Università della Svizzera italiana

Faculty of Informatics

USI Technical Report Series in Informatics

Dynamic Scalable State Machine Replication

Long Hoang Le[1], Carlos Eduardo Bezerra[1], Fernando Pedone[1]

[1] Faculty of Informatics, Università della Svizzera italiana, Switzerland

Abstract

State machine replication (SMR) is a well-known technique that guarantees strong consistency (i.e., linearizability) to online services. In SMR, client commands are executed in the same order on all server replicas, and after executing each command, every replica reaches the same state. However, SMR lacks scalability: every replica executes all commands, so adding servers does not increase the maximum throughput. Scalable SMR (S-SMR) addresses this problem by partitioning the service state, allowing commands to execute only in some replicas, providing scalability while still ensuring linearizability. One problem is that S-SMR quickly saturates when executing multi-partition commands, as partitions must communicate. Dynamic S-SMR (DS-SMR) solves this issue by repartitioning the state dynamically, based on the workload. Variables that are usually accessed together are moved to the same partition, which significantly improves scalability. We evaluate the performance of DS-SMR with a scalable social network application.

Report Info

Published
March 2016

Number
USI-INF-TR-2016-1

Institution
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland

Online Access
www.inf.usi.ch/techreports

1 Introduction

State machine replication (SMR) is a well-established technique to develop highly available services (e.g., [1, 2, 3, 4]). In essence, the idea is that replicas deterministically execute the same sequence of client commands in the same order and in doing so traverse the same sequence of states and produce the same results. State machine replication provides configurable fault tolerance in the sense that the system can be set to tolerate any number of faulty replicas. Unfortunately, increasing the number of replicas will not scale performance since each replica must execute every command.

Conceptually, scalable performance can be achieved with state partitioning (e.g., [5, 6, 7]). Ideally, if the service state can be divided such that commands access one partition only and are equally distributed among partitions, then system throughput (i.e., the number of commands that can be executed per time unit) will increase linearly with the number of partitions. Although promising, exploiting partitioning in SMR is challenging. First, most applications cannot be partitioned in such a way that commands always fall within a single partition. Therefore, a partitioning scheme must cope with multi-partition commands. Second, determining an efficient partitioning of the state is computationally expensive and requires an accurate characterization of the workload.

There are two general solutions to handle multi-partition commands. One solution is to weaken the guarantees of commands that involve multiple partitions (e.g., [5]). In the context of SMR, this would mean that single-partition commands are strongly consistent (i.e., linearizable) but multi-partition commands are not. Another solution is to provide strong consistency guarantees for both single- and multi-partition commands, at the cost of a more complex execution path for commands that involve multiple partitions. Scalable State Machine Replication (S-SMR) [8] is a solution in this category. S-SMR partitions the service state and replicates each partition. It relies on an atomic multicast primitive to consistently order commands within and across partitions. Single-partition commands are multicast to their concerned partition

and executed just like in classical SMR. Multi-partition commands are multicast to all involved partitions; to prevent command interleaves that violate strong consistency, S-SMR implements execution atomicity. With execution atomicity, partitions coordinate during the execution of multi-partition commands. Unsurprisingly, multi-partition commands are more expensive than single-partition commands, and thus, the performance of S-SMR is particularly sensitive to the way the service state is partitioned.

Determining a partitioning of the state that avoids load imbalances and favors single-partition commands normally requires a good understanding about the workload. Even if enough information is available, finding a good partitioning is a complex optimization problem [9, 10]. Moreover, many online applications experience variations in demand. These happen for a number of reasons. In social networks, some users may experience a surge increase in their number of followers (e.g., new “celebrities"); workload demand may shift along the hours of the day and the days of the week; and unexpected (e.g., a video that goes viral) or planned events (e.g., a new company starts trading in the stock exchange) may lead to exceptional periods when requests increase significantly higher than in normal periods. S-SMR assumes a static workload partitioning. Any state reorganization requires system shutdown and manual intervention.

Given these issues, it is crucial that highly available partitioned systems be able to dynamically adapt to the workload. In this paper, we present Dynamic Scalable State Machine Replication (DS-SMR), a technique that allows a partitioned SMR system to reconfigure its data placement on-the-fly. DS-SMR achieves dynamic data reconfiguration without sacrificing scalability or violating the properties of classical SMR. These requirements introduce significant challenges. Since state variables may change location, clients must find the current location of variables. The scalability requirement rules out the use of a centralized oracle that clients can consult to find out the partitions a command must be multicast to. Even if clients can determine the current location of the variables needed to execute a command, by the time the command is delivered at the involved partitions one or more variables may have changed their location. Although the client can retry the command with the new locations, how to guarantee that the command will succeed in the second attempt? In classical SMR, every command invoked by a non-faulty client always succeeds. DS-SMR should provide similar guarantees.

DS-SMR was designed to exploit workload locality. Our scheme benefits from simple manifestations of locality, such as commands that repeatedly access the same state variables, and more complex manifestations, such as structural locality in social network applications, where users with common interests have a higher probability of being interconnected in the social graph. Focusing on locality allows us to adopt a simple but effective approach to state reconfiguration: whenever a command requires data from multiple partitions, the variables involved are moved to a single partition and the command is executed against this partition. To reduce the chances of skewed load among partitions, the destination partition is chosen randomly. Although DS-SMR could use more sophisticated forms of partitioning, formulated as an optimization problem (e.g., [9, 10]), our technique has the advantage that it does not need any prior information about the workload and is not computationally expensive.

To track object locations without compromising scalability, in addition to a centralized oracle that contains accurate information about the location of state variables, each client caches previous consults to the oracle. As a result, the oracle is only contacted the first time a client accesses a variable or after a variable changes its partition. Under the assumption of locality, we expect that most queries to the oracle will be accurately resolved by the client’s cache. To ensure that commands always succeed, despite concurrent relocations, after attempting to execute a command a few times unsuccessfully, DS-SMR retries the command using S-SMR’s execution atomicity and involving all partitions. Doing so increases the cost to execute the command but guarantees that relocations will not interfere with the execution of the command.

We have fully implemented DS-SMR as the Eyrie Java library, and we performed a number of experiments using Chirper, a social network application built with Eyrie. We compared the performance of DS-SMR to S-SMR using different workloads. With a mixed workload that combines various operations issued in a social network application, DS-SMR reached 74 kcps (thousands of commands per second), against less than 33 kcps achieved by S-SMR, improving by a factor of over 2.2. Moreover, DS-SMR’s performance scales with the number of partitions under all workloads.

The paper makes the following contributions: (1) It introduces DS-SMR and discusses some performance optimizations, including the caching technique. (2) It details Eyrie, a Java library to simplify the design of services based on DS-SMR. (3) It describes Chirper to demonstrate how Eyrie can be used to implement a scalable social network service. (4) It presents a detailed experimental evaluation of Chirper, deploying it with S-SMR and DS-SMR in order to compare the performance of the two replication techniques.

The rest of the paper is structured as follows. Section 2 describes our system model. Section 3 reviews

SMR and Scalable SMR. Section 4 introduces DS-SMR; we explain the technique in detail and argue about its correctness. Section 5 details the implementation of Eyrie and Chirper. Section 6 reports on the results of our experiments with DS-SMR. Section 7 surveys related work and Section 8 concludes the paper.

2 System model and definitions

We consider a distributed system consisting of an unbounded set of client processes $\mathscr{C} = \{c_1, c_2, ...\}$ and a bounded set of server processes (replicas) $\mathscr{S} = \{s_1, ..., s_n\}$. Set $\mathscr{S}$ is divided into disjoint groups of servers $\mathscr{S}_0, ..., \mathscr{S}_k$. Processes are either *correct*, if they never fail, or *faulty*, otherwise. In either case, processes do not experience arbitrary behavior (i.e., no Byzantine failures).

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

Our consistency criterion is linearizability. A system is *linearizable* if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time precedence of commands [13].

2.1 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)*.

2.2 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). We define delivery order $<$ as follows: $m < m'$ iff there exists a process that delivers m before m'.

Atomic multicast ensures the following properties:

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

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

[1] Solving atomic multicast requires additional assumptions [11, 12]. In the following, we simply assume the existence of an atomic multicast oracle.

3 Background and motivation

State machine replication is a fundamental approach to implementing a fault-tolerant service by replicating servers and coordinating the execution of client commands against server replicas [14, 15]. State machine replication ensures strong consistency (i.e., linearizability [13]) by coordinating the execution of commands in the different replicas: Every replica has a full copy of the service state $\mathcal{V} = \{v_1, ..., v_m\}$ and executes commands submitted by the clients in the same order. A command is a program consisting of a sequence of operations, which can be of three types: $read(v)$, $write(v, val)$, or a deterministic computation.

3.1 Scaling state machine replication

By starting in the same initial state and executing the same sequence of deterministic commands, servers make the same state changes and produce the same reply for each command. To guarantee that servers deliver the same sequence of commands, SMR 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 [16, 17].

Despite its simple execution model, classical SMR does not scale: adding resources (e.g., replicas) will not translate into sustainable improvements in throughput. This happens for a couple reasons. First, the underlying communication protocol needed to ensure ordered message delivery may not scale itself (i.e., a communication bottleneck). Second, every command must be executed sequentially by each replica (i.e., an execution bottleneck).

Several approaches have been proposed to address SMR's scalability limitations. To cope with communication overhead, some proposals have suggested to spread the load of ordering commands among multiple processes (e.g., [18, 19, 20]), as opposed to dedicating a single process to determine the order of commands (e.g., [21]).

Two directions of research have been suggested to overcome execution bottlenecks. One approach (scaling up) is to take advantage of multiple cores to execute commands concurrently without sacrificing consistency [22, 23, 24, 25]. Another approach (scaling out) is to partition the service's state and replicate each partition (e.g., [26, 27]). In the following section, we review Scalable State Machine Replication (S-SMR), a proposal in the second category.

3.2 Scalable State Machine Replication

In S-SMR [8], the service state $\mathcal{V}$ is composed of k partitions, in set $\Psi = \{\mathcal{P}_1, ..., \mathcal{P}_k\}$. Server group $\mathcal{S}_i$ is assigned to partition $\mathcal{P}_i$. For brevity, we say that server s belongs to $\mathcal{P}_i$ meaning that $s \in \mathcal{S}_i$, and say "multicast to $\mathcal{P}_i$" meaning "multicast to server group $\mathcal{S}_i$". S-SMR relies on an *oracle*, which tells which partitions are accessed by each command.[2]

To execute a command, the client multicasts the command to the appropriate partitions, as determined by the oracle. Commands that access a single partition are executed as in classical SMR: replicas of the concerned partition agree on the execution order and each replica executes the command independently. In the case of a multi-partition command, replicas of the involved partitions deliver the command and then may need to exchange state in order to execute the command since some partitions may not have all the values read in the command. This mechanism allows commands to execute seamlessly despite the partitioned state.

Algorithm 1 shows precisely how S-SMR operates. When a server s of partition $\mathcal{P}$, while executing a command C, reaches a $read(v)$ operation, there are two possibilities: either v belongs to the local partition $\mathcal{P}$, or it is part of a remote partition $\mathcal{P}'$. If v is local, s will retrieve its value and send it to the servers of other partitions concerned by C; if v is remote, s will wait until its value is received from a server of $\mathcal{P}'$. A $write(v, val)$ operation does not depend on the previous value of v, not requiring communication between partitions, even if v is not assigned to the partition of the server executing C. To ensure linearizability, all partitions involved in the execution of a multi-partition command C must coordinate before a reply can be sent to the client. In S-SMR, partitions exchange signals while executing multi-partition commands [8]. This guarantees linearizability, at the cost of synchronizing partitions.

[2] The oracle returns a set with the partitions accessed by the command, but this set does not need to be minimal; it may contain all partitions in the worst case, when the partitions accessed by the command cannot be determined before the command is executed.

Algorithm 1 Scalable State Machine Replication (S-SMR)

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

Algorithm variables:

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

$C.id$: unique identifier of command C

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

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

$others$: set of all partitions, other than $\mathcal{P}$, where C is executed.

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

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

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

S-SMR improves on classical SMR by allowing replicated systems to scale, while ensuring linearizability. Under workloads with multi-partition commands, however, it has limited performance, in terms of latency and throughput scalability. Such decreased performance when executing multi-partition commands is due to partitions (i) exchanging state variables and (ii) synchronizing by exchanging signals. S-SMR performs better as the number of multi-partition commands decreases.

One way to reduce the number of multi-partition commands is by dynamically changing the partitioning, putting variables that are usually accessed together in the same partition. However, the partitioning oracle of S-SMR relies on a static mapping of variables to partitions. One advantage of this implementation is that all clients and servers can have their own local oracle, which always returns a correct set of partitions for every query. Such a static mapping has the major limitation of not allowing the service to dynamically adapt to different access patterns.

4 Dynamic Scalable State Machine Replication

In this section, we introduce Dynamic S-SMR (DS-SMR), discuss performance optimizations, and argue about its correctness.

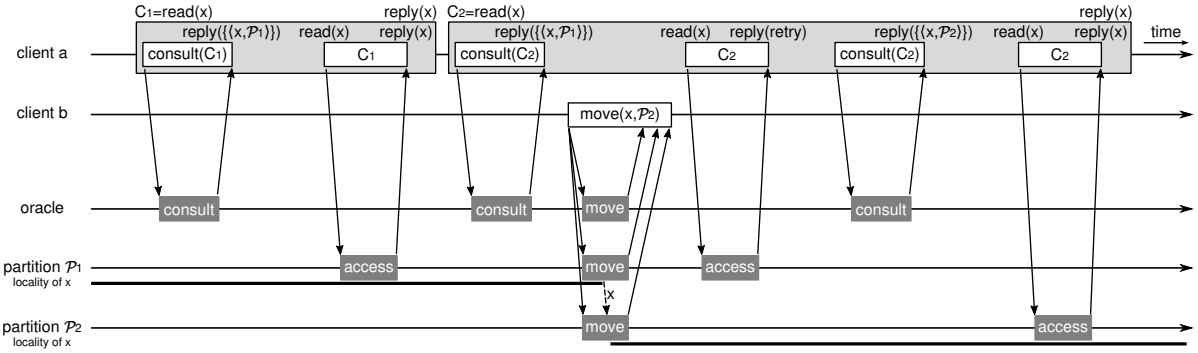

Figure 1: Consulting the oracle and issuing a command are done in multiple calls to atomic-multicast. White boxes represent actions of the client proxy.

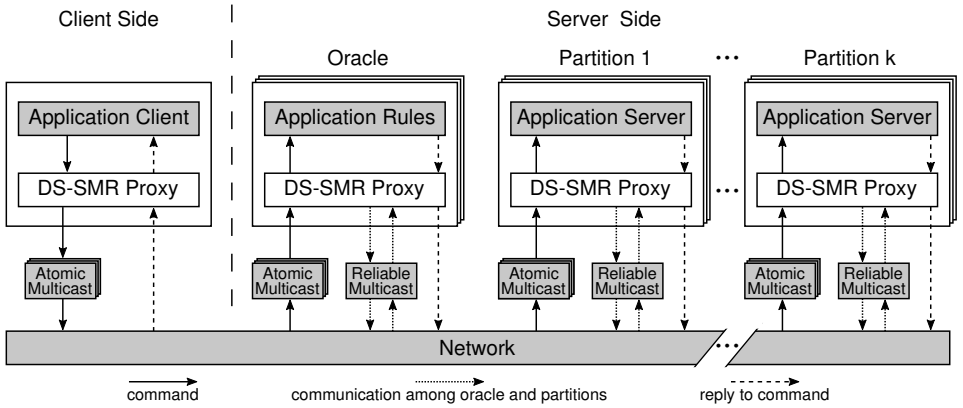

Figure 2: The architecture of Dynamic Scalable State Machine Replication.

4.1 General idea

Dynamic S-SMR (DS-SMR) defines a dynamic mapping of variables to partitions. Each variable v is mapped to partition $\mathscr{P}$, meaning that $v \in \mathscr{P}$. Such a mapping is managed by a partitioning oracle, which is implemented as a replicated service run by group of server processes $\mathscr{S}_0$. The oracle service allows the mapping of variables to partitions to be retrieved or changed during execution. In more detail, DS-SMR distinguishes five types of commands: $access(\omega)$ is an application command that accesses (reads or writes) variables in set $\omega \subseteq \mathscr{V}$ (as described in Section 3), $create(v)$ creates a new variable v and initially maps it to a partition defined by the oracle, $delete(v)$ removes v from the service state, $move(v, \mathscr{P}_s, \mathscr{P}_d)$ moves variable v from partition $\mathscr{P}_s$ to partition $\mathscr{P}_d$, and $consult(C)$ asks the oracle which variables are accessed by command C, and which partition contains each of them. The reply from the oracle to a $consult$ command is called a $prophecy$. A prophecy usually consists of a set of tuples $\langle v, \mathscr{P} \rangle$, meaning that variable v is mapped to partition $\mathscr{P}$. The other possible values for a prophecy are ok and nok, which mean that command can and cannot be executed, respectively (more details in Section 4.2).

Clients can consult the oracle to know which partitions each command should be multicast to, based on which variables are accessed by the command. If the reply received from the oracle tells the client that the command accesses a single partition, the client multicasts the command to that partition. If the command accesses variables from multiple partitions, the client first multicasts one or more $move$ commands to the oracle and to the involved partitions, with the intent of having all variables in the same partition. Then, the command itself is multicast to the one partition that now holds all variables accessed by the command. If a subsequent command accesses the same variables, it will also access a single partition. With this scheme, the access patterns of commands will shape the mapping of variables to partitions, reducing the number of multi-partition commands.

Consulting the oracle and issuing the application command are done with separate calls to atomic multicast in DS-SMR. It may happen that, between those operations, the partitioning changes. We illustrate this in Figure 1. Commands C_1 and C_2 read variable x. Since partitioning is dynamic, the client issuing

the commands first consults the oracle before multicasting each command. C_1 executes without the interference of other commands, so consulting the oracle and multicasting the command only once is enough for C_1 to be executed. However, before C_2 is multicast to $\mathscr{P}_1$, another client issues a $move$ command that relocates x to $\mathscr{P}_2$. When C_2 is delivered at the servers of $\mathscr{P}_1$, the command is not executed, since x is not available at $\mathscr{P}_1$ anymore. A similar situation may arise when a command accesses variables from multiple partitions, as it consists of multicasting at least three commands separately: $consult$, $move$ and $access$. The partitioning can change between the execution of any two of those commands.

To solve this problem, the client multicasts the set of variables accessed along with each access command. Upon delivery, each server checks the set of variables sent by the client. If all variables in the set belong to the local partition, the command is executed; otherwise, a $retry$ message is sent back to the client. When the client receives a $retry$ message, it consults the oracle again, possibly moving variables across partitions, and then reissues the access command. To guarantee termination, if the command fails a certain number of times, the client multicasts the command to all partitions and the servers execute it as in the original S-SMR.

The DS-SMR client consists of the application logic and a client proxy. The application does not see the state variables divided into partitions. When the application issues a command, it sends the command to the proxy and eventually receives a reply. All commands that deal with partitioning (i.e., consulting the oracle, moving objects across partitions and retrying commands as described in the previous paragraph) are executed by the client proxy, transparently to the application. When the client proxy multicasts a partitioning-related command to multiple partitions and the oracle, partitions and oracle exchange signals to ensure linearizability, as mentioned in Section 3.2. Every server and oracle process has its own DS-SMR proxy as well. At each server, the proxy checks whether commands can be executed and manages the exchange of data and signals between processes. At the oracle, the service designer defines the application-dependent rules that must be followed (e.g., where each variable is created at first) and a proxy is responsible for managing the communication of the oracle with both clients and servers when executing commands. DS-SMR relies on a fault-tolerant multicast layer for disseminating commands across replicas and implementing reliable communication between partitions. Replies to commands are sent directly through the network. Figure 2 illustrates the architecture of DS-SMR.

4.2 Detailed algorithm

When issuing a command, the application simply forwards the command to the client proxy and waits for the reply. Consulting the oracle and multicasting the command to different partitions is done internally by the proxy at the client. Algorithms 2, 3, and 4 describe in detail how the DS-SMR proxy works respectively at client, server and oracle processes. Every server proxy at a server in $\mathscr{S}_i$ has only partial knowledge of the partitioning: it knows only which variables belong to $\mathscr{P}_i$. The oracle proxy has knowledge of every $\mathscr{P} \in \Psi$. To maintain such a global knowledge, the oracle must atomic-deliver every command that creates, moves, or deletes variables. (In Section 4.3, we introduce a caching mechanism to prevent the oracle from becoming a performance bottleneck.)

The client proxy. To execute a command C, the proxy first consults the oracle. The oracle knows all state variables and which partition contains each of them. Because of this, the oracle may already tell the client whether the command can be executed or not. Such is the case of the $access(\omega)$ command: if there is a variable $v \in \omega$ that the command tries to read or write and v does not exist, the oracle already tells the client that the command cannot be executed, by sending nok as the prophecy. A nok prophecy is also returned for a $create(v)$ command when v already exists. For a $delete(v)$ command when v already does not exist, an ok prophecy is returned. If the command can be executed, the client proxy receives a prophecy containing a pair $\langle v, \mathscr{P} \rangle$, for every variable v created, accessed or deleted by the command. If the prophecy regarding an $access(\omega)$ command contains multiple partitions, the client proxy chooses one of them, $\mathscr{P}_d$, and tries to move all variables in ω to $\mathscr{P}_d$. Then, the command C itself is multicast to $\mathscr{P}_d$. As discussed in Section 4.1, there is no guarantee that an interleave of commands will not happen, even if the client waits for the replies to the move commands. For this reason, and to save time, the client proxy multicasts all move commands at once. Commands that change the partitioning (i.e., create and delete) are also multicast to the oracle. If the reply received to the command is $retry$, the procedure restarts: the proxy consults the oracle again, possibly moves variables across partitions, and multicasts C to the appropriate partitions once more. After reaching a given threshold of retries for C, the proxy falls back to S-SMR, multicasting C to all partitions (and the oracle, in case C is a create or delete command), which ensures the command's

Algorithm 2 DS-SMR Client Proxy

```
1: To issue a command C, the client proxy does:
2:   do
3:     atomic-multicast(oracle, consult(C))
4:     wait for prophecy
5:     if prophecy ∈ {ok, nok} then
6:       reply ← prophecy
7:     else
8:       C.dests ← {P : ∃⟨v, P⟩ ∈ prophecy}
9:       if C is an access(ω) command and |C.dests| > 1 then
10:        let P_d be one of the partitions in C.dests
11:        for each v ∈ ω do
12:          // move v to partition P_d
13:          let P_s be P : ⟨v, P⟩ ∈ prophecy
14:          if P_s ≠ P_d then
15:            C_move ← move(v, P_s, P_d)
16:            C_move.dests ← {oracle, P_s, P_d}
17:            atomic-multicast(C_move.dests, C_move)
18:        C.dests ← {P_d}
19:      if C is create or delete then
20:        C.dests ← dests ∪ {oracle}
21:      atomic-multicast(C.dests, C)
22:      wait for reply
23:  while reply = retry // after many retries, fall back to S-SMR
24:  return reply to the application client
```

termination.

The server proxy. Upon delivery, access commands are intercepted by the DS-SMR proxy before they are executed by the application server. In DS-SMR, every access command is executed in a single partition. If a server proxy in partition $\mathscr{P}$ intercepts an $access(\omega)$ command that accesses a variable $v \in \omega$ that does not belong to $\mathscr{P}$, it means that the variable is in some other partition, or it does not exist. Either way, the client should retry with a different set of partitions, if the variable does exist. To execute a $delete(v)$ command, the server proxy at partition $\mathscr{P}$ simply removes v from partition $\mathscr{P}$, in case $v \in \mathscr{P}$. In case $v \notin \mathscr{P}$, it might be that the variable exists but belongs to some other partition $\mathscr{P}'$. Since only the oracle and the servers at $\mathscr{P}'$ have this knowledge, it is the oracle who replies to delete commands.

DS-SMR server and oracle proxies coordinate to execute commands that create or move variables. Such coordination is done by means of reliable-multicast. When a $create(v)$ command is delivered at $\mathscr{P}$, the server proxy waits for a message from the oracle, telling whether the variable can be created or not, to be reliable-delivered. Such a message from the oracle is necessary because v might not belong to $\mathscr{P}$, but it might belong to some other partition $\mathscr{P}'$ that servers of $\mathscr{P}$ have no knowledge of. If the create command can be executed, the oracle can already reply to the client with a positive acknowledgement, saving time. This can be done because atomic multicast guarantees that all non-faulty servers at $\mathscr{P}$ will eventually deliver and execute the command. As for move commands, each $move(v, \mathscr{P}_s, \mathscr{P}_d)$ command consists of moving variable v from a source partition $\mathscr{P}_s$ to a destination partition $\mathscr{P}_d$. If the server's partition $\mathscr{P}$ is the source partition (i.e., $\mathscr{P} = \mathscr{P}_s$), the server proxy checks whether v belongs to $\mathscr{P}$. If $v \in \mathscr{P}$, the proxy reliable-multicasts $\langle v, C\rangle$ to $\mathscr{P}_d$, so that servers at the destination partition know the most recent value of v; C is sent along with v to inform which move command that message is related to. If $v \notin \mathscr{P}$, a $\langle null, C\rangle$ message is reliable-multicast to $\mathscr{P}_d$, informing $\mathscr{P}_d$ that the move command cannot be executed.

The oracle proxy. One of the purposes of the oracle proxy is to make prophecies regarding the location of state variables. Such prophecies are used by client proxies to multicast commands to the right partitions. A prophecy regarding an $access(\omega)$ command contains, for each $v \in \omega$, a pair $\langle v, \mathscr{P}\rangle$, meaning that $v \in \mathscr{P}$. If any of the variables in ω does not exist, the prophecy already tells the client that the command cannot be executed (with a nok value). For a $create(v)$ command, the prophecy tells where v should be created, based on rules defined by the application, if v does not exist. If v already exists, the prophecy will contain nok, so that the client knows that the create command cannot be executed. The prophecy regarding a $delete(v)$ command contains the partition that contains v, or ok, in case v was already deleted or

Algorithm 3 DS-SMR Server Proxy

```
1:  To execute a command C, the server proxy in partition 𝒫 does:
2:    when reliable-deliver(⟨val, C⟩)
3:      rcvd_msgs ← rcvd_msgs ∪ {⟨val, C⟩}
4:    when atomic-deliver(C)
5:      if C is an access(ω) command then
6:        if ∃v ∈ ω : v ∉ 𝒫 then
7:          reply with retry
8:        else
9:          have the command executed by the application server
10:         send the reply to the client
11:     else if C is a move(v, 𝒫_s, 𝒫_d) command then
12:       if 𝒫 = 𝒫_s then
13:         if v ∈ 𝒫 then
14:           reliable-multicast(𝒫_d, ⟨v, C⟩)
15:           𝒫 ← 𝒫 \ {v}
16:         else
17:           reliable-multicast(𝒫_d, ⟨null, C⟩)
18:       else
19:         wait until ∃val : ⟨val, C⟩ ∈ rcvd_msgs
20:         if val ≠ null then
21:           v ← val
22:           𝒫 ← 𝒫 ∪ {v}
23:     else if C is a create(v) command then
24:       wait until ⟨val, C⟩ ∈ rcvd_msgs
25:       if val = ok then
26:         𝒫 ← 𝒫 ∪ {v}
27:     else if C is a delete(v) command then
28:       if v ∈ 𝒫 then
29:         𝒫 ← 𝒫 \ {v}
```

never existed.

Besides dispensing prophecies, the oracle is responsible for executing create, move, and delete commands, coordinating with server proxies when necessary, and replying directly to clients in some cases. For each $move(v, \mathscr{P}_s, \mathscr{P}_d)$ command, the oracle checks whether v in fact belongs to the source partition $\mathscr{P}_s$. If that is the case, the command is executed, moving v to $\mathscr{P}_d$. Each $create(v)$ command is multicast to the oracle and to a partition $\mathscr{P}$. If v already exists, the oracle tells $\mathscr{P}$ that the command cannot be executed, by reliable-multicasting nok to $\mathscr{P}$. The oracle also sends nok to the client as reply, meaning that v already exists. If v does not exist, the oracle tells $\mathscr{P}$ that the command can be executed, by reliable-multicasting ok to $\mathscr{P}$. It also tells the client that the command succeeded with an ok reply. Finally, each $delete(v)$ command is multicast to the oracle and to a partition $\mathscr{P}$, where the client proxy assumed v to be located. If v belongs to $\mathscr{P}$, or v does not exist, the oracle tells the client that the delete command succeeded. Otherwise, that is, if v exists, but $delete(v)$ was multicast to the wrong partition, the oracle tells the client to retry.

4.3 Performance optimizations

In this section, we introduce two optimizations for DS-SMR: caching and load balancing.

Caching. In Algorithm 2, for every command issued by the client, the proxy consults the oracle. If every command passes by the oracle, the system is unlikely to scale, as the oracle is prone to becoming a bottleneck. To provide a scalable solution, each client proxy has a local cache of the partitioning information. Before multicasting an application command C to be executed, the client proxy checks whether the cache has information about every variable concerned by C. If the cache does have such a knowledge, the oracle is not consulted and the information contained in the cache is used instead. If the reply to C is $retry$, the oracle is consulted and the returned prophecy is used to update the client proxy's cache. Algorithm 2 is followed from the second attempt to execute C on. The cache is a local service that follows an algorithm similar to that of the oracle, except it responds only to $consult(C)$ commands and, in situations where the

Algorithm 4 DS-SMR Oracle Proxy

```
1:  To execute a command C, the oracle proxy does:
2:    when atomic-deliver(C)
3:      if C is a consult(C_c) command then
4:        prophecy ← ∅
5:        if C_c is an access(ω) command then
6:          if ∃v ∈ ω : (∄P ∈ Ψ : v ∈ P) then
7:            prophecy ← nok
8:          else
9:            for each v ∈ ω do
10:             prophecy ← prophecy ∪ {⟨v, P⟩} : v ∈ P
11:       else if C_c is a create(v) command then
12:         if ∃P ∈ Ψ : v ∈ P then
13:           prophecy ← nok
14:         else
15:           P ← initial partition, defined by application rules
16:           prophecy ← {⟨v, P⟩}
17:       else if C_c is a delete(v) command then
18:         if ∄P ∈ Ψ : v ∈ P then
19:           prophecy ← ok
20:         else
21:           prophecy ← {⟨v, P⟩} : v ∈ P
22:       send prophecy to the client
23:     else if C is a move(v, P_s, P_d) command then
24:       if v ∈ P_s then
25:         P_s ← P_s \ {v}
26:         P_d ← P_d ∪ {v}
27:     else if C is a create(v) command then
28:       let P_c be P : {P} = C.dests \ {oracle}
29:       if ∄P ∈ Ψ : v ∈ P then
30:         outcome ← ok
31:       else
32:         outcome ← nok
33:       reliable-multicast(P_c, ⟨outcome, C⟩)
34:       send outcome to the client
35:     else if C is a delete(v) command then
36:       let P_d be P : {P} = C.dests \ {oracle}
37:       if ∄P ∈ Ψ : v ∈ P or v ∈ P_d then
38:         send ok to the client
39:       else
40:         send retry to the client
```

oracle would return ok or nok, the cache tells the client proxy to consult the actual oracle.

Naturally, the cached partitioning information held by the client proxy may be out of date. On the one hand, this may lead a command to be multicast to the wrong set of partitions, which will probably incur in the client proxy having to retry executing the command. For instance, in Figure 3 the client has an out-of-date cache, incurring in a new consultation to the oracle when executing C_3. On the other hand, the client proxy may already have to retry commands, even if the oracle is always consulted first, as shown in Figure 1. If most commands are executed without consulting the oracle, as in the case of C_4 in Figure 3, we avoid turning the oracle into a bottleneck. Moreover, such a cache can be updated ahead of time, not having to wait for an actual application command to be issued to only then consult the oracle. This way, the client proxy can keep a cache of partitioning information of variables that the proxy deems likely to be accessed in the future.

Load balancing. When moving variables, the client proxies may try to distribute them in a way that balances the workload among partitions. This way, the system is more likely to scale throughput with the number of server groups. One way of balancing load is by having roughly the same number of state vari-

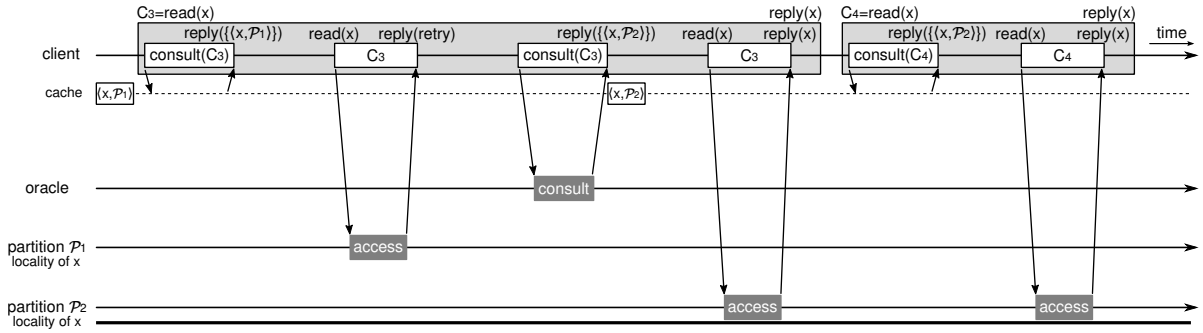

Figure 3: Each client proxy in DS-SMR maintains a cache in order to avoid consulting the oracle. White boxes represent actions of the client proxy.

ables in every partition. This can be implemented by having client proxies choosing randomly the partition that will receive all variables concerned by each command (at line 10 of Algorithm 2). Besides improving performance, balancing the load among partitions prevents the system from degenerating into a single-partition system, with all variables being moved to the same place as commands are executed.

4.4 Correctness

In this section, we argue that DS-SMR ensures termination and linearizability. By ensuring termination, we mean that for every command C issued by a correct client, a reply to C different than $retry$ is eventually received by the client. This assumes that at least one oracle process is correct and that every partition has at least one correct server. Given these constraints, the only thing that could prevent a command from terminating would be an execution that forced the client proxy to keep retrying a command. This problem is trivially solved by falling back to S-SMR after a predefined number of retries: at a certain point, the client proxy multicast the command to all server and oracle processes, which execute the command as in S-SMR, i.e., with coordination among all partitions and the oracle.

As for linearizability, we argue that, if every command in execution $\mathscr{E}$ of DS-SMR is delivered by atomic multicast and is *execution atomic* (as defined in [8]), then $\mathscr{E}$ is linearizable. We denote the order given by atomic multicast by relation $\prec$. Given any two messages m_1 and m_2, "$m_1 \prec m_2$" means that there exists a process that delivers both messages 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 $m_1 \prec m' \prec m_2$. Also, for the purposes of this proof, we consider the oracle to be a partition, as it also atomic-delivers and executes application commands.

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

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

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

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

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

Induction step: If $atomic(i)$, then $atomic(i+1)$.

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

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

5 Implementation

In this section, we describe Eyrie, a library that implements both S-SMR and DS-SMR, and Chirper, a scalable social network application built with Eyrie. Eyrie and Chirper were both implemented in Java.

5.1 Eyrie

To implement a replicated service with Eyrie, the developer (i.e., service designer) must extend three classes: PRObject, StateMachine, OracleStateMachine.

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

The StateMachine class. This class implements the logic of the server proxy. The application server class must extend the StateMachine class. To execute commands, the developer must provide an implementation for the method executeCommand(Command). 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 and oracle transparently. To start the server, method runStateMachine() is called. Method createObject() also needs to be implemented, where the developer defines how new state objects are loaded or created.

The OracleStateMachine class. This class implements the logic of the oracle proxy. It extends StateMachine, so the oracle can be deployed similarly to a fault-tolerant partition in the original S-SMR. Class OracleStateMachine has a default implementation, but the developer is encouraged to override its methods. Method extractObject(Command) returns the set of objects accessed by the command. It should be overridden by the application so that the client proxy can relocate all necessary objects to a destination partition before executing the application command. Method getTargetPartition(Set⟨Object⟩) returns a particular partition to which objects should be moved, when they are not in the same partition yet, in order to execute an application command that accesses those objects. The default implementation of the method returns a random partition. The developer can override it in order to further improve the distribution of objects among partitions. For instance, the destination partition could be chosen based on an attribute of the objects passed to the method.

The client proxy is implemented in class Client, which handles all communication of the application client with the partitioned service. The client proxy provides methods sendCreate(Command, CallbackHandler), sendAccess(Command, CallbackHandler), and sendDelete(Command, CallbackHandler). The client proxy's default behavior is to keep retrying commands (and fallback to S-SMR in case of too many retries) and only call back the application client when the command has been successfully executed. However, the developer can change this behavior by overriding the error() method of CallbackHandler. The error() method is called when a $retry$ reply is received.

5.2 Chirper

We implemented Chirper, a social network application similar to Twitter, using Eyrie. Twitter is an online social networking service in which users can post 140-character messages and read posted messages of other users. The API consists basically of: post (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).

State partitioning in Chirper is based on users' interest. A function $f(uid)$ returns the partition that user with id uid should belong to, based on the user's interest. Function f is implemented in method

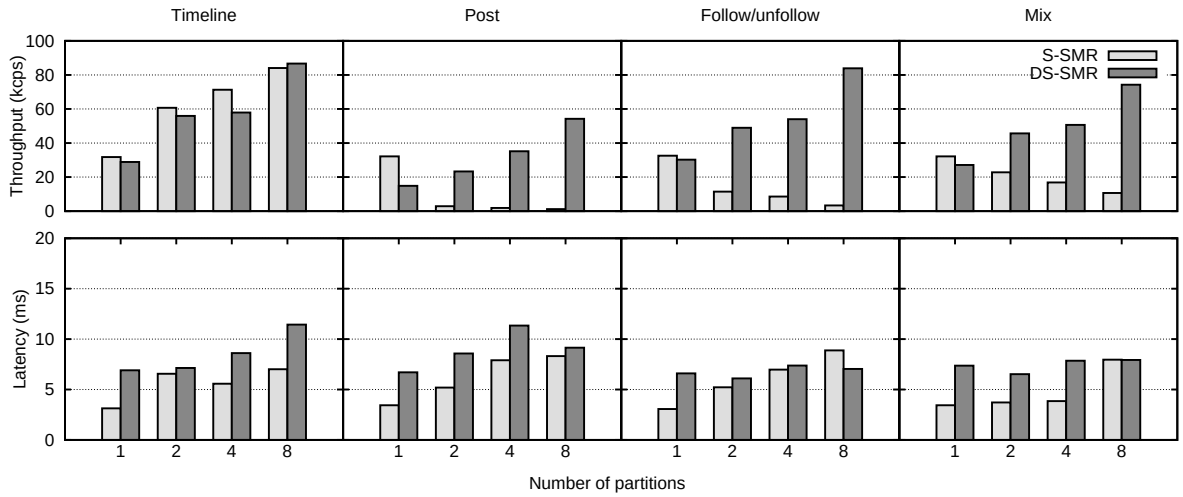

Figure 4: Results of Chirper running with S-SMR and DS-SMR. Throughput is shown in thousands of commands per second (kcps).

getObjectPlacement(User) of class ChirperOracle, which extends OracleStateMachine (class User extends PRObject). 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. The Chirper client does not need any knowledge about partitions, since it uses method sendAccessCommand(command) of the DS-SMR client proxy to issue its commands.

One detail about the post request is that it needs access to all users that follow the user issuing the post. To ensure linearizability when executing a post request, the Chirper server overrides the extractObject(command) method to check if all followers that will be accessed by the command are available in the local partition (i.e., the partition of the server executing the post command). If this is the case, the request is executed. Otherwise, the server sends a $retry(\gamma)$ message, where γ is the complete set of followers of the user who was posting. Then, the Chirper server proceeds to the next command. Upon receiving the $retry(\gamma)$ message, the client proxy tries to move all users in γ to the same partition before retrying to execute the post command.

6 Performance evaluation

In this section, we present the results found for Chirper with different loads and partitionings and compare them with the original S-SMR [8]. In Section 6.1, we describe the environment where we conducted our experiments. In Section 6.2, we show the results.

6.1 Environment setup and configuration parameters

We conducted all experiments on a cluster that had two types of nodes: (a) HP SE1102 nodes, equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b) Dell SC1435 nodes, equipped with two AMD Opteron 2212 processors running at 2.0 GHz and with 4 GB of main memory. The HP nodes were connected to an HP ProCurve 2920-48G gigabit network switch, and the Dell nodes were connected to another, identical switch. Those switches were interconnected by a 20 Gbps

Table 1: Absolute values of Chirper running S-SMR and DS-SMR.

	Timeline				Post				Follow/unfollow				Mix			
	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
	Throughput (commands per second)															
S-SMR	31757	60699	71274	84065	32151	2884	1894	1200	32541	11476	8580	3371	32151	22803	16822	10657
DS-SMR	28882	55925	57900	86685	14874	23295	35188	54250	30215	48976	54025	83880	27101	45686	50671	74257
	Throughput rate = DS-SMR tput / S-SMR tput															
	0.91	**0.92**	**0.81**	**1.03**	**0.46**	**8.08**	**18.48**	**45.00**	**0.93**	**4.27**	**6.30**	**24.88**	**0.84**	**2.00**	**3.01**	**6.97**
	Latency (milliseconds)															
S-SMR	3.1	6.6	5.6	7.0	3.4	5.2	7.9	8.3	3.0	5.2	7.0	8.8	3.4	3.7	3.8	7.9
DS-SMR	6.9	7.1	8.6	11.4	6.7	8.6	11.3	9.1	6.6	6.1	7.4	7.0	7.3	6.5	7.8	7.9

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).

For the experiments, we use the following workloads: Timeline (composed only of getTimeline requests), Post (only post requests), Follow/unfollow (50% of follow requests and 50% of unfollow), and Mix (7.5% post, 3.75% follow, 3.75% unfollow, and 85% getTimeline).

6.2 Results

We can see in Figure 4 and Table 1 the results achieved with Chirper. For the Timeline workload, the throughput with DS-SMR and S-SMR are very similar. 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. This is the ideal workload for S-SMR. In DS-SMR, the partitioning does not change, and consulting the oracle becomes unnecessary thanks to the local cache at each client. This happens because there are no other commands in the Timeline workload.

In the Post workload, every command accesses up to all partitions in the system, which is the worst case for S-SMR: the more partitions are involved in the execution of a command, the worst is the system's performance. We can see that the throughput of S-SMR decreases significantly as the number of partitions increases. For DS-SMR, we can see that the system throughput scales with the number of partitions. This happens because User objects that are accessed together, but which are in different partitions, are moved to the same partition based on the interests of the users. As the execution proceeds, this leads to a lower rate of multi-partition commands, which allows throughput to scale. (In the case of posts on 2 partitions, the number of move commands started at 3 kcps, with throughput of 23 kps, and eventually reduced to less than 0.1 kcps.) As a result the throughput improvement of DS-SMR with respect to S-SMR increases over time. With eight partitions, DS-SMR sports a performance that is 45 times that of S-SMR!

With the Follow/unfollow workload, the system performs in a similar way to that observed with the Post workload. The difference is that each follow or unfollow request accesses only two User objects, whereas every post request may affect an unbounded number of users. For this reason, each follow/unfollow command is executed at most by two partitions in S-SMR. In DS-SMR, a single move command is enough to have all User objects affected by such a command in the same partition. For this reason, both replication techniques have better throughput under the Follow/unfollow workload than with Post. As with the Post workload, DS-SMR's advantage over S-SMR increases with the number of partitions, reaching up to almost 25 times with eight partitions.

We approximate a realistic distribution of commands with the Mix workload. With such a workload, S-SMR does not perform as bad as in the Post or Follow/unfollow workloads, but the system throughput still decreases as partitions are added. As with the other workloads, DS-SMR scaled under the Mix workload. With eight partitions, it reached 74 kcps (thousands of commands per second), fairly close to the ideal case (the Timeline workload), where DS-SMR reached 86 kcps. Under the Mix workload, S-SMR had less than 33 kcps in the best case (one partition) and around 10 kcps with eight partitions. In the configuration with eight partitions, DS-SMR reaches almost seven times S-SMR's throughput.

Latency values with DS-SMR are higher than with S-SMR. This was expected for two reasons. First, there

is an extra group of servers (the oracle) to communicate with. Second, executing a command often means moving all accessed objects to the same partition. Taking this into account, we consider the (often slight) increase in latency observed with DS-SMR a low price to pay for the significant increase in throughput and the scalability that DS-SMR brought to the system; with S-SMR, the system did not scale with multi-partition commands.

7 Related work

State machine replication is a well-known approach to replication (e.g., [14, 15, 22, 28, 29]). It requires replicas to execute commands deterministically, which implies sequential execution. Even though improving the performance of SMR is non-trivial, different techniques have been proposed to achieve scalable systems, such as optimizing the propagation and ordering of commands (i.e., the underlying atomic broadcast algorithm). In [30], 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 [31], Paxos [21] 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.

Other works have proposed multi-threaded implementations of state machine replication, circumventing the non-determinism caused by concurrency in some way. In [29], the authors propose to receive, batch, and dispatch commands in parallel, while commands themselves are still executed sequentially. In [28], a parallelizer module uses application semantics to determine which commands can be executed concurrently without reducing determinism (e.g., read-only commands, which can be executed in any order relative to one another). In [22], commands are speculatively executed in parallel. 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.

Many database replication schemes also aim at improving the system throughput, although commonly they do not ensure linearizability. Many works (e.g., [6, 32, 33, 34]) 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 [35] or snapshot isolation [36]. 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 consistency level is enough, allowing the system to scale better, but services that require linearizability cannot be implemented with such techniques.

Besides S-SMR [8], other works have tried to make linearizable systems scalable [37, 38, 39]. In [38], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In [39], a partitioned variant of SMR is proposed. However, it requires total order (i.e., all commands have to be ordered against each other) and it does not allow update commands to access variables from different partitions. Spanner [37] 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. Dynamic Scalable State Machine Replication employs state partitioning and ensures linearizability for any possible execution, while allowing throughput to scale as partitions are added, even when commands update multiple partitions and without synchronized clocks.

DS-SMR is not to be confused with other dynamic replication schemes though. Systems such as [40, 41, 42] are "dynamic" in the sense that they allow the membership to be reconfigured during execution. For instance, a multicast layer based on Paxos can be reconfigured by adding or removing acceptors. They also allow server replicas to be added and removed during execution. However, this is orthogonal to what DS-SMR proposes. Dynamic Scalable State Machine Replication consists of allowing the *state partitioning*, that is, which state variables belong to which partition, to change dynamically. The greatest challenge that is addressed by DS-SMR is how to provide such a solution, with a dynamic partitioning oracle, while ensuring the strongest level of consistency (linearizability), as variables are created, deleted, and moved across partitions, based on the access patterns of the workload.

Schism [9] proposes an automatic graph-based partitioning of transactional databases that uses the

workload to decide where to place data items. The authors detail how the workload is used to create a graph that captures dependencies between data items, but they do not provide much detail about how the repartitioning can be done dynamically without violating consistency. E-Store [10] is a reconfiguration system for transactional databases that repartitions the data according to the access patterns detected in the workload. It strives to minimize the number of multi-partition accesses and is able to redistribute data items among partitions during execution. However, E-Store assumes that all non-replicated tables form a tree-schema based on foreign key relationships. This has the drawback of ruling out graph-structured schemas and m-n relationships. DS-SMR is a more general approach that works with any kind of relationship between data items, while also ensuring the strongest level of consistency.

8 Conclusion

This work introduces Dynamic Scalable State Machine Replication (DS-SMR), a scalable variant of the well-known state machine replication technique. DS-SMR implements dynamic state partitioning to adapt to different access patterns throughout the execution, while scaling throughput with the number of partitions and ensuring linearizability. To evaluate DS-SMR, we developed the Eyrie library and implemented Chirper, a scalable social network application, with Eyrie. In our experiments, we deployed Chirper with both DS-SMR and S-SMR. The results demonstrate that DS-SMR significantly improves the performance of Chirper over S-SMR in the presence of multi-partition commands.

Acknowledgements

We wish to thank the anonymous reviewers for the constructive feedback. This work was supported in part by the Swiss National Science Foundation under grant number 146404.

References

[1] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP*, 2003.

[3] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.

[4] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou, "Boxwood: Abstractions as the foundation for storage infrastructure," in *OSDI*, 2004.

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

[6] 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.

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

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

[9] C. Curino, E. Jones, Y. Zhang, and S. Madden, "Schism: A workload-driven approach to database replication and partitioning," *Proc. VLDB Endow.*, 2010.

[10] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker, "E-Store: Fine-grained elastic partitioning for distributed transaction processing systems," *Proc. VLDB Endow.*, 2014.

[11] 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.

[12] 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.

[13] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.

[14] 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.

[15] 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.

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

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

[18] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in Egalitarian parliaments," *SOSP*, pp. 358–372, 2013.

[19] Y. Mao, F. P. Junqueira, and K. Marzullo, "Mencius: building efficient replicated state machines for wans," *OSDI*, pp. 369–384, 2008.

[20] P. J. Marandi, M. Primi, and F. Pedone, "Multi-Ring Paxos," *DSN*, pp. 1–12, 2012.

[21] L. Lamport, "The Part-Time Parliament," *ACM Trans. Comput. Syst. ()*, vol. 16, no. 2, pp. 133–169, 1998.

[22] M. Kapritsos, Y. Wang, V. Quéma, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-Verify Replication for Multi-Core Servers," *OSDI*, pp. 237–250, 2012.

[23] P. J. Marandi, C. E. B. Bezerra, and F. Pedone, "Rethinking State-Machine Replication for Parallelism," *ICDCS*, pp. 368–377, 2014.

[24] R. Kotla and M. Dahlin, "High Throughput Byzantine Fault Tolerance," *DSN*, pp. 575–584, 2004.

[25] Z. Guo, C. Hong, M. Yang, D. Zhou, L. Zhou, and L. Zhuang, "Rex: replication at the speed of multi-core," *EuroSys :1-11:14*, p. 11, 2014.

[26] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. E. Anderson, "Scalable consistency in Scatter," *SOSP*, pp. 15–28, 2011.

[27] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," *DSN*, pp. 454–465, 2011.

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

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

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

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] K. Birman, D. Malkhi, and R. van Renesse, "Virtually synchronous methodology for dynamic service replication," Tech. Rep. MSR-TR-2010-151, Microsoft Research, 2010.

[41] Z. Guessoum, J.-P. Briot, O. Marin, A. Hamel, and P. Sens, "Dynamic and adaptive replication for large-scale reliable multi-agent systems," in *Software Engineering for Large-Scale Multi-Agent Systems* (A. Garcia, C. Lucena, F. Zambonelli, A. Omicini, and J. Castro, eds.), vol. 2603 of *Lecture Notes in Computer Science*, pp. 182–198, Springer Berlin Heidelberg, 2003.

[42] S. Dustdar and L. Juszczyk, "Dynamic replication and synchronization of web services for high availability in mobile ad-hoc networks," *Service Oriented Computing and Applications*, vol. 1, no. 1, pp. 19–33, 2007.