabit network switch, and the Dell nodes were connected to another, identical switch. Those switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 7.1 with kernel 3.10 and had the OpenJDK Runtime Environment 8 with the 64-Bit Server

VM (build 25.45-b02). We kept the clocks synchronized using NTP in order to measure latency components involving events in different computers.

For the experiments, we used the following workloads: Timeline (composed only of getTimeline requests), Post (only post requests), and Follow/unfollow (50% of follow-noop requests and 50% of unfollow-noop).

5.3 Results

For the Timeline workload, the throughput and latency when signaling was turned on and off are very similar in both workloads without and with locality (Figures 3 and 2, respectively). This happens because getTimeline requests are optimized to be single-partition: all posts in a user's timeline are stored along with the User object. Every getTimeline request accesses a single User object (of the user whose timeline is being requested). The structure and content of the network do not change, and partitions do not need to exchange either signal or data. Thus, signals do not affect the performance of Chirper when executing getTimeline requests only.

In the Post workload, every command accesses up to all partitions, which requires partitions to exchange both data and signals. In the execution with only one partition, where signals and data are not exchanged anyway, turning off signaling does not change performance. With two partitions or more, signals and data are exchanged across partitions. However, we can see that the signaling does not affect the performance of Chirper significantly. This happens because the Post command changes the content of the User object (changes in timeline), so in both tests, partitions have to exchange data anyhow. For this reason, even when signaling is turned off, the partitions still have to communicate. As a result, the throughput and delay with signaling on or off are similar.

In the Follow/unfollow workload, each command accesses up to two partitions in the system, requires partitions to exchange signal (no data exchanged since the noop command does not change the content of the network). With one partition, Chirper performs in a similar way to that observed with the Post workload in the experiment with one partition. With two partitions or more, performance of Chirper decreased when signaling was turned on. This was expected since the partitions started to exchange the signals: partitions had to wait for signal during the execution of the command.

For both Post and Follow/unfollow workloads, datasets with locality (Figure 2) result in higher throughput and lower latency than datasets without locality (Figure 3).

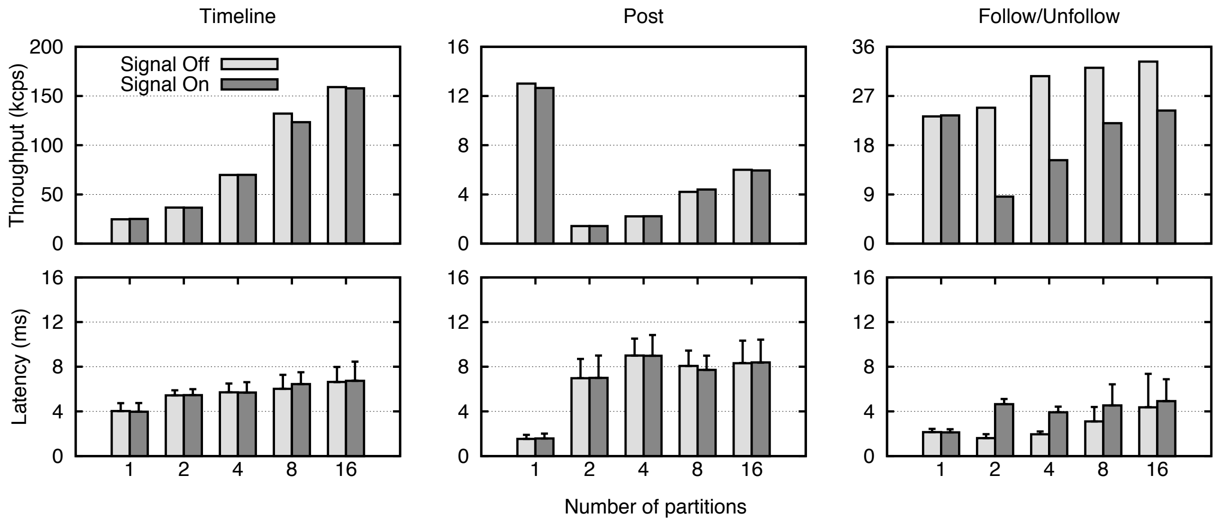


Figure 2: Results of Chirper running with signaling turned on and off, on a dataset with locality. Throughput is shown in thousands of commands per second (kcps). Latency is measured in milliseconds (bars show average and whiskers show 95-th percentile).

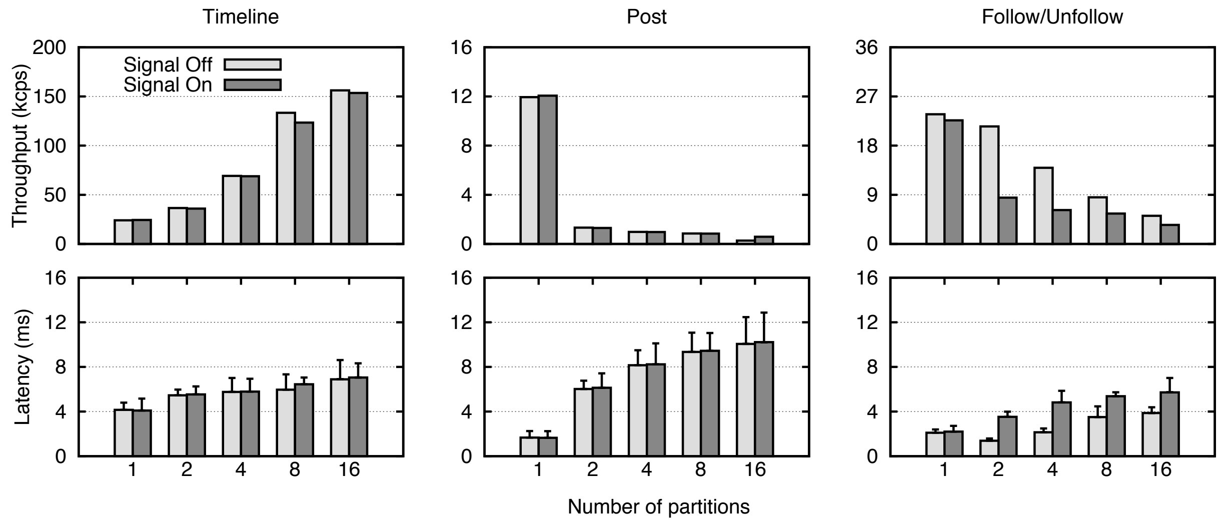


Figure 3: Results of Chirper running with signaling turned on and off, on a balanced dataset. Throughput is shown in thousands of commands per second (kcps). Latency is measured in milliseconds (bars show average and whiskers show 95-th percentile).

6 Related work

State machine replication is a well-known approach to replication and has been extensively studied (e.g., [1, 2, 6, 7, 8]). 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 [9], the authors propose to have clients send their requests to multiple computer clusters, where each such cluster executes the ordering protocol only for the requests it received, and then forwards this partial order to every server replica. The server replicas, then, must deterministically merge all different partial orders received from the ordering clusters. In [10], Paxos [11] 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 [8], 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.

Some works have proposed to parallelize the execution of commands in SMR. In [7], 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 [6], 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 [7] and [6] 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., [12, 13, 14, 15]) 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 [16] or snapshot isolation [17]. 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 [18, 19, 20]. In [19], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In [20], 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 [20] does not allow multi-partition update commands. Spanner [18] 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.

7 Final remarks

This paper described 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. We evaluate S-SMR with Chirper, a scalable social network application. Our experiments demonstrate that throughput scales with the number of partitions, with nearly ideal (i.e., linear) scalability for workloads composed solely of single-partition commands. Moreover, the results show replica coordination, needed to ensure linearizability, has a relatively small cost (in throughput and latency) and this cost decreases with the number of partitions. For multi-partition commands that already require data exchange between partitions, this extra cost is virtually zero.

Acknowledgements

This work was supported in part by the Swiss National Science Foundation under grant number 146404.

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] C. E. Bezerra, F. Pedone, and R. van Renesse, “Scalable state machine replication,” *DSN*, pp. 331–342, 2014.

[4] 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.

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

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

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

[8] 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, pp. 266–275, IEEE Computer Society, 2013.

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

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

[11] L. Lamport, “The part-time parliament,” *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, 1998.

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

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

[14] 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, pp. 1–12, IEEE Computer Society, 2012.

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

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

[17] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño-Martínez, and J. Armendáriz-Iñigo, “Snapshot isolation and integrity constraints in replicated databases,” *ACM Transactions on Database Systems*, vol. 34, no. 2, pp. 11:1–11:49, 2009.

[18] J. Corbett *et al.*, “Spanner: Google’s globally distributed database,” *ACM Transactions on Computer Systems*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

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

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