Scaling State Machine Replication

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Long Hoang Le

under the supervision of
Fernando Pedone

July 2020

Dissertation Committee

Marc Langheinrich	Università della Svizzera Italiana, Switzerland
Robert Soulé	Università della Svizzera Italiana, Switzerland
Miguel Correia	Universidade de Lisboa, Portugal
Rachid Guerraoui	École Polytechnique Fédérale de Lausanne, Switzerland
PaweÅĆ T. Wojciechowski	Poznań University of Technology, Poland

Dissertation accepted on 23 July 2020

Research Advisor	PhD Program Director
Fernando Pedone	**The PhD program Director *Walter Binder***

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Long Hoang Le
Lugano, 23 July 2020

To Nam Phuong

Research is what I’m doing when I
don’t know what I’m doing.

Wernher von Braun

Abstract

Today's online services must meet strict availability and performance requirements. State machine replication (SMR), one of the most fundamental approaches to increasing the availability of services without sacrificing strong consistency, provides configurable availability but limited performance scalability. The lacking of scalability of SMR due to the fact that every replica has to execute all commands, which limits the overall performance by the throughput of a single replica, so adding servers does not increase the maximum throughput. 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 this thesis, we propose and implement the following ideas: (i) Dynamic scalable state machine replication (DS-SMR), (ii) Dynastar: Optimized partitioning for SMR. DS-SMR addresses the problem of S-SMR by allowing repartitioning the service state dynamically, based on the workload. Variables that are usually accessed together are moved to the same partition, which significantly improves scalability. To provide better partitioning for DS-SMR, we develop DynaStar, a novel approach to scaling SMR. DynaStar also uses dynamically repartitioning state technique, combined with a centralized oracle, to maintain a global view of workload and inform heuristics about data placement. Using this oracle, DynaStar is able to adapt to workload changes over time, while also minimizing the number of state changes. The performance evaluation using two benchmarks, a social network based on real data and TPC-C, shows that DynaStar is a practical technique that achieves excellent throughput.

Acknowledgements

This thesis is the product of an interesting, albeit long and challenging, journey. It would not be possible without the help of many people. I am extremely grateful to all of them.

First and foremost, I would like to express my genuine gratitude to my advisor Prof. Fernando Pedone, for the support of my Ph.D. study and research, and for his inexhaustible patience, enthusiasm, and continuous encouragement. I have learned many valuable lessons from his immense knowledge, from doing research, presenting ideas, writing reports, and even windsurfing. Much of this dissertation is a result of his endless support. I thank you.

I am grateful to the dissertation committee members, Miguel Correia, Robert Soulé, Rachid Guerraoui, and Marc Langheinrich for their insightful comments and the time they spent examining this thesis. In particular, I would like to thank PaweÅĆ T. Wojciechowski for his detailed, constructive feedback and also for the great time he hosted me in PoznaÅĎ.

It has always been a privilege to be a part of the Distributed Systems Lab. I would like to thank the current and former members of the group that I had the opportunity to spend time with: Eduardo, Paulo, Edson, Daniele, Loan, Tu, Mojtaba, Odorico, Pietro, Samuel, Theo, Tu, Nenad, Elia, Tarcisio, Leandro and Fynn. I would never forget those coffee times that we had together.

I am grateful to my big family for encouraging me and give unconditional support throughout my whole life, especially my parents for their patience and love. Lastly, I am and will always be thankful to my little family for standing with me during last five year and be a part of some of the happiest moments in my life.

Preface

The result of this research appears in the following publications:

- Hoang Le, L., Bezerra, C. E. and Pedone, F. [2016]. Dynamic scalable state machine replication, *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 13âĂŞ24.

- Hoang Le, L., Fynn, E., Eslahi-Kelorazi, M., Soulé, R. and Pedone, F. [2019]. Dynastar: Optimized dynamic partitioning for scalable state machine replication, *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 1453âĂŞ1465.

- Bezerra, C. E., Le, L. H. and Pedone, F. [2016]. *Strong consistency at scale., IEEE Data Eng. Bull* **39**(1): 93âĂŞ103.

Contents

Contents **xii**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**
1.1 Context and goal of the thesis 1
1.2 Research contributions 3
1.3 Thesis outline 4

2 System model and definitions **7**
2.1 System model 7
2.1.1 Reliable multicast 8
2.1.2 Atomic multicast 8
2.2 Consensus 9
2.3 Consistency 9
2.4 State machine replication 10
2.5 Scaling state machine replication 11
2.6 Graph partitioning problem 12
2.7 Workload locality 12

3 Scaling partitioned replicated systems **13**
3.1 Full replication schemes 14
3.2 Partial replication schemes 17
3.2.1 Agreement coordination without server coordination . . . 18
3.2.2 Server coordination without agreement coordination . . . 21
3.2.3 Server coordination with agreement coordination 22
3.3 Conclusion 23

4 Dynamic partitioning for SMR **25**
4.1 General idea . . . 27
4.2 Detailed algorithm . . . 30
4.3 Performance optimizations . . . 33
4.4 Correctness . . . 35
4.5 Implementation . . . 37
4.5.1 Eyrie . . . 38
4.5.2 Chirper . . . 39
4.6 Performance evaluation . . . 40
4.6.1 Environment setup and configuration parameters . . . 40
4.6.2 Results for strong locality . . . 40
4.6.3 Results for weak locality . . . 43
4.7 Conclusion . . . 44

5 Optimized partitioning for SMR **47**
5.1 General idea . . . 49
5.2 Detailed Algorithm . . . 50
5.3 Performance optimizations . . . 54
5.4 Correctness . . . 55
5.5 Implementation . . . 56
5.5.1 Atomic multicast . . . 56
5.5.2 Graph partitioning with METIS . . . 57
5.5.3 DynaStar . . . 57
5.5.4 TPC-C benchmark . . . 58
5.5.5 Social network application . . . 59
5.5.6 Alternative system . . . 59
5.6 Performance evaluation . . . 60
5.6.1 Experimental environment . . . 60
5.6.2 Methodology and goals . . . 60
5.6.3 TPC-C benchmark . . . 60
5.6.4 Social network . . . 61
5.6.5 The performance of the oracle . . . 67
5.7 Conclusion . . . 69

6 Related work **71**
6.1 State machine replication . . . 71
6.2 Consensus and state machine replication . . . 71
6.3 Scaling state machine replication . . . 72
6.4 Graph partitioning in performance scaling . . . 74

7 Conclusion **77**
7.1 Research assessment 78
7.2 Future directions 79

Bibliography **83**

Figures

3.1 Functional model with the five phases 15
3.2 Communication steps involved in active replication 16
3.3 Communication steps involved in passive replication 16
3.4 Coordination in a partitioned replicated system 18
3.5 Coordination in single-partition transactions in Spanner 19
3.6 Coordination of multi-partition transactions in Spanner 20
3.7 Transaction processing in Calvin 22
3.8 Atomic multicast and S-SMR 24

4.1 Consulting the oracle and issuing a command are done in multiple calls to atomic multicast. 28
4.2 The architecture of Dynamic Scalable State Machine Replication. 29
4.3 Each client proxy in DS-SMR maintains a cache in order to avoid consulting the oracle. White boxes represent actions of the client proxy. 36
4.4 Number of commands require moving of objects versus throughput of DS-SMR with Post commands, running on strong-locality workload on 4 partitions 41
4.5 Results of Chirper running with S-SMR and DS-SMR with strong-locality workload (throughput in thousands of commands per second, kcps). 42
4.6 Cumulative distribution function (CDF) of latency for mix workloads of DS-SMR and S-SMR. 44
4.7 Results of Chirper running with S-SMR and DS-SMR with weak-locality workload. Throughput is shown in thousands of commands per second (kcps). 45
4.8 Number of commands require moving of objects versus throughput of DS-SMR with Post commands, running on weak-locality workload with 4 partitions. 45

5.1 Execution of a create (C1) and a write without client cache (C2) and with client cache (C3) in DynaStar. 52
5.2 Repartitioning in DynaStar; throughput (top), objects exchanged between partitions (middle), and percentage of multi-partition commands (bottom). 62
5.3 Performance scalability with TPC-C. Throughput (in thousands of transactions per second, ktps) and latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile). 63
5.4 Performance of timeline command of social network service. Throughput (in thousands of commands per second, kcps) and latency for different partitions. Latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile). . . 64
5.5 Performance of post command social network service. Throughput (in thousands of commands per second, kcps) and latency for different partitions. Latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile). . . 65
5.6 Repartitioning a dynamic workload with DynaStar (a) and S-SMR* without repartitioning (b). 66
5.7 Cumulative distribution function (CDF) of latency for mix workloads on different partitioning configurations. 66
5.8 METIS processor and memory usage. 68
5.9 Queries sent to oracle in the social network service 68

Tables

4.1 Absolute values of Chirper running S-SMR and DS-SMR. 42

5.1 Average load at partitions at peak throughput. 67

Chapter 1

Introduction

1.1 Context and goal of the thesis

Modern applications today deploy services across a massive and continuously expanding infrastructure, in order to serve users efficiently and reliably. With the wide availability of commodity hardware, failures in data centers are increasingly common and inevitable, even in well-established service providers [69]. The causes of failures are varied. Machines fail individually due to power failures, hardware failures, or collectively due to human errors [5], software bugs, or even extreme weather events [13]. While confronting such failures, applications still need to offer uninterrupted service to their users. To this end, introducing fault tolerance through redundancy is critical for the availability and integrity of the applications.

Redundancy by replication can increase both availability and scalability. By having several copies of the application running on multiple replicas, the failure of one replica does not result in the interruption of the service. Moreover, requests from users can be distributed across multiple replicas, so the workload can be divided among different machines, which would translate into better scalability. A main difficulty with replicating an application though is managing consistency among replicas. A strong consistency system gives programmers an intuitive model for the effects of concurrent updates (i.e., clients perform concurrent updates as there is a single copy of the application state), making it less likely that complex system behavior will result in bugs.

State machine replication is a well-known form of software replication, in particular of *active replication*, to build fault-tolerant services using commodity hardware (e.g., [70; 29; 15; 52]). In essence, the idea is to model the application as a deterministic state machine whose state transitions consist of the execution

of client requests [47; 66]. Then, the service is fully replicated on several servers that deterministically execute every client command in the same order to reach 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. In terms of scalability, unfortunately, classic state machine replication does not scale. Increasing the number of replicas will not scale performance since each replica must execute every command. Thus, the overall performance is limited by the throughput of a single replica.

Conceptually, scalable performance can be achieved with state partitioning, also known as sharding (e.g., [14; 67; 3]), which partitions the persistent state of the service across multiple servers. Ideally, if the service state (e.g., state variables) 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 that avoids load imbalance and favors single-partition commands normally is computationally expensive and requires an accurate characterization of the workload. Even if enough information is available, finding a good partitioning is a complex optimization problem [23; 73]. Third, 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.

It is crucial that highly available partitioned systems be able to dynamically adapt to the workload in an optimized way (e.g., minimize cross-partition communication). We believe there should be a way to provide strong consistency in a scalable manner, that is, to create a linearizable, scalable and efficient replicated service. Throughout this thesis, we aim to solve the problems mentioned above, while substantiating the following claim:

> "It is possible to devise an approach that allows a fault-tolerant system to scale by dynamically adapting to the changes in the workload, while providing strong consistency, that is, ensuring linearizability."

1.2 Research contributions

The contribution of this thesis is centered on scaling performance of a replicated state machine system. In this section, we outline the main contributions of this work and provide a short description of each one. We defer detailed discussions to the next chapters.

To achieve the goal defined above for the doctoral thesis, we first conducted a survey of different classes of techniques to partial replication. In particular, we extended the replication framework introduced in [78] to account for partial replication. We then proposed **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. This scheme benefits from simple manifestations of locality, such as commands that repeatedly access the same state variables, and more complex cases, 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., [23; 73]), it has the advantage that it does not need any prior information about the workload and is not computationally expensive.

DS-SMR addresses the limitations of static approaches by adapting the partitioning scheme as workloads change, by moving data "on demand" to maximize the number of single partition user commands, while avoiding imbalances in the

load of the partitions. The major challenge in this approach is determining how the system selects the partition to which to move data. DS-SMR selects partitions randomly, which allows for a completely decentralized implementation, in which partitions make only local choices about data movement. We refer to this approach as *decentralized partitioning*. This approach works well for data with *strong locality*, but it is unstable for workloads with *weak locality*.[1] This happens because with weak locality, objects in DS-SMR are constantly being moved back and forth between partitions without converging to a stable configuration.

For this reason, we introduce the main contribution of this thesis, **DynaStar, Optimized Dynamic Partitioning for Scalable State Machine Replication**. Like DS-SMR, DynaStar does not require any a priori knowledge about the workload. However, DynaStar differs from the prior approach because it creates the workload graph on-the-fly and uses graph partitioning techniques to efficiently relocate application state on-demand. In order to reach an optimized partitioning, DynaStar maintains a location oracle with a global view of the application state. The oracle minimizes the number of state relocations by monitoring the workload, and re-computing an optimized partitioning on demand using a static partitioning algorithm. When a client submits a command, it first contacts the oracle to discover the partitions on which state variables are stored. If the command accesses variables in multiple partitions, the oracle issues a move command to the partitions, re-locating variables to a single partition. When re-locating a variable, the oracle is faced with a choice of which partition to use as a destination. DynaStar chooses the partition for relocation (i.e., one that would minimize the number of state relocations) by partitioning the workload graph using the METIS [1] partitioner. To track object location without compromising scalability, in addition to information about the location of state variables in the oracle, 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.

1.3 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 provides the background for the thesis, by describing the system model and the communication primitives used throughout this thesis. Chapter 3 considers the general problem of scaling replicated systems. Chapter 4 discusses DS-SMR, a dynamic partitioning scheme

[1] The definition of *strong*- and *weak-locality* is postponed until Section 2.7.

for state machine replication, explaining how it may achieve dynamic state reconfiguration while still maintaining strong consistency for state machine replication. Chapter 5 presents DynaStar, which combines the dynamic repartitioning state technique and the centralized oracle to maintain a global view of the workload and partition application state. Chapter 6 surveys related work on scaling state machine replication. Finally, Chapter 7 concludes the thesis and proposes future research directions.

Chapter 2

System model and definitions

In this chapter, we present the system model, introduce two variations of a multicast communication primitive, and define our correctness criterion (i.e., linearizability).

2.1 System model

Unless mentioned otherwise, we make the following assumptions about processes, failures and system synchrony.

- **Processes and communication.** 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 communicate by message passing, using either one-to-one or one-to-many communication. 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, defined in §2.1.1 and §2.1.2, respectively.

- **Failure model.** We assume the *crash* failure model. 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).

- **Synchrony model.** Consensus cannot be solved deterministically in an asynchronous system with faults, as established by the FLP impossibility result [28]. So, we consider a system that is *partially synchronous* [27]:

the system is initially asynchronous and eventually becomes synchronous. When the system is asynchronous, there are no bounds on the time it takes for messages to be transmitted and actions to be executed; when the system is synchronous, such bounds exist but are unknown to the processes. The partially synchronous assumption allows consensus defined in §2.2, to be implemented under realistic conditions [28; 48].

2.1.1 Reliable multicast

Our protocol uses a uniform reliable multicast primitive that allows to multicast messages reliably to a subset of the groups in $\mathscr{S}$. For each message m, γ denotes the groups to which the message is reliably multicast. By abuse of notation, we write $p \in \gamma$ instead of $\exists g \in \mathscr{S} : g \in \gamma \wedge p \in g$ To reliably multicast a message m to a set of groups γ in $\mathscr{S}$, processes use primitive r-mcast(γ, m). Message m is delivered at the destinations with r-deliver(m). Reliable multicast has the following properties [34]:

- If a correct process r-mcasts m, then some correct process $p \in \gamma$ eventually r-delivers m or no process in that group is correct*(validity)*.
- If a correct process $p \in \gamma$ r-delivers m, then every correct process $p \in \gamma$ eventually r-delivers m *(uniform agreement)*.
- For any message m, every process $p \in \gamma$ r-delivers m at most once, and only if some process has r-mcast m to γ previously *(uniform integrity)*.

2.1.2 Atomic multicast

Atomic multicast allows messages to be addressed to a subset of the processes in the system. Atomic multicast ensures that the destination processes of every message agree either to deliver or to not deliver the message, and no two processes deliver any two messages in different order. To atomically multicast a message m to a set of groups γ, processes use primitive a-mcast(γ, m). Message m is delivered at the destinations with a-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 a-mcasts m, then some correct process $p \in \gamma$ eventually a-delivers m or no process in that group is correct*(validity)*.

- If a correct process $p \in \gamma$ a-delivers m, then every correct process $p \in \gamma$ eventually a-delivers m *(uniform agreement)*.

- For any message m, every process $p \in \gamma$ a-delivers m at most once, and only if some process has a-mcast m previously *(integrity)*.

- If a process a-mcasts m and then m' to group γ, then no process in γ a-delivers m' before m *(fifo order)*.

- 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.

2.2 Consensus

Consensus is a fundamental coordination problem of distributed computing [47; 64]. The problem is related to replication and appears when implementing atomic broadcast, group membership, or similar services. Given a set of servers proposing values, it is the problem of deciding on one value among the servers. The consensus problem is defined by the primitives *propose(v)* and *decide(v)*, where v is an arbitrary value. Any uniform consensus must satisfy the following three properties:

- If a process decides v, then v was previously proposed by some process *(uniform integrity)*.

- If one or more correct processes propose a value then eventually some value is decided by all correct processes *(termination)*.

- No two processes decide different values *(uniform agreement)*.

2.3 Consistency

An object that can be concurrently accessed by many processes is called a concurrent object [36]. Interleaving accesses to the same object can sometimes lead

to unexpected behaviors. The effect of this issue can be captured by defining a consistency criterion over the shared object, which specifies how operations can interleave in accessing the object. Informally, systems that provide strong consistency are easier to interact with, because they behave in an intuitive way: they behave as the system had only one copy of the object, and all operations modified and read that object atomically. In this thesis, we specifically focus on strong consistency.

Strong consistency commonly refers to the formal concept of either strict serializability or linearizability [36]. A system is strictly serializable if the outcome of any sequence of operations, as observed by its clients, is equivalent to a serialization of those operations in which the temporal ordering of non-overlapping operations is respected (i.e., if an operation o_1 is acknowledged by the system before another operation o_2 is proposed by some clients, then o_1 comes before o_2 in the equivalent serialization). Linearizability is a sub-case of strict serializability in which every operation reads or updates a single object. Linearizability is a "local" and "non-blocking" property. Linearizability is local in that it is sufficient for a system to linearize operations for each individual object to achieve global linearizability. Linearizability is non-blocking in that pending operations do not have to wait for other pending operations to complete.

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.

2.4 State machine replication

State machine replication, also called active replication, is a common approach to building fault-tolerant systems [47; 66]. State machines model deterministic applications. They atomically execute commands issued by clients. This results in a modification of the internal state of the state machine and also in the pro-

duction of an output to a client. An execution of a state machine is completely determined by the sequence of commands it executes and is independent of external inputs such as timeouts. A fault-tolerant state machine can be implemented by replicating it over multiple servers. Commands must be executed by every replica in a consistent order, despite the fact that different replicas might receive them in different orders. 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 [11; 24]. In this thesis, we consider implementations of state machine replication that ensure linearizability.

2.5 Scaling state machine replication

State machine replication yields configurable availability but limited scalability. Adding resources (i.e., replicas) results in a service that tolerates more failures, but does 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., [57; 53; 55]), as opposed to dedicating a single process to determine the order of commands (e.g., [48]).

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 [41; 54; 46; 32]. Ideally, one could use a replication technique that supports parallelism (multi-threading) to scale up a replicated service. But existing techniques have at least one sequential part in their execution. Another approach (scaling out) is to partition the service's state (also known as *sharding*) and replicate each partition (e.g., [30; 56]). 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. In the next chapter, we review some approaches in the second category,

which include Scalable State Machine Replication (S-SMR) [9], Google Spanner [22], and some other techniques.

2.6 Graph partitioning problem

In computer systems, graphs are widely used to describe the dependencies of the data within a computation. All applications can model their data and/or workload using a graph. For example, in a social network application, the workload could be modeled as a weighted graph, the users are represented by nodes and the connections/relations between those users by the edges between nodes. Once the workload is modeled as a graph, graph partitioning can be used to determine how to divide the work and data for efficient concurrent access or computation. A weighted (directed) graph G consists of a set of nodes V and a set of edges $E \subset V \times V$ to represent relations between the nodes, as well as two cost functions. One function assigns weights to the nodes $c : V \rightarrow R_{>0}$ and a second function $\omega : E \rightarrow R$ assigns costs to the edges. A subgraph is a graph whose node and edge set are subsets of another graph. An edge that connects vertices of two different subgraph is called a *crossing edge*. Graph partitioning is a fundamental problem of finding a specified number of distinct subsets of the set of vertices of a graph (subgraph). *Balance* is the constraint that requires that all subgraph have about equal size. *Total edge cut* is the number of edges crossing those subgraphs. Intuitively, a good partitioning is the one that minimizes the number of the total edges connecting subgraphs, while maintaining the balance among partitions. This is generally considered an NP-hard problem [42]

2.7 Workload locality

In order to achieve a good partitioning, it is necessary to exploit the locality of a workload. Locality is one of the long-known and much-exploited principles in computer science. Operating system and databases have been exploiting the locality for years [25]. In the scope of this thesis, we define the locality of a workload as the data access patterns produced by the application. A workload has *strong locality* if it can be partitioned in a way that would both (i) allow commands to be executed at a single partition only, and (ii) evenly distribute data so that load is balanced among partitions. Conversely, workloads that cannot avoid multi-partition commands with balanced load among partitions exhibit *weak locality*.

Chapter 3

Scaling partitioned replicated systems

Replication refers to the technique of managing redundant data on a set of servers (replicas) in a way to ensure that each replica of the service keeps a consistent state, given a consistency criterion. Over more than three decades, replication has been an area of interest and has been studied in many domains, such as database systems, file systems, and distributed object systems. Although replication is often used to achieve high availability, replication can also help improve performance. Availability is the capability of a system to continue to work, even in the presence of failures, as long as the number of failures is below a given threshold. Performance refers to the throughput and response time of the system. For instance, a service that serves mainly read operations can distribute the requests to the different replicas. Each replica can then process requests in parallel and improve the throughput and the response of the system.

Conceptually, replication can be categorized into two main categories: full and partial replication. Full replication means that the whole state of the service is available on all replicas, while in partial replication, each replica only contains a subset of the state. For example, in a distributed database, full replication means that all rows of a table are available on all replicated servers, while partial replication means that some replicas contain only a subset of the rows. Intuitively, full replication often comes at a higher cost. If all replicas have to keep the whole same state and execute the same sequence of commands, the system cannot scale under update operations. It may also become infeasible to have a full copy of the system's state when data continually grows, as the replicas may not have enough space to store it (i.e., typically in main memory databases). Therefore, increasing the number of replicas in a fully replicated system results in bounded improvements in performance. The main idea of partial replication is that not all replicas have to process each request. The replicas in a partially

replicated system only store a subset of the state and handle requests that involve that data, thus providing the potential for scalable performance. However, the main challenge is how to keep the replicas in a partially replicated system consistent. In this chapter, we will survey several approaches to replication and scaling the performance of a replicated system.

3.1 Full replication schemes

Replication has become a standard approach to fault tolerance: if one server fails, another one takes over. In a fully replicated system, every server replicates a full copy of the service's state. All replicas use an algorithm that ensures data coherence across all these nodes. Although there are many replication techniques [18], there are two major classes of replication techniques that have become widely known to ensure strong consistency: *active replication* and *passive replication*.

Active replication

With the *active replication* (also called state machine replication) approach, client requests are sent to all members of a given group of replicas in the same order. The replicas will then execute the requests as though they were the only member of the group, to reach the same state, and reply to the client. If a client sends a request to a group of n replicas (assuming no failures), then the client will receive n identical replies to the request. With this replication scheme, a crash does not increase the latency experienced by the client. State machine replication is a fundamentally powerful building block that enables the implementation of highly available distributed services by replication. Some other replication schemes, such as chain replication [77; 76], multi-primary replication, and deferred-update replication [67; 18; 19] are based on state machine replication.

Passive replication

In the *passive replication* (also called *primary-backup*) approach, the requests are sent to only one member of the replica group (the *primary*), while all other members are *backups*. The primary will execute each request and send the response to the client. Any modifications to the primary's state is updated to other members of the group (the backups). If the primary fails, one of the backups takes over the service by becoming a new primary. Unlike the active replication approach, passive replication does not waste resource on redundant execution of

requests, and allows non-deterministic requests. However, a failure on the leader will affect the latency of a command.

In [78], the authors presented a framework to compare and distinguish replication protocols in databases and distributed systems (Figure 3.1). This model classifies replication protocols by five generic steps of replication as following:

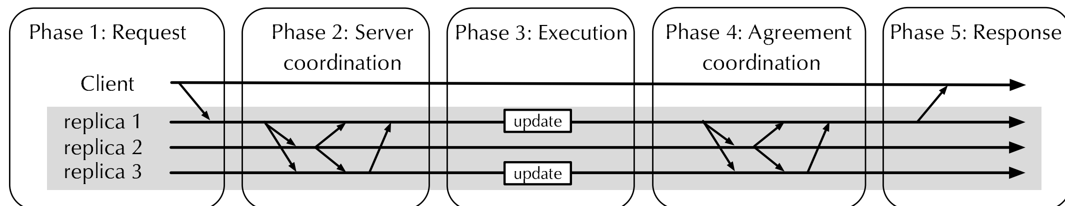

Figure 3.1. Functional model with the five phases

- *Request phase (RE):* In the request phase, a client submits an operation to the system by sending the request to one ore more replicas.
- *Server coordination phase (SC):* The replicas coordinate with each other to synchronize the execution of the request.
- *Execution phase (EX):* The operation is executed on the replica servers.
- *Agreement coordination phase (AC):* The replica servers coordinate to agree on the result of the execution.
- *Response phase (END):* The result of the operation is transmitted back to the client.

The differences between different replicated systems come from the different ways in which each phase is implemented. In some cases, a phase could be omitted (e.g., when messages are ordered by an atomic multicast/broadcast primitive in the ordering phase (*SC*), it is not necessary to run the agreement coordination (*AC*) phase). With this generic functional model, the involved steps in the active replication technique can be depicted as follows (Figure 3.2):

1. The client submits the request to the server by using an atomic broadcast primitive.
2. Server coordination is given by the total order property of the Atomic Broadcast.

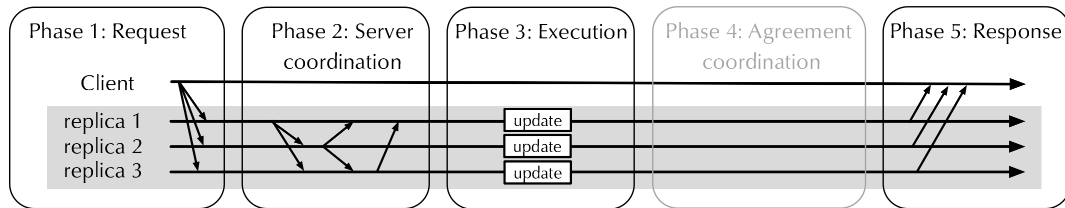

Figure 3.2. Communication steps involved in active replication

3. All replicas deterministically execute the request in the order they are delivered.

4. There is no coordination required after the execution. All replicas should reach the same state.

5. All replicas send the reply to the client.

The involved steps in the passive replication technique can be depicted according to the generic functional model as follows (Figure 3.3):

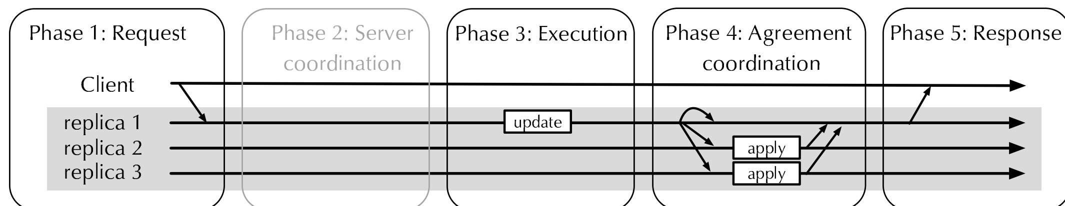

Figure 3.3. Communication steps involved in passive replication

1. The client submits the request to the primary.

2. Server coordination is not required.

3. The primary is the only process that executes the request.

4. The primary coordinates with the backups by broadcasting the update information.

5. The primary sends the reply to the client.

3.2 Partial replication schemes

Full replication ensures the update of every operation to be replicated on every replica in the system. This prevents the system from scaling with the increase in the number of replicas. In fact, the performance of the system is limited by the performance of a single replica. Partial replication addresses these issues by replicating only a subset of data at each replica. Therefore, different subsets of data can be accessed and updated concurrently.

Partitioning is a technique that divides the state of a service in multiple partitions so that most commands access one partition only and are equally distributed among partitions. Partitioned replicated systems can provide highly scalable and available systems; however, they introduce other challenges. Most services cannot be "perfectly partitioned"; that is, the service state cannot be divided in a way that commands access one partition only. Therefore, a partitioning protocol must cope with multi-partition commands. In database systems, partitioning is usually referred to as *sharding*, where the dataset is divided into horizontal fragmentation, known as partitions. Each partition essentially has the same schema and columns but contains different subsets of the total data set. The partitions are then distributed across separate database servers, referred to as physical shards, which can hold multiple logical partitions. Those physical shards can be replicated to tolerate failures. Despite this, the data held within all the partitions collectively represent an entire logical dataset.

To guarantee consistency across multiple replicas of the data, a consensus protocol (e.g., Paxos [48], Raft [59], Viewstamped Replication [58]) is used. Essentially, this protocol works as a quorum mechanism. Any change to the dataset requires a majority of replicas to agree to the change. Google Spanner [22] and Calvin [74] are two database solutions in this category. Both systems were designed to be highly scalable distributed relational databases. The main difference between the two systems is that Spanner uses two-phase commit as an agreement protocol to synchronize the changes between partition, without the need of server coordination before the execution, while Calvin uses a single, global consensus protocol per database to synchronize transactions before the execution. All involved replicas of Calvin then deterministically execute and commit the transaction, without an agreement protocol. The next sections will describe how those database systems work. In addition, we will detail Scalable State Machine Replication, a system in the third category, which requires both server coordination and agreement for providing a strong consistency guarantee.

In general, we can model partitioned replication protocols by extending the five-phase functional model in [78] to include the coordination between compo-

nents in a partitioned replicated system (see Figure 3.4). Those phases are:

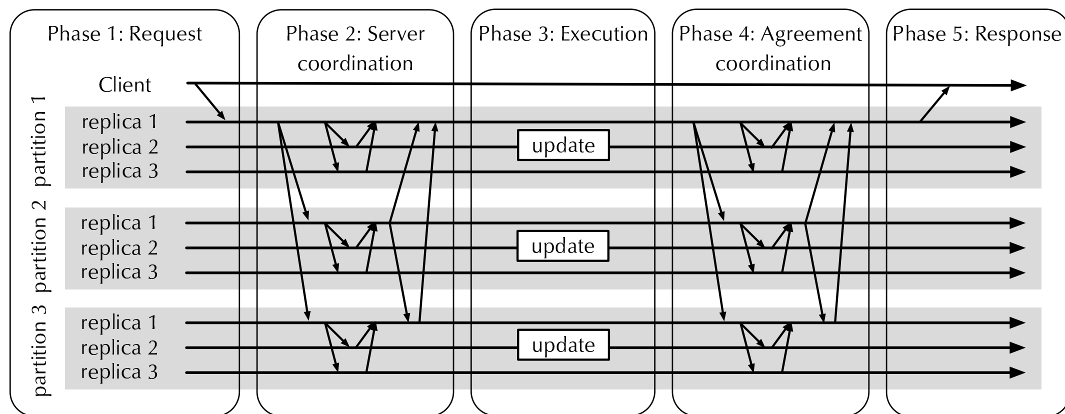

Figure 3.4. Coordination in a partitioned replicated system

- *Request phase (RE):* In the request phase, a client submits the request to the system. This can be done in different ways: the request could be sent to all servers involved in the request or to one server/partition, which would then forward the request to the other processes. In any case, the client should be able to identify the involved partitions that contain the data accessed by the request. The client can send the request directly or use a proxy layer to handle the transmission.
- *Server coordination phase (SC):* The replicas of all partitions that contain the data required by the request will coordinate with each other to synchronize the execution of the request. Depending on the consistency level, concurrent requests will be ordered within and/or across the involved partitions.
- *Execution phase (EX):* The involved servers execute the request.
- *Agreement coordination phase (AC):* The servers of all involved partitions agree on the outcome of the execution.
- *Response phase (END):* The result of the request is sent back to the client.

3.2.1 Agreement coordination without server coordination

Spanner is a NewSQL [31] globally distributed database system developed by Google [22]. Spanner provides features such as global transactions, strongly con-

sistent reads, and automatic multi-site replication and automatic failover. Spanner partitions data into multiple shards, and replicates every shard via Paxos across independent regions for high availability. So, every operation in a transaction is a replicated operation within a Paxos replicated state machine.

Spanner consists of multiple *zones*, each of which is a deployment of Bigtable servers. A zone uses one *zonemaster* to assign data to one hundred to several thousand sets of partitions. Each partition is a set of Paxos-based state machines (so-called *spanservers*). To implement transactions, including transactions that span across multiple partitions, Spanner uses two-phase locking, for concurrency control, and two-phase commit, for transaction termination. Each spanserver implements a lock table to support two-phase-locking and a transaction manager to support distributed transactions. Operations that require synchronization have to acquire a lock from the lock table. The Paxos leader in each partition is responsible for participating in these protocols on behalf of that partition. Clients in Spanner use per-zone *location proxies* to locate the spanservers assigned to serve their data. To order the transaction between partitions, Spanner uses TrueTime API, a combination of GPS and atomic clocks in all of their regions (i.e., zones) to synchronize time within a known uncertainty bound. If two transactions are processed during time periods that do not have overlapping uncertainty bounds, Spanner can be certain that the later transaction will see all the writes of the earlier transaction.

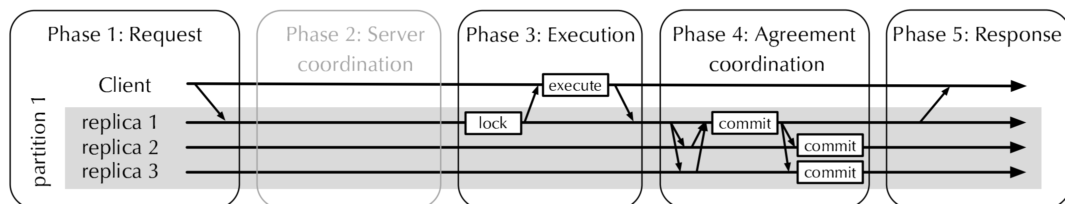

Figure 3.5. Coordination in single-partition transactions in Spanner

Single-partition transactions

In Spanner, if a transaction accesses data in a single partition (single-shard transaction), Spanner processes that transaction as follows (see Figure 3.5). First, the client process queries the location proxy to determine which partitions store the data accessed by the transaction and send the transaction to the Paxos leader of that partition. The leader acquires the read locks on the involved objects and acknowledges the client. The client executes the transaction, then initiates the

commit on the leader. The leader then coordinates with other replicas in its Paxos group to commit the transaction, and responds to the client.

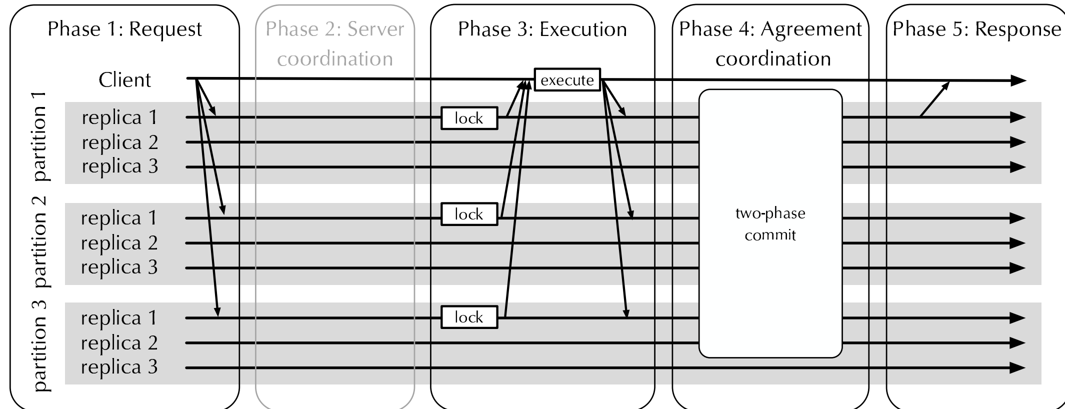

Figure 3.6. Coordination of multi-partition transactions in Spanner

Multi-partition transactions

In the case the transaction accesses more than a single partition, the leaders of all involved partitions have to coordinate and perform a two-phase commit to ensure consistency and use two-phase locking to ensure isolation (see Figure 3.6). First, the client needs to determine the involved partitions by querying a location proxy. Then, the client sends the transaction to the leader of each involved group, which acquires read locks and returns the most recent data. The client then performs the execution of the transaction locally. To commit the transaction, the client chooses a coordinator from the set of involved leaders, then initiates the commit by sending a commit message to each leader, together with the information of the coordinator. On receiving the commit message from the client, the involved leaders coordinate to acquire write locks by performing two-phase locking followed by two-phase commit. After having the transaction committed on all involved partitions, the coordinator leader sends a response to the client.

In general, the transaction processing of Spanner can be mapped to the coordination protocol in Figure 3.4 as follows:

1. Client sends the transaction to the Paxos leader process.
2. There is no initial coordination required between involved replicas.
3. The leader(s) acquires locks and retrieves data for the client. The client executes the transaction and initiates the commit protocol.

4. The leader(s) coordinates with other leaders to commit the transaction.

5. The leader sends the response to the client.

Spanner allows data to be re-sharded across *spanservers*, or *zones* data centers, to balance loads and in response to failures by the *placement driver* [22]. Periodically, the *placement driver* communicates with spanservers to re-arrange data. During these transfers, clients can still execute transactions (including updates) on the database.

3.2.2 Server coordination without agreement coordination

Calvin is a distributed transaction protocol that consists of a transaction scheduling and replication management layer for distributed storage systems [74]. Similar to Spanner, Calvin also shards its data on multiple partitions for performance, and replicates those partitions for availability. The main difference between the two systems is the way Calvin deals with the problem of transaction synchronization. Spanner solves the problem in a traditional way (i.e., two-phase locking and two-phase commit) and reduces the cost of synchronization by decreasing the time during which transactions hold locks using TrueTime API.

By executing transactions using two-phase locking and two-phase commit, traditional database systems have no a priori deterministic transaction order. This means that the servers do not need to ensure some transaction order defined before the execution of the transaction. Calvin chooses to use a deterministic transaction order. This approach could remove the overhead of coordinating after the execution phase, since all the nodes execute the same sequence of input transactions in a deterministic way, to produce the same output. Calvin achieves deterministic order by using a sequencer (*preprocessing*) to let replicas agree on the execution order and transaction boundaries.

Client processes in Calvin first submit transactions to a distributed, replicated log before the transactions are sent to partitions for execution. The sequencer then processes this request log, determines the order in which transactions are executed, and establishes a global transaction input sequence. Then, each replica simply reads and processes transactions from this global order (Figure 3.7).

Essentially, Calvin can be described in the following steps, according to the framework of coordination protocol in Figure 3.4:

1. Client submits the transaction to the sequencer nodes.

2. Sequencers coordinate to order and schedule the execution of the transaction in a global sequence.,

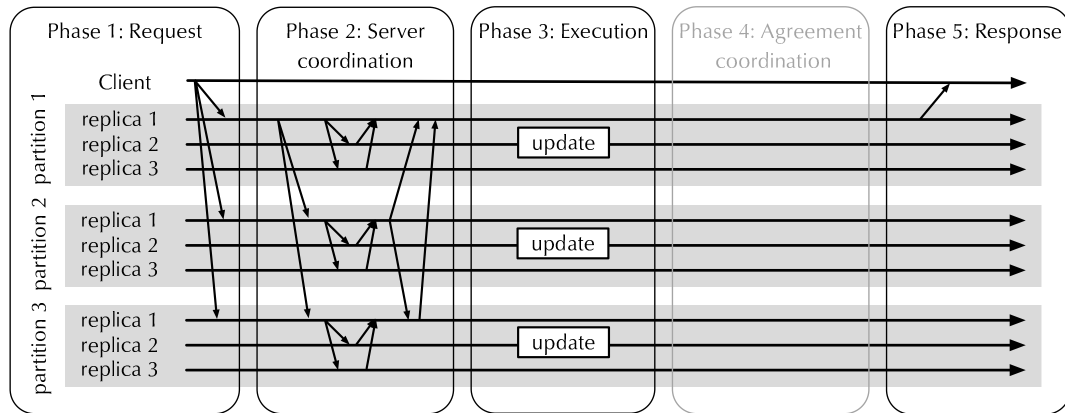

Figure 3.7. Transaction processing in Calvin

3. The involved replicas execute the same sequence of transactions, produce the same output.

4. As all the replicas reach the same state, no agreement coordination is necessary.

5. The replicas send the response to the client.

3.2.3 Server coordination with agreement coordination

Scalable State Machine Replication (S-SMR) is an extension to state machine replication (SMR) that under certain workloads allows performance to grow proportionally to the number of replicas. 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. In addition, S-SMR assumes a static workload partitioning and a location oracle *oracle*, which tells which partitions are accessed by each command.[1]. Any state reorganization requires system shutdown and manual intervention.

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

[1]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.

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.

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. Thus, the performance of S-SMR is particularly sensitive to the way the service state is partitioned.

One way to reduce the number of multi-partition commands is by dynamically changing the partitioning, placing 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.

In summary, the following steps are involved in the processing of a request in S-SMR according to our functional model.

1. Client sends the requests to the servers of all involved partitions.
2. Server coordination is given by the total order property of the Atomic multicast.
3. All involved replicas execute the requests in the order they are delivered.
4. Partitions exchanges signals to ensure linearizability.
5. All replicas send back the result to the client, and the client typically only waits for the first answer.

3.3 Conclusion

In this chapter, we surveyed a number of approaches to replication and partitioning of a replicated system. Replication and partitioning are widely used by several system, from databases, file systems to distributed object systems. We introduced an abstract framework that identifies five basic coordination steps in

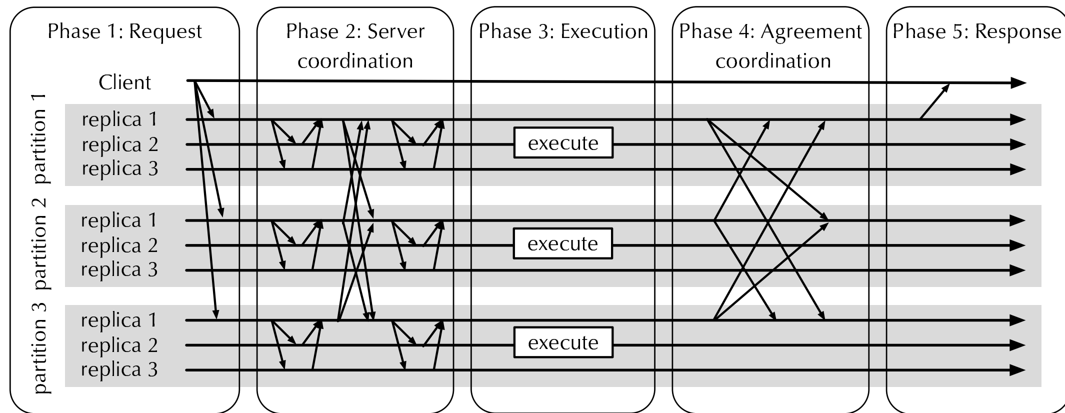

Figure 3.8. Atomic multicast and S-SMR

a partitioned replicated system. This framework allows us to give an insight of design choices of several approaches in the literature. By understanding these approaches, we came up with techniques that aim to minimize the overhead of cross partition coordination, thus providing better performance scalability. We discuss these techniques in the next chapters.

Chapter 4

Dynamic partitioning for SMR

An inherent problem of traditional SMR is that it is not scalable: any replica added to the system will deliver all requests, so throughput is not increased. Scalable SMR addresses this issue in two ways: (i) by partitioning the application state, while allowing every command to access (read/write) any combination of partitions and (ii) using caching to reduce the communication across partitions, while keeping the execution linearizable.

On the downside of this approach, as the number of multi-partition commands increases, performance of S-SMR decreases, as partitions must communicate. One way to reduce the number of multi-partition commands is by dynamically changing the partitioning, placing 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 approach 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. 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 chapter, 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. If there exists a centralized oracle that clients can consult to find out the partitions a command must be multicast to, the system is still

unlikely to scale, as the oracle will likely become a bottleneck. 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 in 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., [23; 73]), 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 following contributions are presented in this chapter: (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 remainder of this chapter is organized as follows. Section 4.1 gives an overview of DS-SMR. Section 4.2 explains the algorithm in detail. Section 4.3 proposes some performance optimizations. Section 4.4 argues about the correctness of the algorithm. Section 4.5 details the implementation of Eyrie. Section 4.6 reports on the performance of Eyrieand Chirper. Section 4.7 concludes the chapter.

4.1 General idea

In Dynamic S-SMR (DS-SMR), the service state, identified by a set of state variables $\mathcal{V} = \{v_1, ..., v_m\}$, is composed of k partitions, $\mathcal{P}_1, ..., \mathcal{P}_k$, where each partition $\mathcal{P}_i$ is assigned to server group $\mathcal{S}_i$. (For brevity, we say that server s belongs to $\mathcal{P}_i$ meaning that $s \in \mathcal{S}_i$). DS-SMR defines a dynamic mapping of variables to partitions. Each variable v is mapped to partition $\mathcal{P}$, meaning that $v \in \mathcal{P}$. Such a mapping is managed by a partitioning oracle, which is implemented as a replicated service run by a group of server processes $\mathcal{S}_0$. 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 $\omega \subseteq \mathcal{V}$ (as described in Section 2), $create(v)$ creates a new variable v and initially maps it to a partition defined by the oracle, $delete(v)$ removes v from the service state, $move(v, \mathcal{P}_s, \mathcal{P}_d)$ moves variable v from partition $\mathcal{P}_s$ to partition $\mathcal{P}_d$, 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 $\langle v, \mathcal{P} \rangle$, meaning that variable v is mapped to partition $\mathcal{P}$. The other possible values for a prophecy are ok and nok, which mean that a command can and cannot be executed, respectively.

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

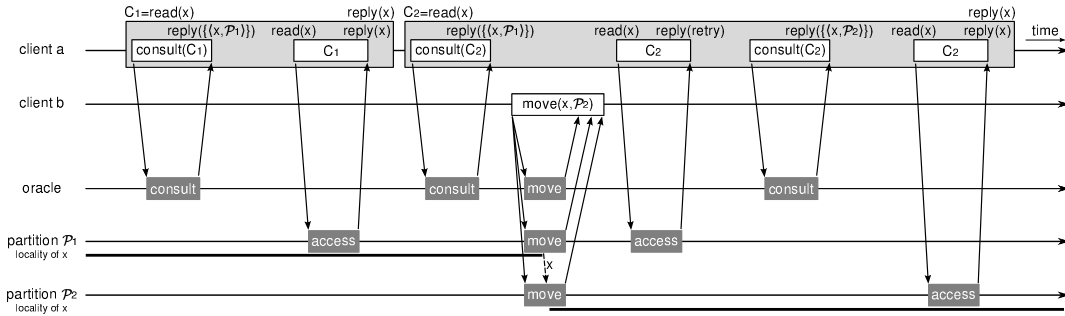

Figure 4.1. Consulting the oracle and issuing a command are done in multiple calls to atomic multicast.

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 4.1. 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

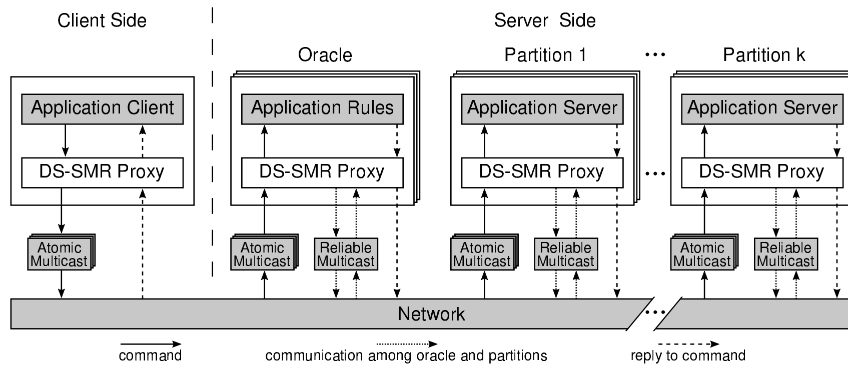

Figure 4.2. The architecture of Dynamic Scalable State Machine Replication.

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 in every partition 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 3.2.3. Every server and oracle process has their 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 4.2 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 1, 2, and 3 describe in detail how the DS-SMR proxy works respectively at client, server and oracle processes. Every server proxy at a server in $\mathcal{S}_i$ has only partial knowledge of the partitioning: it knows only which variables belong to $\mathcal{P}_i$. The oracle proxy has knowledge of every $\mathcal{P} \in \Psi$. To maintain such a global knowledge, the oracle must a-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.)

Algorithm 1 DS-SMR Client Proxy

1: To issue a command C, the client proxy does:
2:   **do**
3:     a-mcast(oracle, $consult(C)$)
4:     wait for $prophecy$
5:     **if** $prophecy \in \{ok, nok\}$ **then**
6:       $reply \leftarrow prophecy$
7:     **else**
8:       $C.dests \leftarrow \{\mathcal{P} : \exists \langle v, \mathcal{P} \rangle \in prophecy\}$
9:       **if** C is an $access(\omega)$ command and $|C.dests| > 1$ **then**
10:         let $\mathcal{P}_d$ be one of the partitions in $C.dests$
11:         **for** each $v \in \omega$ **do**
12:           *// move* v *to partition* $\mathcal{P}_d$
13:           let $\mathcal{P}_s$ be $\mathcal{P} : \langle v, \mathcal{P} \rangle \in prophecy$
14:           **if** $\mathcal{P}_s \neq \mathcal{P}_d$ **then**
15:             $C_{move} \leftarrow move(v, \mathcal{P}_s, \mathcal{P}_d)$
16:             $C_{move}.dests \leftarrow \{$oracle$, \mathcal{P}_s, \mathcal{P}_d\}$
17:             a-mcast($C_{move}.dests$, C_{move})
18:         $C.dests \leftarrow \{\mathcal{P}_d\}$
19:       **if** C is $create$ or $delete$ **then**
20:         $C.dests \leftarrow C.dests \cup \{oracle\}$
21:       a-mcast($C.dests$, C)
22:       wait for $reply$
23:   **while** $reply = retry$ *// after many retries, fall back to S-SMR*
24:   return $reply$ to the application client

The client proxy. To execute a command C, the proxy first consults the or-

acle. 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 randomly 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 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.

The DS-SMR server and oracle proxies coordinate to execute commands that create or move variables. Such coordination is done by means of r-mcast. 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 r-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

Algorithm 2 DS-SMR Server Proxy

```
1: To execute a command C, the server proxy in partition 𝒫 does:
2:    when r-deliver(⟨val, C⟩)
3:       rcvd_msgs ← rcvd_msgs ∪ {⟨val, C⟩}
4:    when a-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:               r-mcast(𝒫_d, ⟨v, C⟩)
15:               𝒫 ← 𝒫 \ {v}
16:            else
17:               r-mcast(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}
```

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 r-mcasts $\langle v, C\rangle$ 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 $v \notin \mathcal{P}$, a $\langle null, C \rangle$ message is r-mcast to $\mathcal{P}_d$, informing $\mathcal{P}_d$ 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 $\langle v, \mathcal{P} \rangle$, meaning that $v \in \mathcal{P}$. 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 has the partition that contains v, or ok, in case v was already deleted or 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 $move(v, \mathcal{P}_s, \mathcal{P}_d)$ command, the oracle checks whether v in fact belongs to the source partition $\mathcal{P}_s$. If that is the case, the command is executed, moving v to $\mathcal{P}_d$. Each $create(v)$ command is multicast to the oracle and to a partition $\mathcal{P}$. If v already exists, the oracle tells $\mathcal{P}$ that the command cannot be executed, by r-mcasting nok to $\mathcal{P}$. The oracle also sends nok to the client as reply, meaning that v already exists. If v does not exist, the oracle tells $\mathcal{P}$ that the command can be executed, by r-mcasting ok to $\mathcal{P}$. 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 $\mathcal{P}$, where the client proxy assumed v to be located. If v belongs to $\mathcal{P}$, 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 1, 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 informa-

Algorithm 3 DS-SMR Oracle Proxy

1: To execute a command C, the oracle proxy does:
2:   **when** a-deliver(C)
3:     **if** C is a $consult(C_c)$ command **then**
4:       $prophecy \leftarrow \emptyset$
5:       **if** C_c is an $access(\omega)$ command **then**
6:         **if** $\exists v \in \omega : (\nexists \mathscr{P} \in \Psi : v \in \mathscr{P})$ **then**
7:           $prophecy \leftarrow nok$
8:         **else**
9:           **for** each $v \in \omega$ **do**
10:             $prophecy \leftarrow prophecy \cup \{\langle v, \mathscr{P} \rangle\} : v \in \mathscr{P}$
11:       **else if** C_c is a $create(v)$ command **then**
12:         **if** $\exists \mathscr{P} \in \Psi : v \in \mathscr{P}$ **then**
13:           $prophecy \leftarrow nok$
14:         **else**
15:           $\mathscr{P} \leftarrow$ initial partition, defined by application rules
16:           $prophecy \leftarrow \{\langle v, \mathscr{P} \rangle\}$
17:       **else if** C_c is a $delete(v)$ command **then**
18:         **if** $\nexists \mathscr{P} \in \Psi : v \in \mathscr{P}$ **then**
19:           $prophecy \leftarrow ok$
20:         **else**
21:           $prophecy \leftarrow \{\langle v, \mathscr{P} \rangle\} : v \in \mathscr{P}$
22:       send $prophecy$ to the client
23:     **else if** C is a $move(v, \mathscr{P}_s, \mathscr{P}_d)$ command **then**
24:       **if** $v \in \mathscr{P}_s$ **then**
25:         $\mathscr{P}_s \leftarrow \mathscr{P}_s \setminus \{v\}$
26:         $\mathscr{P}_d \leftarrow \mathscr{P}_d \cup \{v\}$
27:     **else if** C is a $create(v)$ command **then**
28:       let $\mathscr{P}_c$ be $\mathscr{P} : \{\mathscr{P}\} = C.dests \setminus$ {oracle}
29:       **if** $\nexists \mathscr{P} \in \Psi : v \in \mathscr{P}$ **then**
30:         $outcome \leftarrow ok$
31:       **else**
32:         $outcome \leftarrow nok$
33:       r-mcast($\mathscr{P}_c, \langle outcome, C \rangle$)
34:       send $outcome$ to the client
35:     **else if** C is a $delete(v)$ command **then**
36:       let $\mathscr{P}_d$ be $\mathscr{P} : \{\mathscr{P}\} = C.dests \setminus$ {oracle}
37:       **if** $\nexists \mathscr{P} \in \Psi : v \in \mathscr{P}$ or $v \in \mathscr{P}_d$ **then**
38:         send ok to the client
39:       **else**
40:         send $retry$ to the client

tion. 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 1 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 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 outdated. On the one hand, this may lead a command to be multicast to the wrong set of partitions, which will incur in the client proxy having to retry executing the command. For instance, in Figure 4.3 the client has an outdated 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 4.1. If most commands are executed without consulting the oracle, as in the case of C_4 in Figure 4.3, 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 variables 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 1). 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

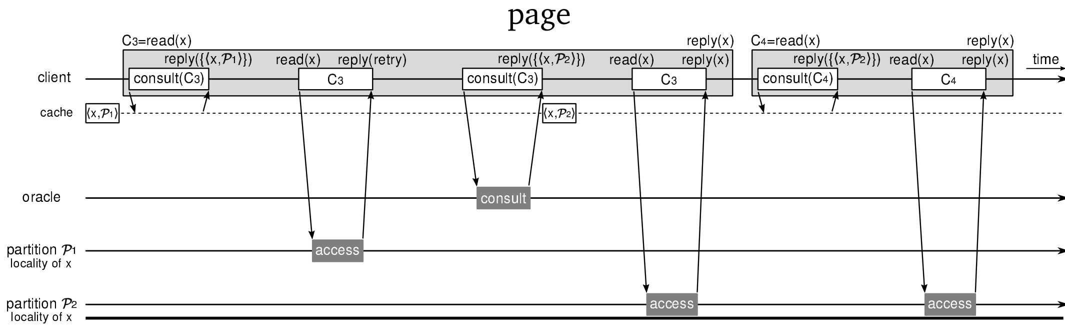

Figure 4.3. Each client proxy in DS-SMR maintains a cache in order to avoid consulting the oracle. White boxes represent actions of the client proxy.

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 multicasts 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 $\mathcal{E}$ of DS-SMR is delivered by atomic multicast and is *execution atomic* (as defined in [9]), then $\mathcal{E}$ 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 a-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 $\mathcal{E}$, then the execution of x and y could be done in any order, which would contradict the supposition that $y \prec x$.

Therefore, there are commands $z_1, ..., z_n$ 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.

4.5 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. Our DS-SMR prototype uses the Ridge atomic multicast library [8], available as open source.[1]. Ridge is a Paxos-based atomic multicast protocol where each message is initially forwarded to a single f distributor, whose responsibility is to propagate the message to all other destinations. Processes in Ridge alternate in the role of distributor to utilize all bandwidth available in the system.

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

4.5.1 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 a 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, Callback)*, *sendAccess(Command, Call-*

back), and *sendDelete(Command, Callback)*. 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 Callback. The error() method is called when a $retry$ reply is received.

4.5.2 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 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.

4.6 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 [9]. In these experiments, we are interested in assessing DS-SMR's performance with workloads that present different levels of locality. By locality, we mean the likelihood that certain groups of data items are accessed together (by the same command). In Section 4.6.1, we describe the environment where we conducted our experiments. In Section 4.6.2, we show the results with strong-locality workloads. In Section 4.6.3, we show the results for weak-locality workloads.

4.6.1 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 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).

4.6.2 Results for strong locality

Figure 4.4 shows the number of commands that involved the moving of objects versus the throughput of the system. The experiment contained Post commands

with the strong-locality workload and four partitions. At the beginning of the experiment, all the user objects were randomly distributed across all partitions. As a result, most of the commands accessed more than one partition. Thus, many commands required coordination between partitions, and objects were moved across partitions. Coordination of so many commands hurt the throughput of the system. This can be observed in the first fifteen seconds of the experiment. As the experiments progressed, due to the effect of locality, those user objects that were often accessed together gradually converged to the same partitions, minimized the movement of the object between partition, and helped to maximize the throughput.

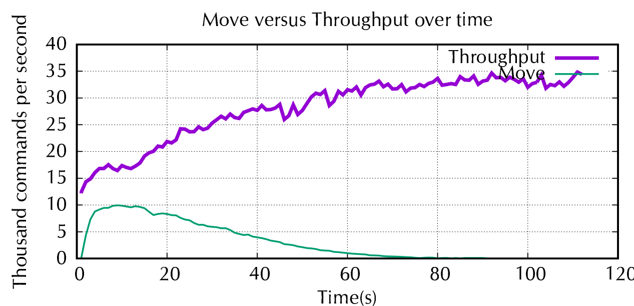

Figure 4.4. Number of commands require moving of objects versus throughput of DS-SMR with Post commands, running on strong-locality workload on 4 partitions

Figure 4.5 and Table 4.1 depict the results achieved with Chirper, running with the strong-locality workload. 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. Every getTimeline request accesses a single User object (of the user whose timeline is being requested). 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 are in different partitions are moved to the same partition based on

Table 4.1. Absolute values of Chirper running S-SMR and DS-SMR.

	Timeline				Post				Follow/unfollow				Mix			
Number of partitions	1	2	4	8	1	2	4	8	1	2	4	8	1	2	4	8
	Throughput (commands per second)															
S-SMR	32561	61220	75812	11867	32483	2893	1882	1190	32541	11476	8580	3371	32151	22803	16822	10657
DS-SMR	29248	56347	70717	11175	19248	23667	35547	54025	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

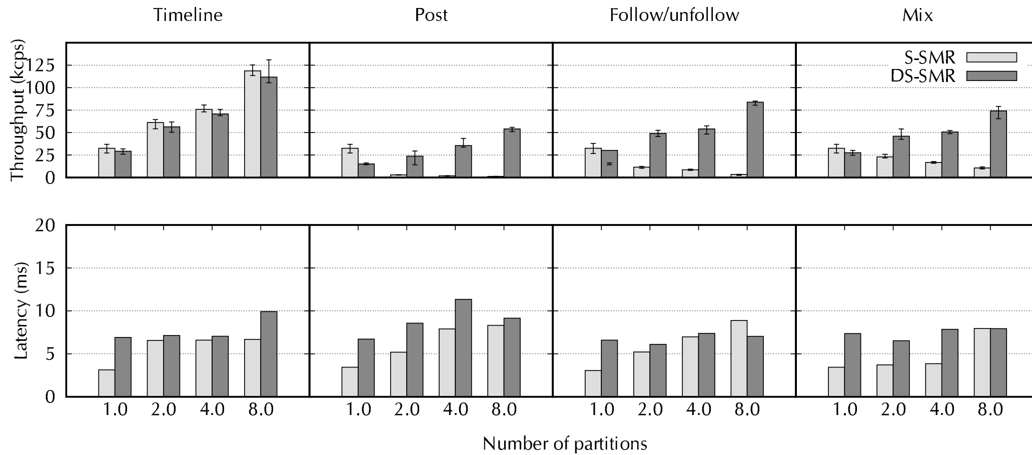

Figure 4.5. Results of Chirper running with S-SMR and DS-SMR with strong-locality workload (throughput in thousands of commands per second, kcps).

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.

In Figure 4.6 we show the cumulative distribution functions (CDFs) of latency for mixed and post workloads of DS-SMR and S-SMR on different configurations of the number of partitions. The results show that the latency distributions for DS-SMR are consistently higher than S-SMR on experiments with a small number of partitions. However, with the higher number of partitions (8 partitions), the latency of DS-SMR improves and in some cases it is even better than S-SMR's in both workloads.

4.6.3 Results for weak locality

Figure 4.7 compares DS-SMR and S-SMR on the weak-locality workload. Although both protocols have similar performance in most of the experiments, S-SMR outperforms DS-SMR in several cases. This is expected, as the objects in DS-SMR are constantly moving back and forth between partitions, without converging to a stable configuration.

Figure 4.8 shows the number of move and throughput of DS-SMR running Post commands on a four-partition configuration. We see that the performance

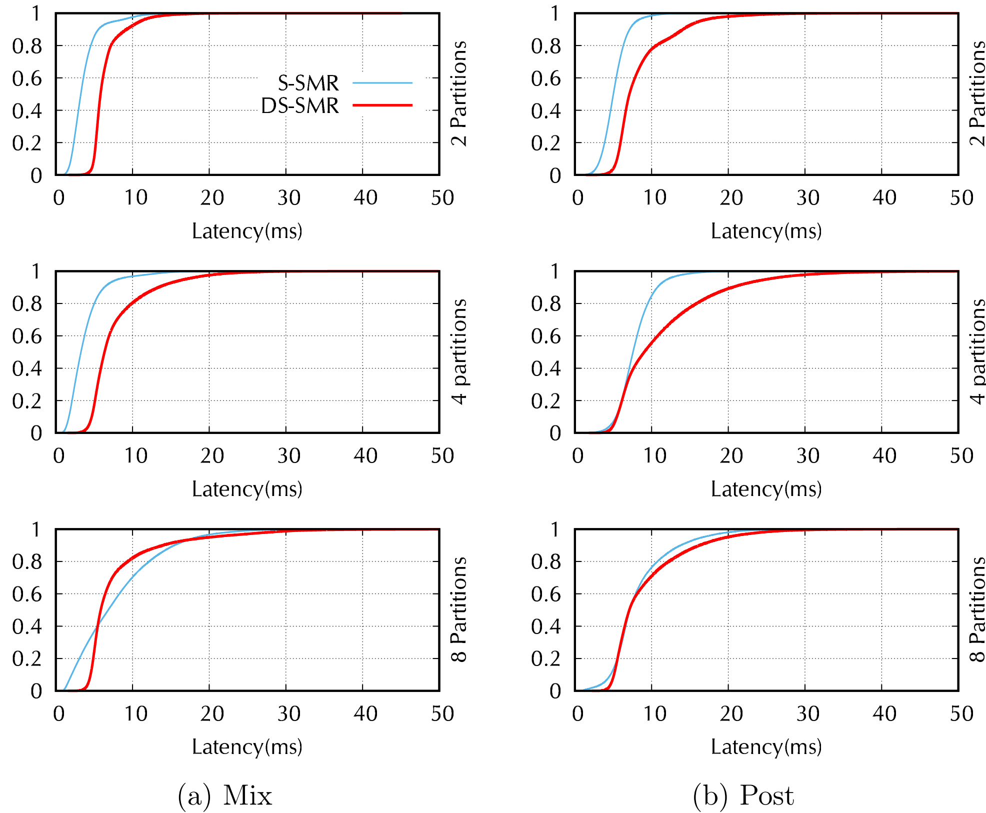

Figure 4.6. Cumulative distribution function (CDF) of latency for mix workloads of DS-SMR and S-SMR.

of DS-SMR decreases significantly, while the number of moves is constantly high. This happens because user objects in DS-SMR will only converge if there is a perfect way to partition the data, that is, data items can be grouped such that eventually no command accesses objects across partitions.

These results show the limitations of DS-SMR. Although DS-SMR works well for data with strong locality, it is unstable for workloads with weak locality. This happens because data can not converge to a favorable partitioning. It also shows that the mechanism of randomly selecting partition in DS-SMR, which allows for a completely decentralized implementation, may not work well in some cases.

4.7 Conclusion

This chapter introduced DS-SMR. It proposes an approach that allows scaling state machine replication by dynamically repartitioning application state based

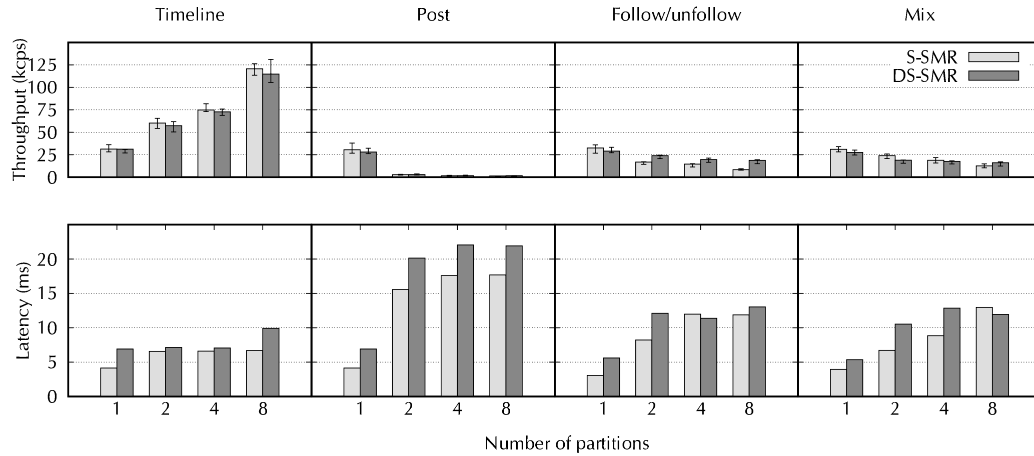

Figure 4.7. Results of Chirper running with S-SMR and DS-SMR with weak-locality workload. Throughput is shown in thousands of commands per second (kcps).

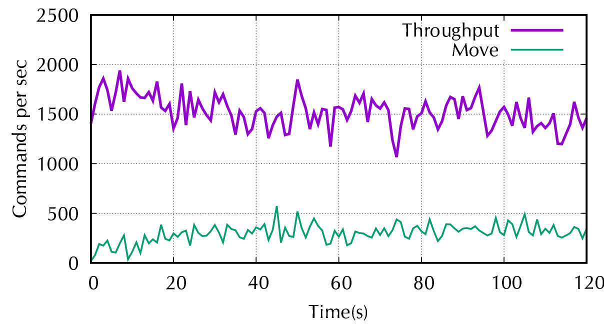

Figure 4.8. Number of commands require moving of objects versus throughput of DS-SMR with Post commands, running on weak-locality workload with 4 partitions.

on the workload. Variables that are usually accessed together are moved to the same partition, which significantly improves scalability. However, in order to reduce the chances of skewed load among partitions in DS-SMR the destination partition is chosen randomly. Although this heuristic algorithm could bring a balanced partitioning, it fails to guarantee optimal partitioning and to minimize the rate of multi-partition commands. In the next chapter, we will discuss how we can reach an optimized partitioning.

Chapter 5

Optimized partitioning for SMR

DS-SMR addresses the limitations of S-SMR by adapting the partitioning scheme as workloads change, by moving data "on demand" to maximize the number of single-partition user commands, while avoiding imbalances in the load of the partitions. The major challenge in this approach is determining how the system selects the partition to which to move data. DS-SMR selects partitions randomly, which allows for a completely decentralized implementation, in which partitions make only local choices about data movement. We refer to this approach as *decentralized partitioning*. This approach works well for data with *strong locality*, but it is unstable for workloads with *weak locality*.[1] This happens because with weak locality, objects in DS-SMR are constantly being moved back and forth between partitions without converging to a stable configuration.

In this chapter, we introduce DynaStar, a new dynamic approach to the state partitioning problem. Like the other dynamic approach, DynaStar does not require any a priori knowledge about the workload. However, DynaStar differs from the prior approach because it maintains a location oracle with a global view of the application state. The oracle minimizes the number of state relocations by monitoring the workload, and re-computing an optimized partitioning on demand using a static partitioning algorithm.

The location oracle maintains two data structures: (i) a mapping of application state variables to partitions, and (ii) a *workload graph* with state variables as vertices and edges as commands that access the variables. Before a client submits a command, it contacts the location oracle to discover the partitions on which

[1]Workloads are referred as *strong locality* if that can be partitioned in a way that would both (i) allow commands to be executed at a single partition only, and (ii) evenly distribute data so that load is balanced among partitions. Conversely, workloads that cannot avoid multi-partition commands with balanced load among partitions exhibit *weak locality*.

state variables are stored. If the command accesses variables in multiple partitions, the oracle chooses one partition to execute the command and instructs the other involved partitions to temporarily relocate the needed variables to the chosen partition. Of course, when relocating a variable, the oracle is faced with a choice of which partition to use as a destination. DynaStar chooses the partition for relocation by partitioning the workload graph using the METIS [1] partitioner and selecting the partition that would minimize the number of state relocations.

To tolerate failures, DynaStar implements the oracle as a regular partition, replicated in a set of nodes. To ensure that the oracle does not become a performance bottleneck, clients cache location information. Therefore, clients only query the oracle when they access a variable for the first time or when cached entries become invalid (i.e., because a variable changed location). DynaStar copes with commands addressed to wrong partitions by telling clients to retry a command.

We implemented DynaStar and compared its performance to other schemes, including an idealized approach that knows the workload ahead of time. Although this scheme is not achievable in practice, it provides an interesting baseline to compare against. DynaStar's performance rivals that of the idealized scheme, while having no a priori knowledge of the workload. We show that DynaStar largely outperforms existing dynamic schemes under two representative workloads, the TPC-C benchmark and a social network service. We also show that the location oracle never becomes a bottleneck and can handle workload graphs with millions of vertices.

In summary, this chapter makes the following contributions:

- It introduces DynaStar and discusses its implementation.
- It evaluates different partitioning schemes for state machine replication under a variety of conditions.
- It presents a detailed experimental evaluation of DynaStar using the TPC-C benchmark and a social network service populated with a real social network graph that contains half a million users and 14 million edges.

The rest of the chapter is structured as follows. Section 5.1 introduces DynaStar and describes the technique in detail. Section 5.2 overviews our prototype. Section 5.6 reports on the results of our experiments. Section 5.7 concludes the chapter.

5.1 General idea

DynaStar defines a dynamic mapping of application variables to partitions. Application programmers may also define other granularity of data when mapping application state to partitions. For example, in our social network application (§5.5.5), each user (together with the information associated with the user) is mapped to a partition; in our TPC-C implementation (§5.5.4), every district in a warehouse is mapped to a partition. Such a mapping is managed by a partitioning oracle, which is handled as a replicated partition. The oracle allows the mapping of variables to partitions to be retrieved or changed during execution. To simplify the discussion, in §5.1–5.2 we initially assume that every command involves the oracle. In §5.3, we explain how clients can use a cache to avoid the oracle in the execution of most commands.

Clients in DynaStar submit commands to the oracle and wait for the reply. DynaStar supports three types of commands: $create(v)$ creates a new variable v and initially maps it to a partition defined by the oracle; $access(\omega)$ is an application command that reads and modifies variables in set $\omega \subseteq \mathcal{V}$; and $delete(v)$ removes v from the service state. The reply from the oracle is called a $prophecy$, and usually consists of a set of tuples $\langle v, \mathcal{P} \rangle$, meaning $v \in \mathcal{P}$, and a target partition $\mathcal{P}_d$ on which the command will be executed. The $prophecy$ could also tell the clients if a command cannot be executed (e.g., it accesses variables that do not exist). If the command can be executed, the client waits for the reply from the target partition.

If a command C accesses variables in ω on a single partition, the oracle multicasts C to that partition for execution. If the command accesses variables on multiple partitions, the oracle multicasts a $global(\omega, \mathcal{P}_d, C)$ command to the involved partitions to gather all variables in ω to the target partition $\mathcal{P}_d$. After having all required variables, the target partition executes command C, sends the reply to the client, and returns the variables to their source.

The oracle also collects hints from clients and partitions to build up a workload graph and monitors the changes in the graph. In the workload graph, vertices represent state variables and edges dependencies between variables. An edge connects two variables in the graph if a command accesses both of them. Periodically, the oracle computes a new optimized partitioning and sends the partitioning plan to all partitions. Upon delivering the new partitioning, the partitions exchange variables and update their state accordingly. DynaStar relocates variables without blocking the execution of commands.

5.2 Detailed Algorithm

Algorithms 4, 5, and 6 describe the client, oracle, and server processes, respectively. We omit the delete command since the coordination involved in the create and delete commands are analogous.

The client process

Algorithm 4 Client

```
1: To issue a command C, the client does:
2:   a-mcast(oracle, exec(C))                    {multicast command to oracle}
3:   wait for prophecy
4:   if prophecy = nok then                      {if receive nok then...}
5:     reply ← prophecy                          {...there's nothing to execute}
6:   else                                        {in this case, prophecy is (dest)}
7:     wait for reponse from a server in prophecy
8:     reply ← reponse
9:   return reply to the application
```

To execute a command C, the client atomically multicasts C to the oracle (Algorithm 4). The oracle replies with a prophecy, which may already tell the client that C cannot be executed (e.g., it needs a variable that does not exist, it tries to create a variable that already exists). If C can be executed, the client receives a prophecy containing the partition where C will be executed. The client then waits for the result of the execution of C.

The oracle

When the oracle delivers a request, it distinguishes between two cases (Task 1 in Algorithm 5).

- If the command is to create a variable v, and v does not already exist, the oracle chooses a random partition for v, multicasts the create command to the partition and itself, and returns the partition to the client as a prophecy (Figure 5.1).

- If the command reads and writes existing variables, the oracle first checks that all such variables exist. If the variables exist and they are all in a single partition, the oracle multicasts the command to that partition for execution.

Algorithm 5 Oracle

1: *Initialization:*
2: $changes \leftarrow 0$
3: $\mathscr{V} \leftarrow \emptyset$
4: $\mathscr{L} \leftarrow \emptyset$
5: **when** a-deliver($exec(C)$) *{Task 1}*
6: **case** C is a $create(v)$ command:
7: **if** $partition(\{v\}) \neq \perp$ **then** *{if v already exists...}*
8: $prophecy \leftarrow nok$ *{...notify client}*
9: **else** *{if v doesn't exist...}*
10: $\mathscr{P} \leftarrow$ choose v's partition
11: $prophecy \leftarrow \mathscr{P}$ *{prepare client's response}*
12: a-mcast($\{oracle, \mathscr{P}\}, \langle \mathscr{P}, create(v) \rangle$)
13: **case** C is any command, but $create(v)$:
14: $\omega \leftarrow vars(C)$ *{variables accessed by C}*
15: **if** $\exists v \in \omega : partition(\{v\}) = \perp$ **then** *{if v ¬exists:}*
16: $prophecy \leftarrow nok$ *{tell the client}*
17: **else** *{if all vars in ω exist}*
18: $dests \leftarrow partition(\omega)$ *{get all partition involved}*
19: $\mathscr{P}_d \leftarrow target(dests, C)$ *{$\mathscr{P}_d$ will excecute C}*
20: a-mcast($dests, C$)
21: $prophecy \leftarrow \mathscr{P}_d$
22: send $prophecy$ to the client
23: **when** a-deliver($\langle \mathscr{P}_v, create(v) \rangle$) *{Task 2}*
24: r-mcast($\{\mathscr{P}_v, oracle\}, \langle signal, create(v) \rangle$) *{exchange signal...}*
25: wait until $\langle signal, create(v) \rangle \in rcvd_msgs$ *{...to coordinate}*
26: $\mathscr{V} \leftarrow \mathscr{V} \cup \{v\}$ *{add v to service state}*
27: $\mathscr{L} \leftarrow \mathscr{L} \cup \{\langle v, \mathscr{P}_v \rangle\}$
28: **when** r-deliver ($\langle val, C \rangle$) *{Task 3}*
29: $rcvd_msgs \leftarrow rcvd_msgs \cup \{\langle val, C \rangle\}$
30: **when** a-deliver($hint(V_h, E_h)$) *{Task 4}*
31: update G_W with (V_h, E_h)
32: $changes \leftarrow changes + 1$ *{increase number of changes in graph}*
33: **if** $changes \geq threshold$ **then**
34: $\mathscr{L}_{opt} \leftarrow$ optimize($\mathscr{L}, G_W$)
35: a-mcast($\{oracle, all_partitions\}, \mathscr{L}_{opt}$)
36: $changes \leftarrow 0$
37: **when** a-deliver($\mathscr{L}_{opt}$) *{Task 5}*
38: $\mathscr{L} \leftarrow \mathscr{L}_{opt}$

Algorithm variables:

G_W: the graph represent the service state
$changes$: number of changes in the graph
$optimize$: the function to calculate new optimized partitioning from G_W and $\mathscr{L}$
$\mathscr{V}$: the service state
$\mathscr{L}$: the location mapping of state variables

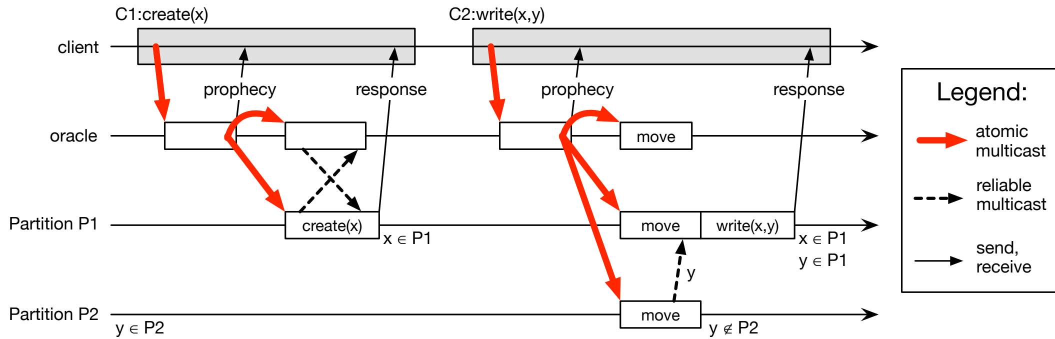

Figure 5.1. Execution of a create (C1) and a write without client cache (C2) and with client cache (C3) in DynaStar.

> If the variables are distributed in multiple partitions, the oracle deterministically determines the destination partition, and atomically multicasts a command to the involved partitions so that all variables are gathered at the destination partition. The oracle chooses as the destination partition the partition that contains most of the variables needed by the command. (In case of a tie, one partition is chosen deterministically, among those that contain most variables.) Once the destination partition has received all variables needed by the command, it executes the command and returns the variables to their source partition.

Upon delivering a create (Task 2), the oracle updates its partition information. The exchange of signals between the partition where the variable will be created and the oracle ensures that interleaved executions between create and delete commands will not lead to violations of linearizability (i.e., this is essentially the execution of a multi-partition command involving the oracle and a partition (Task 3) [9]). The oracle also keeps track of the workload graph by receiving hints with variables (i.e., vertices in the graph) and executing commands (i.e., edges in the graph) (Task 4). These hints can be submitted by the clients or by the partitions, which collect data upon executing commands and periodically inform the oracle. The oracle computes a partitioning plan of the graph and multicasts it to all servers and to itself. Upon delivering a new partition plan, the oracle updates its location map accordingly (Task 5).

To compute an optimized partitioning, the oracle uses a graph partitioner. A new partitioning can be requested by the application, by a partition, or by the oracle itself (e.g., upon delivering a certain number of hints). To determine the destination partition of a set of variables, as part of a move, the oracle uses its mapping of the current location of variables and the last computed partitioning.

The server process

Algorithm 6 Server in partition $\mathscr{P}$

1: *Initialization:*
2: $\mathscr{V}_{\mathscr{P}} \leftarrow \emptyset$
3: $rcvd_msgs \leftarrow \emptyset$
4: **when** a-deliver(C) {*Task 1*}
5: $\omega \leftarrow vars(C)$ {*variables accessed by* C}
6: **if** $\omega \subseteq \mathscr{V}_{\mathscr{P}}$ **then** {*Task 1a*}
7: execute command C
8: send response to the client
9: **else** {*Task 1b*}
10: $dests \leftarrow partition(\omega)$ {*get all involved partition*}
11: $\mathscr{P}_d \leftarrow target(dests, C)$ {$\mathscr{P}_d \in dest$ *will execute* C}
12: $\mathscr{P}_s \leftarrow dests \setminus \mathscr{P}_d$ {$\mathscr{P}_s \in dests$ *will send variables*}
13: **if** $\mathscr{P} = \mathscr{P}_d$ **then** {$\mathscr{P}$ *is the target partition:*}
14: $vars \leftarrow \omega \setminus \mathscr{V}_{\mathscr{P}}$
15: wait until $\forall v \in vars : \exists \langle v, C \rangle \in rcvd_msgs$ {*wait for needed variables*}
16: execute command C
17: send response to the client
18: r-mcast($\mathscr{P}_s, \langle vars, C \rangle$) {*return the variables*}
19: **else** {*if* $\mathscr{P}$ *is not the target partition:*}
20: $vars \leftarrow \omega \cap \mathscr{V}_{\mathscr{P}}$ {*all needed variables in* $\mathscr{P}$}
21: r-mcast($\mathscr{P}_d, \langle vars, C \rangle$) {*send variables*}
22: wait until $\exists \langle vars, C \rangle \in rcvd_msgs$
23: **when** a-deliver($\langle \mathscr{P}_v, create(v) \rangle$) {*Task 2*}
24: r-mcast($\{\mathscr{P}_v, oracle\}, \langle signal, create(v) \rangle$) {*exchange signal*}
25: wait until $\langle signal, create(v) \rangle \in rcvd_msgs$ {*...coordinate*}
26: $\mathscr{V}_{\mathscr{P}} \leftarrow \mathscr{V}_{\mathscr{P}} \cup \{v\}$
27: send *ok* to the client
28: **when** a-deliver($\mathscr{L}_{opt}$) {*Task 3*}
29: **for each** $\mathscr{Q} \in \{\mathscr{P}_1, ..., \mathscr{P}_k\} \wedge \mathscr{Q} \neq \mathscr{P}$ **do**
30: $vars_{\mathscr{Q}} \leftarrow \{v : \langle v, \mathscr{Q} \rangle \in \mathscr{L}_{opt} \wedge v \in \mathscr{V}_{\mathscr{P}}\}$ {*variables of* $\mathscr{Q}$ *that are in* $\mathscr{P}$}
31: r-mcast($\mathscr{Q}, \langle vars_{\mathscr{Q}}, \mathscr{Q} \rangle$) {*send variables from* $\mathscr{P}$ *to* $\mathscr{Q}$}
32: $vars_{\mathscr{P}} \leftarrow \{v : \langle v, \mathscr{P} \rangle \in \mathscr{L}_{opt} \wedge v \notin \mathscr{V}_{\mathscr{P}}\}$
33: wait until $\forall v \in vars_{\mathscr{P}} : \exists \langle v, \mathscr{P} \rangle \in rcvd_msgs$
34: **when** r-deliver ($\langle val, C \rangle$) {*Task 4*}
35: $rcvd_msgs \leftarrow rcvd_msgs \cup \{\langle val, C \rangle\}$

Algorithm variables:

$\mathscr{L}_{opt}$: the optimized partition configuration from the $oracle$
$\mathscr{V}_{\mathscr{P}}$: the subset of service state $\mathscr{V}$ in partition $\mathscr{P}$

When a server delivers a command C, it first checks if it has all variables

needed by C. If the server has all such variables, it executes C and sends the response back to the client (Tasks 1a and 2 in Algorithm 6). If not all the variables needed by C are in that partition, the server runs a deterministic function to determine the destination partition to execute C (Task 1b). The function uses as input the variables needed by C and C itself. In this case, each server that is in the multicast group of C but is not the destination partition sends all the needed variables stored locally to the destination partition and waits to receive them back. The destination partition waits for a message from other partitions. Once all variables needed are available, the destination partition executes the C, sends the response back to the client, and returns the variables to their source. Periodically, the servers deliver a new partitioning plan from the oracle (Task 3). Each server will send the variables to the designated partition, as in the plan, and wait for variables from other partitions. Once a server receives all variables, it updates its location map accordingly. To determine the destination partition for a command, the servers uses the last computed partitioning.

Algorithm 7 Shared functions of the oracle and servers $\mathscr{P}$

1: **function** target($\mathscr{P}, C$)
2: $\quad \mathscr{P}_d \leftarrow$ deterministically compute partition to execute command C from $\forall \mathscr{P}_i \in \mathscr{P}$
3: $\quad$ return $\mathscr{P}_d$
4: **function** partition($vars$)
5: $\quad dests \leftarrow \{\mathscr{P} : \exists v \in vars \cap \mathscr{P}\}$ *{get all involved partitions}*
6: $\quad$ return $dests$

5.3 Performance optimizations

In the algorithm presented in the previous section, clients always need to involve the oracle, and the oracle dispatches every command to the partitions for execution. Obviously, if every command involves the oracle, the system is unlikely to scale, as the oracle will likely become a bottleneck. To address this issue, clients are equipped with a location cache. Before submitting a command to the oracle, the client checks its location cache. If the cache contains the partition of the variables needed by the command, the client can atomically multicast the command to the involved partition and thereby avoid contacting the oracle.

The client still needs to contact the oracle in one of these situations: (a) the cache contains outdated information; or (b) the command is a create, in which

case it must involve the oracle, as explained before. If the cache contains outdated information, it may address a partition that does not have the information of all the variables accessed by the command. In this case, the addressed partition tells the client to retry the command. The client then contacts the oracle and updates its cache with the oracle's response. Although outdated cache information results in execution overhead, it is expected to happen rarely since repartitioning is not frequent.

5.4 Correctness

Similar to DS-SMR, we consider linearizability consistency criterion of DynaStar. To prove that DynaStar ensures linearizability, we must show that for any execution σ of the system, there is a total order π on client commands that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time precedence of commands (§5.4). Let π be a total order of operations in σ that respects $<$, the order atomic multicast induces on commands.

To argue that π respects the semantics of commands, let C_i be the i-th command in π and p a process in partition $\mathscr{P}_p$ that executes C_i. We claim that when p executes C_i, it has updated values of variables in $vars(C_i)$, the variables accessed by C_i. We prove the claim by induction on i. The base step trivially holds from the fact that variables are initialized correctly. Let $v \in vars(C_i)$, C_v be the last client command before C_i in π that accesses v, and q a process in $\mathscr{P}_q$ that executes C_v. From the inductive hypothesis, q has an updated value of v when it executes C_v. There are two cases to consider: (a) $p = q$. In this case, p obviously has an updated value of v when it executes C_i since no other command accesses v between C_v and C_i. (b) $p \neq q$. Since processes in the same partition execute the same commands, it must be that $\mathscr{P}_p \neq \mathscr{P}_q$. From the algorithm, when q executes C_v, $v \in \mathscr{P}_q$ and when p executes C_i, $v \in \mathscr{P}_p$. Thus, q executed a command to move v to another partition after executing C_v and p executed a command to move v to $\mathscr{P}_p$ before executing C_i. Since there is no command that accesses v between C_v and C_i in π, q has an updated v when it executes C_v (from inductive hypothesis), and p receives the value of v at q, it follows that p has an updated v when it executes C_i.

We now argue that there is a total order π that respects the real-time precedence of commands in σ. Assume C_i ends before C_j starts, or more precisely, the time C_i ends at a client is smaller than the time C_j starts at a client, $t^{cli}_{end}(C_i) < t^{cli}_{start}(C_j)$. Since the time C_i ends at the server from which the client receives the

response for C_i is smaller than the time C_i ends at the client, $t^{srv}_{end}(C_i) < t^{cli}_{end}(C_i)$, and the time C_j starts at the client is smaller than the time C_j starts at the first server, $t^{cli}_{start}(C_j) < t^{srv}_{start}(C_j)$, we conclude that $t^{srv}_{end}(C_i) < t^{srv}_{start}(C_j)$.

We must show that either $C_i \prec C_j$; or neither $C_i \prec C_j$ nor $C_j \prec C_i$. For a contradiction, assume that $C_j \prec C_i$ and let C_j be executed by partition $\mathscr{P}_j$.

There are two cases:

(a) C_i is a client command executed by $\mathscr{P}_j$. In this case, since C_i only starts after C_j at a server, it follows that $t^{srv}_{end}(C_j) < t^{srv}_{start}(C_i)$, a contradiction.

(b) C_i is a client command executed by $\mathscr{P}_i$ that first involves a move of variables $vars$ from $\mathscr{P}_j$ to $\mathscr{P}_i$. At $\mathscr{P}_j$, $t^{srv}_{end}(C_j) < t^{srv}_{start}(global(vars, \mathscr{P}_j, \mathscr{P}_i))$ since the move is only executed after C_j ends. Since the move only finishes after variables in $vars$ are in $\mathscr{P}_i$ and C_i can be executed, it must be that $t^{srv}_{end}(global(vars, \mathscr{P}_j, \mathscr{P}_i)) < t^{srv}_{start}(C_i)$. We conclude that $t^{srv}_{end}(C_j) < t^{srv}_{start}(C_i)$, a contradiction.

Therefore, either $C_i \prec C_j$ and from the definition of π, C_i precedes C_j or neither $C_i \prec C_j$ nor $C_j \prec C_i$, and there is a total order in which C_i precedes C_j.

For termination, we argue that every correct client eventually receives a response for every command C that it issues. This assumes that every partition (including the oracle partition) is always operational, despite the failure of some servers in the partition. For a contradiction, assume that some correct client submits a command C that is not executed. Atomic multicast ensures that C is delivered by the involved partition. Therefore, C is delivered at a partition that does not contain all the variables needed by C. As a consequence, the client retries with the oracle, which moves all variables to a single partition and requests the destination partition to execute C, a contradiction that concludes our argument.

5.5 Implementation

5.5.1 Atomic multicast

Our DynaStar prototype uses the BaseCast atomic multicast protocol [20], available as open source.[2] Each group of servers in BaseCast executes an instance of Multi-Paxos.[3] Groups coordinate to ensure that commands multicast to multi-

[2] https://bitbucket.org/paulo_coelho/libmcast
[3] http://libpaxos.sourceforge.net/paxos_projects.php#libpaxos3

ple groups are consistently ordered (as defined by the atomic multicast properties §2.1.2). BaseCast is a genuine atomic multicast in that only the sender and destination replicas of a multicast message communicate to order the multicast message.

5.5.2 Graph partitioning with METIS

The default implementation of DynaStar uses METIS[4] to provide a partitioning based on the workload graph. METIS is a graph partitioning method based on multilevel l-way partitioning algorithms [42], which is able to compute a balanced bipartition. The overall sequence of METIS consists of three phases: coarsening, initial partitioning, and refining. In the first phase, coarsening, METIS takes a large graph and creates successively smaller graphs that are good representations of the original graph. METIS then partitions the smaller graph in the second phase. In the third phase, METIS projects and refines the partition of the smaller graph onto the original graph. While partitioning a graph, METIS aims to reduce the number of multi-partition commands (edge-cuts) while trying to keep the various partitions balanced. Recently, METIS has been improved in performance especially for power-law graphs [1].

We configured METIS to allow a 20% unbalance among partitions. METIS repartitions a graph without considering previous partitions. Consequently, DynaStar may need to relocate a large number of objects upon a repartitioning. Other partitioning techniques could be used to introduce incremental graph partitioning [68].

5.5.3 DynaStar

Our DynaStar prototype is written as a Java 8 library. Application designers who use DynaStar to implement a replicated service must extend three key classes:

- *PRObject*: provides a common interface for replicated data items. All instance of *PRObject*'s sub-classes are replicated by the framework(i.e., objects are distributed among partitions). The application is agnostic to the location of those objects. DynaStar intercepts all calls to replicated objects and forward them to associated partition automatically.

- *ProxyClient*: provides the communication between application client and server, while encapsulating all complex logic of handling caches or retried

[4]http://glaros.dtc.umn.edu/gkhome/views/metis

requests. The proxy client also allows sending multiple asynchronous requests, and delivering corresponding response to each request.

– *PartitionStateMachine*: encapsulates the logic of the server proxy. The server logic is written without knowledge of the actual partitioning scheme. In other words, developer programs for classical state machine replication (i.e., full replication). The DynaStar library handles all communication between partitions and the oracle transparently. Objects that are involved in application's command will be available to the partition at the time it is accessed by the partitions.

– *OracleStateMachine*: computes the mapping of objects to partitions. The oracle can be configured to trigger repartitioning in different ways: repartitioning request from application, based on the number of changes to the graph, or based on some interval of time.

We note one important implementation detail. The oracle is multi-threaded: it can serve requests while computing a new partitioning concurrently. To ensure that all replicas start using the new partitioning consistently, the oracle identifies each partitioning with a unique id. When an oracle replica finishes a repartitioning, it atomically multicasts the id of the new partitioning to all replicas of the oracle. The first delivered id message defines the order of the new partitioning with respect to other oracle operations.

5.5.4 TPC-C benchmark

TPC-C is an industry standard for evaluating the performance of OLTP systems [37]. TPC-C implements a wholesale supplier company. The company has a number of distributed sales districts and associated warehouses. Each warehouse has 10 districts. Each district services 3,000 customers. Each warehouse maintains a stock of 100,000 items. The TPC-C database consists of 9 tables and five transaction types that simulate a warehouse-centric order processing application: *New-Order* (45% of transactions in the workload), *Payment* (43%), *Delivery* (4%), *Order-Status* (4%) and *Stock-Level* (4%).

We implemented a Java version of TPC-C that runs on top of DynaStar. Each row in TPC-C tables is an object in DynaStar. The oracle models the workload at the granularity of districts, thus each district and warehouse is a node in the graph. If a transaction accesses a district and a warehouse, the oracle will create an edge between that district and the warehouse. The objects (e.g., customers, orders) that belong to a district are considered part of district. However, if a

transaction requires objects from multiple districts, only those objects will be moved on demand, rather than the whole district. The ITEM table is replicated in all servers, since it is not updated in the benchmark. A transaction is a set of commands that access those objects, implemented as server procedures.

5.5.5 Social network application

Using DynaStar, we have also developed a Twitter-like social network service, named Chirper. In our social network, users can follow, unfollow, post, or read other users' tweets according to whom the user is following. Like Twitter, users are constrained to posting 140-character messages.

Each user in the social network corresponds to a node in a graph. If one user follows another, a directed edge is created from the follower to the followee. Each user has an associated *timeline*, which is a sequence of post messages from the people that the user follows. Thus, when a user issues a post command, it results in writing the message to the timeline of all the user's followers. In contrast, when users read their own timeline, they only need to access the state associated with their own node.

Since DynaStar guarantees linearizable executions, any causal dependencies between posts in Chirper will be seen in the correct order. More precisely, if user B posts a message after receiving a message posted by user A, no user who follows A and B will see B's message before seeing A's message.

Overall, in Chirper, post, follow or unfollow commands can lead to object moves. Follow and unfollow commands can involve at most two partitions, while posts may require object moves from many partitions.

5.5.6 Alternative system

We compare DynaStar to an optimized version of S-SMR [9] and DS-SMR. S-SMR scales performance with the number of partitions under a variety of workloads. It differs from DynaStar in two important aspects: multi-partition commands are executed by all involved partitions, after the partitions exchange needed state for the execution of the command, and S-SMR supports static state partitioning. In our experiments, we manually optimize S-SMR's partitioning with knowledge about the workload. In the experiments, we refer to this system and configuration as S-SMR*.

5.6 Performance evaluation

In this section, we report results from two benchmarks: TPC-C and Chirper social networking service described in the previous section. Our experiments show that DynaStar is able to rapidly adapt to changing workloads, while achieving throughputs and latencies far better than the existing state-of-the-art approaches to state machine replication partitioning.

5.6.1 Experimental environment

We conducted all experiments on Amazon EC2 T2 large instances (nodes). Each node has 8 GB of RAM, two virtual cores and is equipped with an Amazon EBS standard SSD with a maximal bandwidth 10000 IOPS. All nodes ran Ubuntu Server 16.04 LTS 64 and had the OpenJDK Runtime Environment 8 with the 64-Bit Server VM (build 25.45-b02). In all experiments, the oracle had the same resources as every other partition: 2 replicas and 3 Paxos acceptors (in total five nodes per partition).

5.6.2 Methodology and goals

The experiments seek to answer the following questions:

- *What is the impact of repartitioning on a real dataset?*
- *How does partitioning affect performance when the workload grows with the number of partitions and when the workload has constant size?*
- *How does DynaStar performance compare to other approaches?*
- *How does DynaStar perform under dynamic workloads?*
- *What is the performance of the oracle?*

Performance metrics. The latency was measured as the end-to-end time between issuing the command, and receiving the response. Throughput was measured as the number of posts/second or transactions/second that the clients were able to send.

5.6.3 TPC-C benchmark

In the experiments in this section, we deploy as many partitions as the number of warehouses.

The impact of graph repartitioning

In order to assess the impact of state partitioning on performance, we ran the TPC-C benchmark on an un-partitioned database. Figure 5.2 shows the performance of DynaStar with 8 warehouses and 8 partitions. At the first part of the experiment, all the variables are randomly distributed across all partitions. As a result, almost every transaction accesses all partitions. Thus, every transaction required coordination between partitions, and objects were constantly moving back and forth. This can be observed in the first 50 seconds of the experiment depicted in Figure 5.2: low throughput (i.e., a few transactions executed per second), high latency (i.e., up to several seconds), and a high percentage of cross-partition transactions.

After 50 seconds, the oracle computed a new partitioning based on previously executed transactions and instructed the partitions to apply the new partitioning. When the partitions delivered the partitioning request, they exchanged objects to achieve the new partitioning. It takes about 10 seconds for partitions to reach the new partitioning. During the repartitioning, transactions that access objects that are not being relocated will continue to process. After the state is relocated, most objects involved in a transaction can be found in a local partition, which considerably increases performance and reduces latency.

Scalability

In order to show how DynaStar scales out, we varied the number of partitions from 1 to 128 partitions. We used sufficient clients to saturate the throughput of the system in each experiment. Figure 5.3 shows the peak throughput of DynaStar and S-SMR* as we vary the number of partitions. Notice that we increase the state size as we add partitions (i.e., there is one warehouse per partition). The result shows that DynaStar is capable of applying a partitioning scheme that leads to scalable performance.

5.6.4 Social network

We used the Higgs Twitter dataset [49] as the social graph in the experiments. The graph is a subset of the Twitter network that was built based on the monitoring of the spreading of news on Twitter after the discovery of a new particle with the features of the elusive Higgs boson on 4th July 2012. The dataset consists of 456631 nodes and more than 14 million edges.

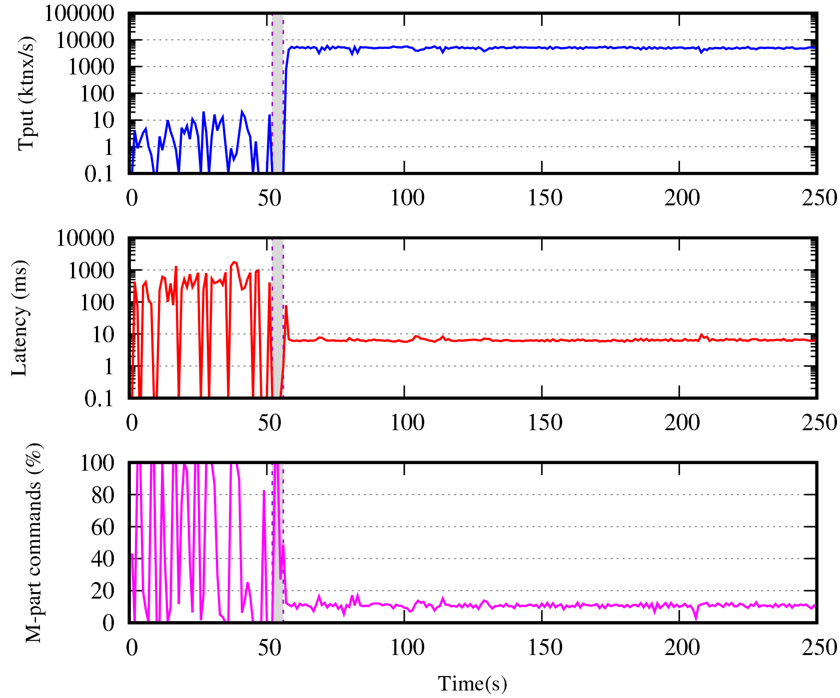

Figure 5.2. Repartitioning in DynaStar; throughput (top), objects exchanged between partitions (middle), and percentage of multi-partition commands (bottom).

We evaluate the performance of DynaStar and other techniques. With S-SMR*, we used METIS to partition the data in advance. Thus, S-SMR* started with an optimized partitioning. DynaStar started with random location of the objects. Each client issues a sequence of commands. For each command, the client selects a random node as the active user with Zipfian access pattern ($\rho = 0.95$). We focused on two types of workloads: timeline-only commands and mix commands (85% timeline and 15% post). Each client operates in a closed loop, that is, the client issues a command and then waits from the response to submit the next command.

DynaStar vs. other techniques

Figure 5.4 and 5.5 shows the peak throughput and latency for approximately 75% of peak throughput (average and 95-th percentile) of the evaluated techniques as we vary the number of partitions of the fixed graph for the social networks.

In the experiment with timeline commands, all three techniques perform sim-

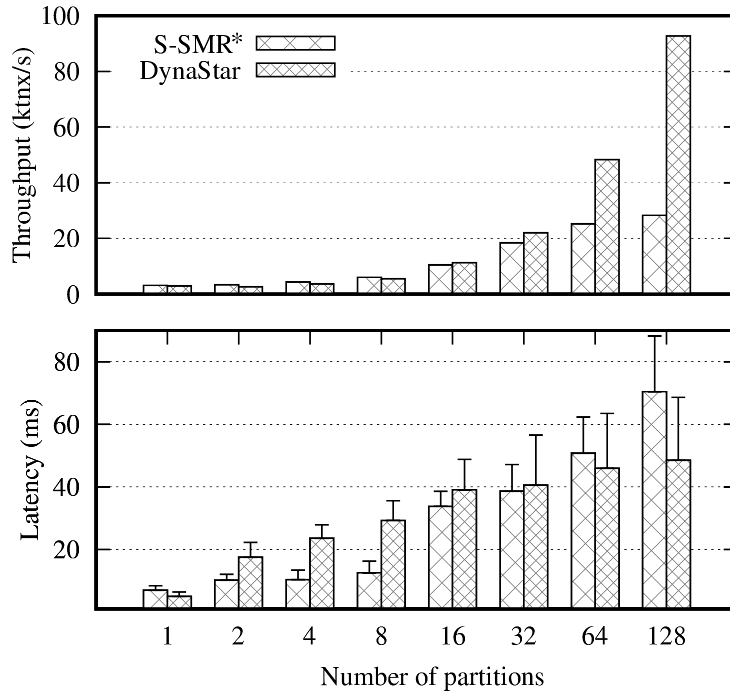

Figure 5.3. Performance scalability with TPC-C. Throughput (in thousands of transactions per second, ktps) and latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile).

ilarly. This happens because no moves occur in DynaStar or DS-SMR, and no synchronization among partitions is necessary for S-SMR* in this case. Consequently, all three schemes scale remarkably well, and the difference in throughput between each technique is due to the implementation of each one.

In the experiment with the mix workload, we see that DS-SMR performance decreases significantly. This happens because objects in DS-SMR will only converge if there is a perfect way to partition the data, that is, data items can be grouped such that eventually no command accesses objects across partitions. In the mix workload experiments, objects in DS-SMR are constantly moving back and forth between partition without converging to a stable configuration.

In contrast, for DynaStar and S-SMR*, the throughput scales with the number of partitions in experiments with up to 8 partitions. However, increasing the number of partitions to 16 with the fixed graph reveals a tradeoff: additional partitions should improve performance as there are more resources to execute commands, but the number of edge cuts increases with the number of partitions, and hurts performance as there are additional operations involving multiple par-

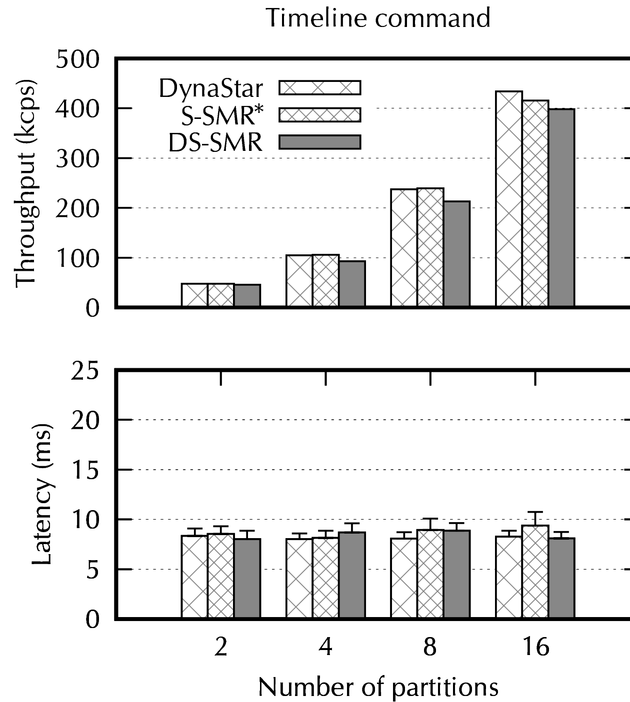

Figure 5.4. Performance of timeline command of social network service. Throughput (in thousands of commands per second, kcps) and latency for different partitions. Latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile).

titions.

Notice that only post operations are subject to this tradeoff since they may involve multiple partitions. The most common operation in social networks is the request to read a user timeline. This is a single-partition command in our application and as a consequence it scales linearly with the number of partitions.

Performance under dynamic workloads

Figure 5.6 depicts the performance of DynaStar and S-SMR* with an evolving social network. We started the system with the original network from Higg dataset. After 200 seconds, we introduced a new celebrity user in the workload. The celebrity user posted more frequently, and other users started following the celebrity.

At the beginning of the experiment, DynaStar performance was not as good as S-SMR* (i.e., lower throughput, higher number of percentage of multi-partition

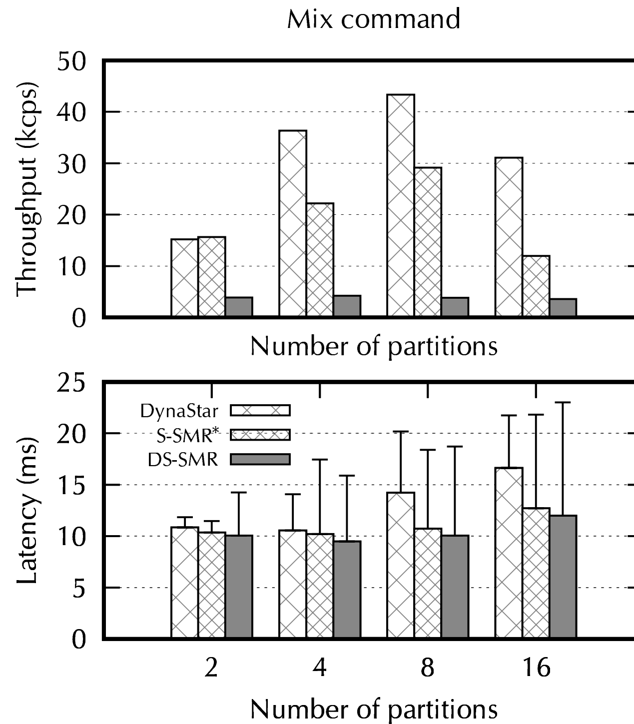

Figure 5.5. Performance of post command social network service. Throughput (in thousands of commands per second, kcps) and latency for different partitions. Latency for ≈75% peak throughput in milliseconds (bars show average, whiskers show 95-th percentile).

commands, and higher number of exchanged objects), because S-SMR* started with an optimized partitioning, while DynaStar started with a random partitioning. After 50 seconds, DynaStar triggered the repartitioning process, which led to an optimized location of data. Repartitioning helped reduce the percentage of multi-partition commands to 10%, and thus increased the throughput. After the repartitioning, DynaStar outperforms S-SMR* with the optimized partitioning. After 200 seconds, the network started to change its structure, as many users started to follow a celebrity, and created more edges between nodes in the graph. Both DynaStar and S-SMR suffered from the change, as the rate of multi-partition command increased, and the throughput decreased. However, when the repartitioning takes place in DynaStar, around 300 seconds into the execution, the previously user mapping got a better location from the oracle, which adapted the changes. After the repartitioning, the objects are moved to a better partition, with a resulting increase in throughput.

Figure 5.7 shows the cumulative distribution functions (CDFs) of latency for

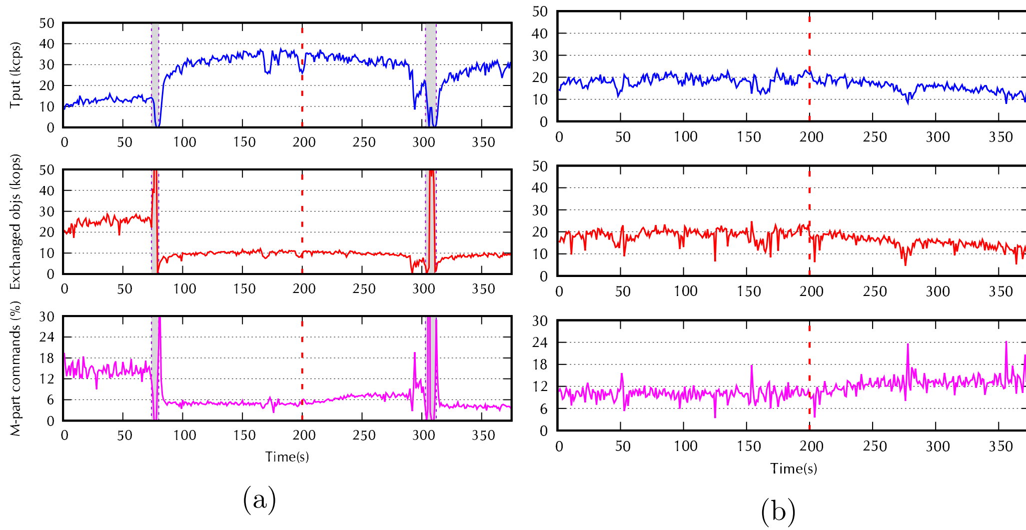

Figure 5.6. Repartitioning a dynamic workload with DynaStar (a) and S-SMR* without repartitioning (b).

the mix workload of DynaStar and S-SMR* on different configurations. The results suggest that S-SMR* achieves lower latency than DynaStar for 80% of the load. This is expected, as for multi-partition commands, partitions in DynaStar have to send additional data to return objects to their original location after command execution .

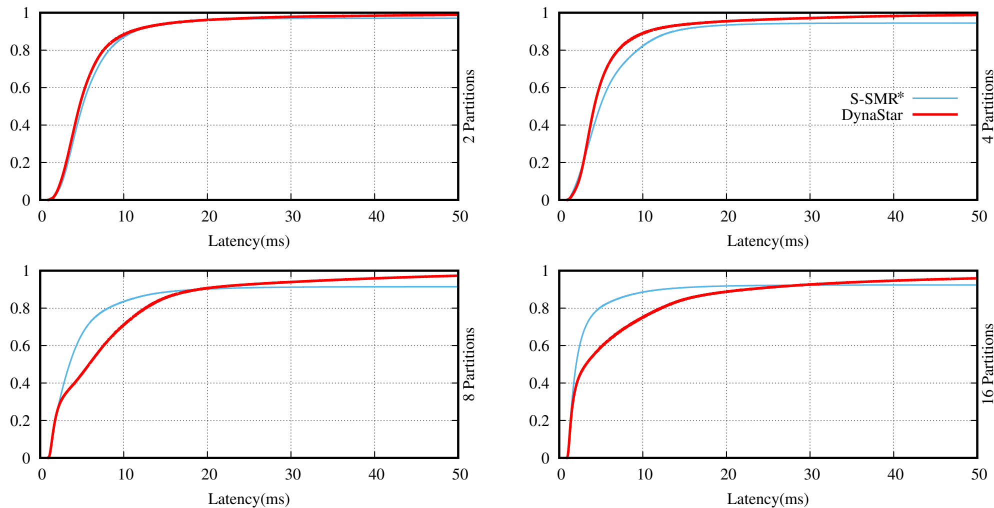

Figure 5.7. Cumulative distribution function (CDF) of latency for mix workloads on different partitioning configurations.

Table 5.1 shows the throughput of each partition when the DynaStar system reached the maximum throughput at the time 180. Although the objects were evenly distributed among partitions, there was still a skew in the load of the system, e.g., partition 1 and 2 served more commands than the other partitions. This happened because of the skew in the access pattern: some users were more active and posted more than the others, thus some servers receive more requests than the other servers.

Table 5.1. Average load at partitions at peak throughput.

Partition	Tput	M-part commands per sec	Exchanged objects per sec
1	12766	887	3907
2	11790	643	3036
3	6775	440	1503
4	6458	400	1490

5.6.5 The performance of the oracle

DynaStar uses an oracle that maintains a global view of the workload graph. The oracle allows DynaStar to make better choices about data movement, resulting in overall better throughput and lower latency. However, introducing a centralized component in a distributed system is always a cause for some skepticism, in case the component becomes a bottleneck, or a single point of failure. To cope with failures, the oracle is implemented as a replicated partition.

The oracle keeps a mapping of objects to partitions and the relations between objects. The size of the mapping depends on the complexity and the granularity of the graph. In the social network dataset, where each user is modeled as an object in the workload graph, the graph uses 1.5 GB of the oracle's memory. In the TPC-C experiments, only district and warehouse objects are in the workload graph; thus, the oracle only needs 1 MB of memory to store the graph for each warehouse.

We conducted experiments to evaluate if the DynaStar oracle is a bottleneck to system performance. The results show that the load on the oracle is low, suggesting that DynaStar scales well. The first experiment assesses the scalability of the METIS algorithm only. We measured the time to compute the partitioning solution, and the memory usage of the algorithms for increasingly large graphs. The results, depicted in Figure 5.8, show that METIS scales linearly in both memory and computation time on graphs of up to 10 million vertices.

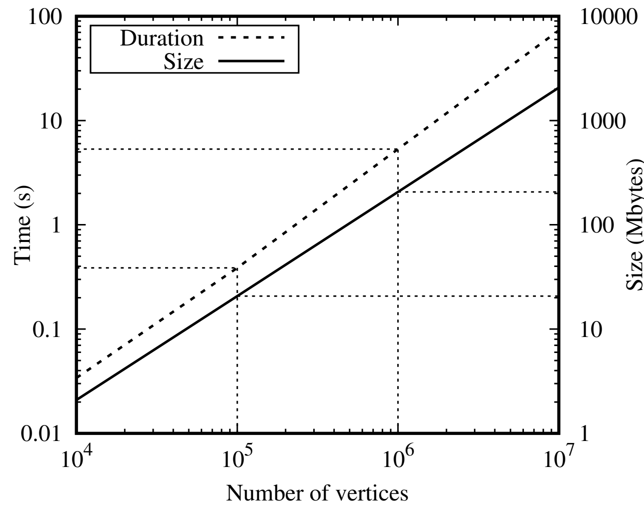

Figure 5.8. METIS processor and memory usage.

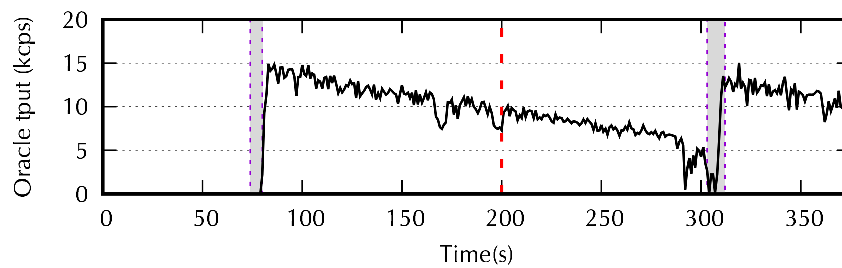

Figure 5.9. Queries sent to oracle in the social network service

The second experiment evaluates the oracle in terms of the number of queries sent to the oracle over time. The results shown in the bottom chart in the Figure 5.9 suggest that the oracle would not become a bottleneck. The number of queries processed to the oracle is zero at the beginning of the experiment, as the clients have cached the location of all objects. After 80 seconds, the repartitioning was triggered, making all cached data on clients invalid. Thus the throughput of queries at the oracle increases, when clients started asking for new location of variables. However, the load diminishes rapidly and gradually reduce. This is because access to the oracle is necessary only when clients have an invalid cache or when a repartition happens

5.7 Conclusion

In this chapter, we present DynaStar, a partitioning strategy for scalable state machine replication. DynaStar is inspired by DS-SMR, a decentralized dynamic scheme of DS-SMR. Differently from DS-SMR, however, DynaStar performs well in all workloads evaluated. When the state can be perfectly partitioned, DynaStar converges more quickly than DS-SMR; when partitioning cannot avoid cross-partition commands, it largely outperforms DS-SMR. The key insights of DynaStar are to build a workload graph on-the-fly and use an optimized partitioning of the workload graph, computed with an online graph partitioner, to decide how to efficiently move state variables. The chapter describes how one can turn this conceptually simple idea into a system that sports performance close to an optimized (but impractical) scalable system.

Chapter 6

Related work

In this chapter, we review some selected publications in the research areas considered by this dissertation, specially in the context of state machine replication and scaling the performance of state machine replication.

6.1 State machine replication

State machine replication (SMR) was first introduced by Lamport in [47]. Schneider [66] then presented a more systematic approach to the design and implementation of SMR protocols. Since then, SMR has become a well-known approach to replication and has been extensively studied, both in academia (e.g., [41; 46; 64; 53]) and in the industry (e.g., [22]). SMR provides strong consistency guarantees, which come from total order and deterministic execution of commands. Traditional SMR relies on a consensus protocol to define a common order among replicas for the execution of commands. Deterministic execution is usually ensured by having every replica execute the set of ordered commands sequentially.

6.2 Consensus and state machine replication

Consensus [47; 60] is a problem that requires one or more processes to cooperate, each with an initial value, to eventually agree on a single value. It was proved that in any asynchronous system with faulty processors, there is no deterministic algorithm providing termination [28]. One solution to ensure that processes make progress is to augment the asynchronous system with failure detectors [2]. Chandra and Toueg [17] propose a class of algorithms that use failure detectors

to solve consensus. Another solution is to introduce synchrony and assume a known delay to the system's communication [4].

Traditional consensus-based SMR repeatedly runs multiple instances of a consensus protocol to allow replicas to reach an agreed order of commands. The best-known consensus algorithm is the Paxos protocol by Lamport [48]. However, Paxos's description leaves many open design questions. Several works have argued that Paxos is not an easy algorithm to implement [16; 44]. Raft [59], a Paxos alternative, also implements consensus-based SMR, and it is suggested to be easier to understand (than Paxos) from an engineering point of view. Despite its complexity, several systems have been using Paxos to provide various abstractions such as storage systems [12], locking services [15], and distributed databases [6]. Google Chubby [15] and Google Spanner [22] employ an implementation of the Paxos algorithm to achieve consensus and also to cater for replication. Zap [38], the atomic broadcast protocol at the core of ZooKeeper, is a modified version of Paxos with a focus on high performance.

6.3 Scaling state machine replication

Even though increasing the performance of state machine replication is nontrivial, different techniques have been proposed for achieving scalable systems, such as partitioning the application state, parallelizing execution of commands, weakening consistency or optimizing the propagation and ordering of commands.

Partitioning application state

Partitioning the state of a replicated service is conceptually similar to partial replication of databases. Efforts to make linearizable systems scalable have been made in the past (e.g., [9; 22; 30; 56]). In Scatter [30], the authors propose a scalable key-value store based on DHTs, ensuring linearizability, but only for requests that access the same key. In the work of Marandi et al. [56], a variant of SMR is proposed in which data items are partitioned, but commands have to be totally ordered. M^2Paxos [61] proposes a scheme where leases are used instead of partitions owning objects, but assumes full state replication. S-SMR [9] ensures consistency across partitions without any assumptions about clock synchronization, but relies on a static partitioning of the state. P-Store [65] is a genuine partial replication protocol for wide-area networks. P-Store divides replicas into groups and partitiones data between the groups to ensure that all replicas in the same group replicate the same data items. When a transaction wants to commit,

it sends a message to all involved replicas and it is then validated. If a transaction is local to the group, each replica can individually decide whether to commit or abort the transaction; otherwise, if the transaction is global, all the involved replicas exchange votes in order to decide the outcome of the transaction.

Optimizing ordering protocol

Several works have focused on scaling performance of state machine replication by improving the performance of the ordering protocol. This allows the ordering layer (i.e., the underlying atomic broadcast algorithm) to be itself also scalable. For instance, Kapritsos and Junqueira [40] propose to divide the ordering of commands between different clusters: each cluster orders only some requests, and then forwards the partial order to every server replica, which then merges the partial orders deterministically into a single total order that is consistent across the system. In S-Paxos [10], Paxos [48] 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. Mencius [53] proposes a rotating leader protocol designed for wide-area networks, which improves throughput by distributing over multiple replicas the load that is usually concentrated on the leader. Ring Paxos [55] focus instead on achieving high network efficiency in fast local-area networks.

Parallelizing the execution of commands

Multi-threaded execution is a potential source of non-determinism, depending on how threads are scheduled to be executed in the operating system. However, some works have proposed multi-threaded implementations of state machine replication, circumventing the non-determinism caused by concurrency in some way. In [64], 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. In CBASE [46], 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 Eve [41], commands are tentatively executed in parallel. After the parallel execution, replicas verify whether they reached a consistent state; if not, commands are rolled back and re-executed sequentially. Storyboard [39]

was introduced as an approach that supports deterministic execution in multi-threading environments. Storyboard uses a forecasting mechanism to predict an ordered sequence of locks across replicas based on application-specific knowledge. Guo et al. [32] proposed Rex, a replicated state-machine framework for a multi-core system that uses an execute-agree-follow strategy. In Rex, a so-called primary server receives requests and processes those requests deterministically. The executions of requests could be in parallel. Rex uses locks to synchronize the concurrent access to a shared variable. The primary server periodically proposes the trace for agreement to the other replicas to update.

Weakening consistency guarantees

Many replication schemes aim at achieving high throughput by relaxing consistency; that is, they do not ensure linearizability. In deferred-update replication [19; 45; 67; 72], replicas commit read-only transactions immediately, not always synchronized with each other. Although this indeed improves performance, it allows non-linearizable executions. Database systems usually ensure serializability [7] or snapshot isolation [51], which do not take into account real-time precedence of different commands among different clients. For some applications, these consistency levels may be enough, allowing the system to scale better, but services that require linearizability cannot be implemented with such techniques. Some applications do not require strong consistency for most of their operations, and can weaken the consistency guarantees, for example, to eventual consistency (e.g., Facebook's TAO [14]). Eventual consistency [33] means that the correct replicas will eventually converge to a common state, even though some intermediate states might diverge to a certain extent. Some other systems try to combine the benefits of weak and strong consistency models by supporting both. In Gemini [50], transactions that can execute under weak consistency run fast without the need to coordinate with other datacenters. PNUTS [21] and DynamoDB [71] also combine weak consistency with per-object strong consistency. Both works rely on conditional writes, where a write fails in the presence of concurrent writes.

6.4 Graph partitioning in performance scaling

Graph partitioning is an interesting problem with many proposed solutions [1; 35; 43; 75]. In this work, we do not introduce a new graph partitioning solution, but instead, we use a well-known one (METIS [1]) to partition the state

of a service implemented with state machine replication. Similarly to DynaStar, Schism [23] and Clay [68] also use graph-based partitioning to decide where to place data items in a transactional database. In either case, not much detail is given about how to handle repartitioning dynamically without violating consistency. Turcu et al. [75] proposed a technique that reduces the amount of cross-partition commands and implements advanced transaction routing. Sword [62] is another graph-based dynamic repartitioning technique. It uses a hyper-graph partitioning algorithm to distribute rows of tables in a relational database across database shards. Sword does not ensure linearizability, and it is not clear how it implements repartitioning without violating consistency. E-Store [73] is yet another repartitioning proposal for transactional databases. It repartitions data according to access patterns from the workload. It strives to minimize the number of multi-partition accesses and is able to redistribute data items among partitions during execution. 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. DynaStar is a more general approach that works with any kind of relationship between data items, while also ensuring linearizability.

Chapter 7

Conclusion

With the explosive growth of the Internet, everyday online activities are increasingly dependent on the performance and reliability of large-scale distributed systems, such as e-banking, social networks, and e-commerce platforms. Over the years, state machine replication has become the standard approach to providing highly available stateful services. By replicating deterministic services across a number of servers, the replicated service is available as long as the total number of failures does not exceed a certain threshold. The strong consistency guarantee of state machine replication hides the complexity of data replication among independent replicas, and makes the underlying designs simpler. However, scaling such a replicated system while preserving the strong consistency guarantee is not trivial, since each replica must execute every command. To address this problem, several systems have investigated the use of state partitioning in the context of SMR, allowing client commands to be executed on a subset of replicas.

In this dissertation, we have explored the space of highly available and scalable systems from a performance perspective and proposed new solutions to improve efficiency. The contribution of this thesis is centered on scaling performance of a replicated state machine system. First, we presented a general comparison of several approaches to scaling the performance of a partitioned replicated system in the distributed system and database communities, by using an abstract framework for coordination. Second, we introduced 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. Third, we present DynaStar, a partitioning strategy for scalable state machine replication. Different from DS-SMR, DynaStar performs well in all workloads

evaluated. When the state can be perfectly partitioned, DynaStar converges more quickly than DS-SMR; when partitioning cannot avoid cross-partition commands, it largely outperforms DS-SMR. The key insights of DynaStar are to build a workload graph on-the-fly and use an optimized partitioning of the workload graph, computed with an online graph partitioner, to decide how to efficiently move state variables.

7.1 Research assessment

This dissertation presents three contributions: (i) a generic framework for scaling partitioned replicated system, (ii) a dynamic partitioning scheme for SMR (DS-SMR), and (iii) an optimized partitioning scheme for SMR (DynaStar). In the following, we review and discuss the most salient aspects of these contributions.

Survey on scalable partitioned replicated systems. Scaling a replicated system has been a topic of interest for many years. The service is replicated on a number of servers, to tolerate a certain degree of failure. Partitioning (or sharding) is a technique that divides the state of a service in multiple partitions, so the load could be equally distributed among partitions. A partitioned replicated protocol can be described using five generic phases: (i) the client submits a request to the system, (ii) the replicas of the involved partitions coordinate with each other to synchronize the execution of the request, (iii) the request is executed, (iv) the involved partitions agree on the result of the execution, and (v) the system sends the outcome of the request to the client. Some approaches may skip or simplify some phases: Google Spanner [22] uses two-phase commit and two-phase locking for concurrency control and does not need to order the transactions across partitions; Calvin [74], on the other hand, orders the transactions in a global log, deterministically executes transactions on all involved replicas, and removes the overhead of coordinating after the execution phase.

Dynamic partitioning for state machine replication. Classic SMR provides configurable fault tolerance and strong consistency but limited performance scalability, since all replicated nodes must execute the same sequence of commands. Some recent proposals have extended state machine replication with sharding. Essentially, these approaches partition (shard) the service state and replicate each partition. Commands that access state in a single partition are handled as in classic state machine replication.DS-SMR is the technique that allows a partitioned

SMR system to dynamically reconfigure its data placement on-the-fly. DS-SMR repartitions the service state dynamically, based on the workload. When a command needs variables from different partitions, those variables are first moved to the same partition. Then, the command is executed as a single-partition command. To reduce the negative effects of skewed load among partitions, the destination partition is chosen randomly. As a result, variables that are usually accessed together will tend to stay in the same partition, significantly improving scalability. We evaluate the performance of DS-SMR with a scalable social network application. The results demonstrate that DS-SMR could provide a scalable performance under workloads that exhibit strong locality.

Optimized partitioning for state machine replication. Sharding and replication are the mechanisms of choice of most scalable and fault-tolerant distributed systems. The performance of a partitioned system, however, heavily depends on the partitioning of the data: in order to scale, most commands must involve a single shard, and load must be balanced across shards. Estimating a good partitioning of the application state is challenging since it requires a priori information about the workload. Moreover, even if such information is available, access patterns may change during system execution. A good partitioning of the data for uniform access patterns may lead to poor performance under skewed access patterns. DynaStar is a partitioning strategy for scalable state machine replication. Differently from DS-SMR, DynaStar performs well in workloads that exhibit strong and weak locality. In the presence of strong locality, it converges more quickly than DS-SMR; in the presence of weak locality, it largely outperforms DS-SMR. The key insights of DynaStar are to build a workload graph on-the-fly and use an optimized partitioning of the workload graph, computed with an online graph partitioner, to decide how to efficiently move state variables. We describe DynaStar design and implementation, and presents a detailed performance evaluation using two benchmarks, a social network based on real data and TPC-C.

7.2 Future directions

The main objectives of this dissertation are to explore the space of techniques for scalable, strongly consistent replicated systems. However, many questions remain open, so we point here at possible research directions.

Remote updating objects. DynaStar and DS-SMR have similar execution models: moving objects that are required by a command to the same partition and executing the command against that partition. DynaStar has an extra step of returning back updated objects to their original locations to preserve the optimized partitioning. The operations of moving objects back and forth impose significant overhead on the performance of the system. One direction to remove this overhead is to let partitions update the objects remotely by making use of remote direct memory access (RDMA) [63; 26]. Essentially, RDMA is an approach that allows a host to access the memory of another host without involving the processor at the remote host. RDMA enables zero-copy transfers, low-latency communication and reduces CPU overhead by bypassing the OS kernel and by implementing several layers of the network stack in hardware. DynaStar powered by RDMA could allow partitions to read or write an object of a remote partition, without the need to relocate the object. However, handling the synchronization of the concurrent remote accesses is not trivial. It is worthwhile to study synchronization techniques and propose new and efficient protocols that can combine DynaStar with RDMA.

Optimal scalable atomic multicast protocol. In order to provide scalable performance, DS-SMR and DynaStar make use of state partitioning. The requests from the client are only sent to the replicas of the partitions that store the data being accessed. Traditional SMR relies on total message ordering, but in the case of DS-SMR and DynaStar, this may become a bottleneck. To address this problem, DS-SMR and DynaStar do not totally order the requests but use a partial order multicast primitive instead, which avoids enforcing a precedence relation between messages that do not share the same destinations. This allows the ordering layer (i.e., the atomic multicast protocol) to be itself also scalable. As one of the directions for expanding this study, a scalable atomic multicast protocol that has optimal latency, while providing high throughput, would be the ideal primitive for our system to build on.

Incremental partitioning. Our default implementation of DynaStar uses METIS [1] to provide a partitioning based on the workload graph. Although METIS does not necessarily produce the best possible partitioning of the workload graph, it offers a good compromise between performance and partitioning quality. However, METIS needs to repartition graphs from scratch every time a change is introduced in the graph or the number of partitions. This could result in having DynaStar relocating the whole graph [68]. Techniques that provide incremental

graph partitioning are orthogonal to our paradigm but could be used to further improve DynaStar performance.

Decentralized partitioning. Even though defining optimized partitioning by using a centralized component (i.e., the oracle) shows its advantages over the decentralized approach, the drawback of this approach is the scalability as the oracle is still prone to becoming a bottleneck in some cases. This may happen if (i) the workload becomes too big to be stored in a single machine, or (ii) the workload is very dynamic (i.e., the access patterns change frequently), which reduces the efficiency of the client cache and increases the queries to the oracle. As another direction for improving this work, it is worthwhile to come up with a decentralized graph partitioning and mapping that could help to remove the centralized component.

Bibliography

[1] Abou-Rjeili, A. and Karypis, G. [2006]. Multilevel algorithms for partitioning power-law graphs, *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE, pp. 10–pp.

[2] Aguilera, M., Chen, W. and Toueg, S. [2000]. Failure detection and consensus in the crash-recovery model, *Distributed Computing* **13**: 99–125.

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

[4] Aspnes, J. [2003]. Randomized protocols for asynchronous consensus, *Distributed Computing* **16**(2-3): 165–175.

[5] AWSTeam [2011]. Summary of the amazon ec2 and amazon rds service disruption in the us east region, `https://aws.amazon.com/message/65648/`. [Online; Accessed: 2020-02-20].

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

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

[8] Bezerra, C. E., Cason, D. and Pedone, F. [2015]. Ridge: high-throughput, low-latency atomic multicast, *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, IEEE, pp. 256–265.

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

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

[11] Birman, K. P. and Joseph, T. A. [1987]. Reliable communication in the presence of failures, *ACM Transactions on Computer Systems (TOCS)* **5**(1): 47–76.

[12] Bolosky, W. J., Bradshaw, D., Haagens, R. B., Kusters, N. P. and Li, P. [2011]. Paxos replicated state machines as the basis of a high-performance data store, *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDIâĂŹ11, USENIX Association, USA, p. 141âĂŞ154.

[13] Brodkin, J. [2012]. Hurricane sandy takes data centers offline with flooding, power outages, `https://bit.ly/2SXuLb1`. [Online; Accessed: 2020-02-20].

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

[15] Burrows, M. [2006]. The chubby lock service for loosely-coupled distributed systems, *Proceedings of the 7th symposium on Operating systems design and implementation*, pp. 335–350.

[16] Chandra, T. D., Griesemer, R. and Redstone, J. [2007]. Paxos made live: An engineering perspective, *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC âĂŹ07, Association for Computing Machinery, New York, NY, USA, p. 398âĂŞ407.

[17] Chandra, T. D. and Toueg, S. [1996]. Unreliable failure detectors for reliable distributed systems, *Journal of the ACM* **43**(2): 225–267.

[18] Charron-Bost, B., Pedone, F. and Schiper, A. [2010]. Replication, *LNCS* **5959**: 19–40.

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

[20] Coelho, P. R., Schiper, N. and Pedone, F. [2017]. Fast atomic multicast, *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, IEEE, pp. 37–48.

[21] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D. and Yerneni, R. [2008]. Pnuts: Yahoo!'s hosted data serving platform, *PVLDB* **1**: 1277–1288.

[22] Corbett, J. et al. [2013]. Spanner: Google's globally distributed database, *ACM Transactions on Computer Systems* **31**(3): 8:1–8:22.

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

[24] Défago, X., Schiper, A. and Urbán, P. [2004]. Total order broadcast and multicast algorithms: Taxonomy and survey, *ACM Comput. Surv.* **36**(4): 372–421.

[25] Denning, P. J. [2006]. The locality principle, *Communication Networks And Computer Systems: A Tribute to Professor Erol Gelenbe*, World Scientific, pp. 43–67.

[26] Dragojević, A., Narayanan, D., Castro, M. and Hodson, O. [2014]. Farm: Fast remote memory, *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pp. 401–414.

[27] Dwork, C., Lynch, N. and Stockmeyer, L. [1988]. Consensus in the presence of partial synchrony, *Journal of the ACM* **35**(2): 288–323.

[28] Fischer, M. J., Lynch, N. A. and Paterson, M. S. [1985]. Impossibility of distributed consensus with one faulty processor, *Journal of the ACM* **32**(2): 374–382.

[29] Ghemawat, S., Gobioff, H. and Leung, S.-T. [2003]. The google file system, *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pp. 29–43.

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

[31] Grolinger, K., Higashino, W. A., Tiwari, A. and Capretz, M. A. [2013]. Data management in cloud environments: NoSQL and NewSQL data stores, *Journal of Cloud Computing: Advances, Systems and Applications* **2**(1): 22.

[32] Guo, Z., Hong, C., Yang, M., Zhou, D., Zhou, L. and Zhuang, L. [2014]. Rex: Replication at the Speed of Multi-core, *EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems*, EuroSys.

[33] Gustavsson, S. and Andler, S. F. [2002]. Self-stabilization and eventual consistency in replicated real-time databases, *Proceedings of the first workshop on Self-healing systems*, pp. 105–107.

[34] Hadzilacos, V. and Toueg, S. [1994]. A modular approach to fault-tolerant broadcasts and related problems, *Technical report*, Cornell University.

[35] Hendrickson, B. and Kolda, T. [2000]. Graph partitioning models for parallel computing, *Parallel Computing* .

[36] Herlihy, M. and Wing, J. [1990]. Linearizability: A correctness condition for concurrent objects, *ACM Transactions on Programming Languages and Systems* **12**: 463–.

[37] Hsu, W. W., Smith, A. J. and Young, H. C. [2001]. Characteristics of production database workloads and the tpc benchmarks, *IBM Systems Journal* **40**(3): 781–802.

[38] Hunt, P., Konar, M., Grid, Y., Junqueira, F., Reed, B. and Research, Y. [2010]. Zookeeper: Wait-free coordination for internet-scale systems, *ATC. USENIX* **8**.

[39] Kapitza, R., Schunter, M., Cachin, C., Stengel, K. and Distler, T. [2010]. Storyboard: optimistic deterministic multithreading.

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

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

[42] Karypis, G. and Kumar, V. [1998]. Multilevelk-way partitioning scheme for irregular graphs, *Journal of Parallel and Distributed computing* **48**(1): 96–129.

[43] Kernighan, B. W. and Lin, S. [1970]. An efficient heuristic procedure for partitioning graphs, *Bell system technical journal* .

[44] Kirsch, J. and Amir, Y. [2008]. Paxos for system builders: An overview, *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, LADIS âĂŹ08, Association for Computing Machinery, New York, NY, USA.

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

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

[47] Lamport, L. [1978]. Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**(7): 558–565.

[48] Lamport, L. [1998]. The part-time parliament, *ACM Transactions on Computer Systems* **16**(2): 133–169.

[49] Leskovec, J. and Krevl, A. [2014]. SNAP Datasets: Stanford large network dataset collection, `http://snap.stanford.edu/data`.

[50] Li, C., Porto, D., Clement, A., Gehrke, J., Preguiça, N. and Rodrigues, R. [2012]. Making geo-replicated systems fast as possible, consistent when necessary, *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDIâĂŹ12, USENIX Association, USA, p. 265âĂŞ278.

[51] Lin, Y., Kemme, B., Jiménez-Peris, R., Patiño-Martínez, M. and Armendáriz-Iñigo, J. [2009]. Snapshot isolation and integrity constraints in replicated databases, *ACM Transactions on Database Systems* **34**(2): 11:1–11:49.

[52] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A. and Zhou, L. [2004]. Boxwood: Abstractions as the foundation for storage infrastructure., *OSDI*, Vol. 4, pp. 8–8.

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

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

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

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

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

[58] Oki, B. M. and Liskov, B. H. [1988]. Viewstamped replication: A new primary copy method to support highly-available distributed systems, *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, pp. 8–17.

[59] Ongaro, D. and Ousterhout, J. [2014]. In search of an understandable consensus algorithm, *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, USENIX Association, Philadelphia, PA, pp. 305–319.

[60] Paprzycki, M. [2001]. Distributed computing: Fundamentals, simulations and advanced topics., *Scalable Computing: Practice and Experience* **4**.

[61] Peluso, S., Turcu, A., Palmieri, R., Losa, G. and Ravindran, B. [2016]. Making fast consensus generally faster, *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 156–167.

[62] Quamar, A., Kumar, K. A. and Deshpande, A. [2013]. Sword: Scalable workload-aware data placement for transactional workloads, *EDBT*.

[63] Recio, R., Metzler, B., Culley, P., Hilland, J. and Garcia, D. [2007]. A remote direct memory access protocol specification, *Technical report*, RFC 5040, October.

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

[65] Schiper, N., Sutra, P. and Pedone, F. [2010]. P-store: Genuine partial replication in wide area networks, *2010 29th IEEE Symposium on Reliable Distributed Systems*, IEEE, pp. 214–224.

[66] Schneider, F. B. [1990]. Implementing fault-tolerant services using the state machine approach: A tutorial, *ACM Computing Surveys* **22**(4): 299–319.

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

[68] Serafini, M., Taft, R., Elmore, A. J., Pavlo, A., Aboulnaga, A. and Stonebraker, M. [2016]. Clay: Fine-grained adaptive partitioning for general database schemas, *PVLDB* **10**(4): 445–456.

[69] Shankland, S. [2018]. Google spotlights data center inner workings, `https://www.cnet.com/news/google-spotlights-data-center-inner-workings/`. [Online; Accessed: 2020-02-20].

[70] Shvachko, K., Kuang, H., Radia, S. and Chansler, R. [2010]. The hadoop distributed file system, *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, Ieee, pp. 1–10.

[71] Sivasubramanian, S. [2012]. Amazon dynamodb: a seamlessly scalable non-relational database service, *Proceedings of the 2012 International Conference on Management of Data. SIGMOD'12* .

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

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

[74] Thomson, A., Diamond, T., Weng, S.-C., Ren, K., Shao, P. and Abadi, D. J. [2012]. Calvin: Fast distributed transactions for partitioned database systems, *Proceedings of the 2012 ACM SIGMOD International Conference on*

Management of Data, SIGMOD âĂŹ12, Association for Computing Machinery, New York, NY, USA, p. 1âĂŞ12.

[75] Turcu, A., Palmieri, R., Ravindran, B. and Hirve, S. [2016]. Automated data partitioning for highly scalable and strongly consistent transactions, *IEEE Transactions on Parallel and Distributed Systems* **27**(1): 106–118.

[76] Van Renesse, R., Ho, C. and Schiper, N. [2012]. Byzantine chain replication, *International Conference On Principles Of Distributed Systems*, Springer, pp. 345–359.

[77] Van Renesse, R. and Schneider, F. B. [2004]. Chain replication for supporting high throughput and availability., *OSDI*, Vol. 4.

[78] Wiesmann, M., Pedone, F., Schiper, A., Kemme, B. and Alonso, G. [2000]. Understanding replication in databases and distributed systems, *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, IEEE, pp. 464–474.