Scalable State Machine Replication Revisited

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
Mojtaba Eslahi-Kelorazi

under the supervision of
Fernando Pedone

July 2022

Dissertation Committee

Laura Pozzi	Università della Svizzera Italiana, Switzerland
Patrick Thomas Eugster	Università della Svizzera Italiana, Switzerland
Paolo Romano	University of Lisbon, Portugal
Philippe Cudré-Mauroux	University of Fribourg, Switzerland

Dissertation accepted on 5 July 2022

Research Advisor	PhD Program Director
Fernando Pedone	**Walter Binder, Stefan Wolf**

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.

Mojtaba Eslahi-Kelorazi
Lugano, 5 July 2022

To my beloved family

Abstract

The requirements for availability, performance, and latency in today's online services are strict. State machine replication (SMR), a fundamental technique for increasing the availability of services without compromising consistency, offers configurable availability but limited scalability in terms of performance. Scalability in SMR is limited due to the fact that every replica has to execute the same set of requests, and therefore adding servers does not increase the maximum throughput. Scalable State Machine Replication (S-SMR) systems achieve scalable performance by partitioning the service state and coordinating the ordering and execution of commands to preserve the default consistency guarantee of SMR. While current S-SMR systems scale performance of single-partition requests with the number of partitions, replica coordination and object migration incur substantial overhead in the execution of multi-partition requests.

In this thesis, we first develop DynaTree, a distributed B+Tree algorithm over state-of-the-art S-SMR systems to study the implications of the partitioned SMR model on the development of complex distributed applications. We then look into improving performance and reducing latency of S-SMR systems. We leverage RDMA technology to enable systems with enhanced communication performance. RDMA provides the potential for high throughput and low latency communication by bypassing the kernel and implementing network stack layers in hardware. In this direction, we propose and implement two novel systems: (i) RamCast, the first genuine atomic multicast protocol tailor-made for shared-memory, and (ii) Heron, the first scalable state machine replication system on shared memory. RamCast leverages RDMA to reduce the latency of atomic multicast to microseconds by using RDMA mechanisms to protect memory from concurrent writes. Heron relies on RamCast to consistently order and deliver requests at partitions. It builds on RDMA's shared memory to coordinate and execute distributed operations while ensuring strong consistency. The performance evaluation of the proposed systems show substantial improvement in comparison to their message-passing variants.

Acknowledgements

I wish to express my gratitude to everyone who contributed to this thesis and the people who inspired me during this long journey.

First of all, I wish to express my genuine gratitude to my advisor Professor Fernando Pedone for his support during my PhD. I am extremely fortunate to have met him and learned some of the most valuable life lessons while working with him. I will always stay thankful for his exemplary dedication, enthusiasm, patience, and continuous encouragement. Much of this dissertation is the result of his endless support.

I am grateful to the dissertation committee members, Laura Pozzi, Patrick Thomas Eugster, Paolo Romano, and Philippe Cudré-Mauroux, for the time dedicated to my thesis and the helpful feedback. In particular, I would like to thank Paolo Romano for his hospitality at INESC-ID and the University of Lisbon. I want to express my gratitude to the entire faculty and staff members of the University of Lugano for their dedication and excellent work.

Being part of the Distributed Systems Lab has always been a privilege. I'm grateful for having the opportunity to work with all of my current and former colleagues: Paulo, Tu, Leandro, Enrique, Daniele, Pietro, Sam, Nenad, Elia, Daniel, Vahid, Theo, and Loan. In particular, I want to thank Long for all his help and contributions to this work, friendship, and fruitful discussions.

I want to thank my family, who have been there for me during this long journey. I wish to express all my gratitude to my wife Hanieh for her unconditional support and patience and for standing by my side during my PhD. If it weren't for your daily calls, distant compassion, and support, I'm not sure how I would deal with the distance and the challenges in my path.

My time in Lugano has been nothing short of spectacular, thanks to the people of Lugano. They have been very welcoming, and I have always encountered friendly and warm faces. Lugano will always have my heart for hosting some of the happiest moments in my life. Lastly, I wish to thank the Swiss Government for the financial support of this dissertation.

Preface

The result of this research appears in the following publications:

- M. Eslahi-Kelorazi, L.H. Le, F. Pedone, “Scalable State Machine Replication on Shared Memory”, Submitted to Middleware 2022
- L.H. Le, M. Eslahi-Kelorazi, P. Coelho, and F. Pedone, “RamCast: RDMA-based Atomic Multicast”, Middleware 2021
- M. Eslahi-Kelorazi, L. H. Le, and F. Pedone, “Developing Complex Data Structures over Partitioned State Machine Replication”, EDCC 2020
- L. H. Le, E. Fynn, M. Eslahi-Kelorazi, R. Soule and F. Pedone, “DynaStar: Optimized Dynamic Partitioning for Scalable State Machine Replication”, ICDCS 2019

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Context and goal of the thesis

Many modern online applications require performance scalability and high availability while operating at sub-millisecond latency to accommodate the most demanding services. Performance scalability ensures that by increasing system resources (e.g., servers), an application is able to accommodate additional client requests. High availability guarantees that the application will continue to function despite server failures and datacenter catastrophes.

Redundancy by replication can increase both scalability and availability. 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. Requests from users can be distributed across multiple replicas, dividing the workload across multiple machines and resulting in better scalability. Moreover, the failure of one replica does not result in an interruption of the service if there are multiple copies of the application running on several replicas. A main difficulty with replicating an application though is managing consistency among replicas. A strong consistency system provides programmers with an intuitive model for the effects of concurrent updates, reducing the likelihood of unexpected behavior. Designing latency-critical systems that combine scalability and fault tolerance while ensuring strong consistency is quite challenging.

State machine replication (SMR) is an established software replication technique, also known as active replication, to build highly available services [66; 92]. Essentially, the idea is to model the application as a state machine whose states are transitioned deterministically by executing client requests. Servers replicate the service state and solve a distributed problem known as consensus [38] to agree on the execution order of the requests submitted by the clients.

When replicas execute the same sequence of deterministic operations, they reach the same state and generate the same output. SMR aspires to behave just like a single server, with the required redundancy to make failures transparent to clients. SMR increases the availability of a service but does not improve its performance since each replica stores the complete service state and executes all the requests.

Scalable state machine replication (S-SMR) approaches propose to partition the service state (sharding) to scale performance and replicate each partition to tolerate failures [15; 27; 50; 101]. Requests are ordered consistently within partitions (e.g., with consensus) and across partitions (e.g., using an *ad hoc* ordering protocol [28; 101] or a communication abstraction like atomic multicast [47]). The system throughput (i.e., the number of requests that can be executed per unit of time) ideally increases linearly with the number of partitions if the service state can be partitioned such that requests access only one partition and are evenly distributed across partitions. The majority of applications, however, cannot be partitioned in such a way that requests always fall within a single partition. Therefore, S-SMR systems must cope with multi-partition requests, that are, requests that span multiple partitions. S-SMR systems must coordinate the execution of multi-partition requests in order to maintain the strong consistency guarantee of the system. Handling multiple-partition requests efficiently is a well-known challenge for such systems [51].

A golden rule in designing systems that can tolerate failures is that abstractions can significantly reduce complexity, and avoid design and programming errors. Atomic multicast is a group communication abstraction useful in the design of highly available and scalable systems. It allows messages to be addressed to a subset of the processes in the system reliably and consistently. Since messages can be multicast to different sets of destinations and interleave in non-obvious ways, implementing message order in a distributed setting is challenging. Some atomic multicast protocols address this challenge by ordering all messages using a fixed group of processes, regardless of the destination of the messages. To be efficient, however, an atomic multicast algorithm must be *genuine*: only the message sender and destination processes should communicate to propagate and order a multicast message [44]. A genuine atomic multicast is the foundation of scalable and fault-tolerant systems, since it does not depend on a fixed group of processes, and ensures reliable communication [25]. Current state-of-the-art genuine atomic multicast algorithms require several hundreds of microseconds to deliver requests for single-partition requests and 5x as much for multi-partition requests, which results in substantial delays in the delivery of requests for modern applications.

Message-passing communication has been used to develop practical distributed systems for years. Recent advances in shared memory technology, such as RDMA, have enabled systems to benefit from improved communication performance. RDMA provides the potential for high throughput and low latency communication by bypassing the kernel and implementing network stack layers in hardware. By bypassing the kernel and building the network stack layers directly into the hardware, RDMA offers the possibility of several microsecond latency communication. This allows building systems that serve latency-critical applications.

RDMA has been used to design high-performance SMR systems (e.g., [1; 87; 102]). Although servers can access remote memories without involving the host server's CPU, RDMA introduces the complication of race conditions, i.e., concurrent accesses to the same memory regions by different servers. RDMA-based SMR systems encounter this problem while ordering requests since once a request is ordered, each replica executes the request locally. Despite being fast, RDMA-based SMR systems do not scale performance similar to their message-passing counterparts. It is possible to apply partitioning in order to scale performance, although the solution requires careful deployment to guarantee that consistency is maintained. If not done right, consistency may be violated in requests that span multiple partitions.

This thesis addresses the introduced challenges related to SMR and S-SMR systems from different perspectives: it studies the implications of state partitioning on application design, presents the first shared memory atomic multicast algorithm that reduces the latency of multicast primitive to the level of microsecond, and proposes a shared-memory S-SMR system that substantially improves the performance of linearizable operations in comparison to message-passing systems.

1.2 Research contributions

In this section, we outline the main contributions of this thesis and provide a short description of each one. We defer detailed discussions to the next chapters.

Applications over scalable state machine replication: Some recent studies aim to increase the performance of S-SMR systems by optimizing data placement through dynamic repartitioning [50; 101]. In our first contribution, we investigate the challenges involved in developing complex applications over S-SMR systems. In particular, we develop DynaTree, a distributed B+tree algorithm that distributes tree nodes over a set of partitions, and each partition is replicated

for availability. The experiment results show that both read and update operations scale linearly with the number of partitions and insert operations do not lose performance with the growing number of partitions. The techniques used in DynaTree can be easily extended to other data structures and applications.

Shared-memory atomic multicast: Many atomic multicast algorithms have been designed for the message-passing system model over the years. RDMA's microsecond latency communication allows developing group communication primitives, such as atomic multicast, that targets latency-critical applications. In our second contribution, we have developed RamCast, the first atomic multicast protocol for the shared-memory system model. RamCast is designed by leveraging RDMA technology and by carefully combining techniques from message-passing and shared-memory systems. We show experimentally that RamCast outperforms current state-of-the-art atomic multicast protocols, increasing overall throughput and reducing latency to some microseconds.

Scalable state machine replication on shared memory: The last contribution of this thesis is devoted to the design and implementation of Heron, a scalable state machine replication system that delivers scalable throughput and microsecond latency. Heron achieves scalability through partitioning (sharding) and microsecond latency through a careful design that leverages one-sided RDMA primitives. Heron relies on RamCast to consistently order requests within and across partitions. It relies on RDMA's shared memory model to coordinate and execute distributed requests while ensuring strong consistency. Heron really shines when executing multi-partition requests, where objects in multiple partitions are accessed in a request, the Achilles heel of most partitioned systems. According to our findings, Heron improves the performance of executing complex workloads by one order of magnitude in comparison to state-of-the-art S-SMR systems and reduces the latency of coordinating linearizable executions to the level of microseconds.

1.3 Thesis outline

The rest of the thesis is organized as follows. Chapter 2 provides the foundations for the thesis, outlining the system model and definitions used throughout of the text. Chapter 3 presents DynaTree, a scalable and highly available B+tree algorithm over partitioned state machine replication. Chapter 4 discusses RamCast, an atomic multicast protocol designed specifically for the shared-memory architecture. Chapter 5 introduces Heron, the first scalable state machine repli-

cation system on shared memory. Chapter 6 surveys related work on the topics discussed in this thesis. Finally, Chapter 7 concludes the thesis by outlining our main findings.

Chapter 2

System model and definitions

In this chapter, we present the system model (§2.1), then define consensus (§2.2), atomic multicast (§2.3) and our correctness criterion (i.e., linearizability) (§2.4). The discussion is followed by introducing state machine replication (SMR) (§2.5), a scalable variant of SMR (§2.6), and Remote Direct Memory Access (RDMA) (§2.7).

2.1 System model

We make the following assumptions about processes, communication, failures, and system synchrony.

Processes. 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 $\mathscr{S}_0, ..., \mathscr{S}_k$ called partitions.

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).

Message-passing communication. Processes communicate through messaging, 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. Moreover, a message is received at most once, and only if it was previously sent. One-to-many communication relies on atomic multicast, defined in Section 2.3.

Shared-memory communication. Processes can also communicate by accessing portions of each other's memory. A process can share memory regions

with other processes and define permissions for those regions. Process q can read and write a register v in p's memory region mr with primitives $\mathrm{read}(p, v)$ and $\mathrm{write}(p, v, value)$, respectively. If both ends are correct, then every read and write operations are executed successfully. A permission associated with memory region mr defines disjoint sets of processes, R_{mr}, W_{mr}, and RW_{mr}, that can read, write, and read-write the registers in region mr. Process q has permission to read (respectively, write and read-write) v in p's mr if $q \in R_{mr}$ (resp., W_{mr}, RW_{mr}). A process can initially assign permissions for its shared memory regions and later change these permissions.

Synchrony model We consider a system that is *partially synchronous* [34]: 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. Protocols ensure safety under both asynchronous and synchronous execution periods. The partially synchronous assumption allows consensus defined in Section 2.2, to be implemented under realistic conditions [39; 67].

2.2 Consensus

Consensus is a fundamental coordination problem of distributed computing [66; 90]. 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 Atomic multicast and atomic broadcast

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. A process atomically multicasts a message m to a set of groups γ by invoking primitive a-mcast(γ, m). A process delivers m with primitive 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 $p \in g$, where $g \in \gamma$ eventually a-delivers m or no process in that group is correct *(validity)*.
- If a correct process $p \in g$, where $g \in \gamma$ a-delivers m, then every correct process $q \in h$, where $h \in \gamma$ eventually a-delivers m *(uniform agreement)*.
- For any message m, every process $p \in g$, where $g \in \gamma$ 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 g, where $g \in \gamma$ 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. Atomic multicast offers strong communication guarantees and should not be confused with network-level communication primitives (e.g., IP-multicast), which offer "best-effort" guarantees.

2.4 Consistency

An object that can be concurrently accessed by many processes is called a *concurrent object* [48]. Interleaving accesses to the same object can sometimes lead to unexpected behavior. 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* [48]. 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 $\mathscr{E}$ is linearizable if there is some permutation of the commands executed in $\mathscr{E}$ 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.

2.5 State machine replication

State machine replication, also called active replication, is a common approach to building fault-tolerant systems [66; 93]. 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 production 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 exter-

nal 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 [19; 29]. In this thesis, we consider implementations of state machine replication that ensure linearizability.

2.6 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 of 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., [76; 79; 84]), as opposed to dedicating a single process to determine the order of commands (e.g., [67]).

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 [45; 61; 64; 77]. Ideally, one could use a replication technique that supports parallelism (multithreading) 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., [40; 81]). 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. Long [70] has reviewed some approaches in the second category, which include Scalable State Machine Replication (S-SMR) [15], Google Spanner [27], and some other techniques.

2.7 Remote Direct Memory Access (RDMA)

Remote Direct Memory Access (RDMA) is a protocol that enables direct data access to the memory of a remote machine without involving the operating system and processor of the remote machine. RDMA implements the network stack in hardware, and provides both low latency and high bandwidth by bypassing the kernel and supporting zero-copy communication.

RDMA provides one-sided operations (e.g., read, write), two-sided operations (e.g., send, receive), and atomic operations (e.g., compare-and-swap). The two-sided operations rely on memory copies in user space and involve the CPU of the remote host. Thus, when compared to one-sided RDMA verbs, they introduce additional overhead [31]. Previous studies have established guidelines to use RDMA operations efficiently [58; 59; 82]. We restrain the use of read or write primitives depending on the use case with the goal of having simpler logic.

RDMA offers three transport modes which are Unreliable Datagram (UD), Unreliable Connection (UC), and Reliable Connection (RC). UD supports both one-to-one and one-to-many transmission without establishing connections, whereas UC and RC are connection-oriented and only support one-to-one data transmission. RC guarantees that the data transmission is reliable and correct in the network layer, while UC does not have such a guarantee. In this thesis, we employ RC to provide in-order and reliable delivery. The RDMA-enabled network card (RNIC) on each remote host creates a logical RDMA endpoint known as a Queue Pair (QP), which includes a send queue and a receive queue for storing data transfer requests, to establish a connection between two remote hosts. Operations are posted as Work Requests (WRs) to QPs to be served and executed by the RNIC. A completion event is pushed to a Completion Queue (CQ) when an RDMA operation is completed. By setting a flag in the WR, operations can be made unsignaled; these verbs do not create a completion event, and the application detects completion via application-specific methods. By asking the operating system to pin the memory pages that the RNIC would use, each host makes local Memory Regions (MR) available for remote access. Different access modes (i.e., read-only or read-write) can be set for QPs and MRs. The access mode can be specified when initializing the QP or registering the MR and it can be updated later. The host can register the same memory for various MRs.

Chapter 3

Applications atop partitioned SMR

State machine replication (SMR) is a classic approach to fault tolerance [67]. Partitioned state machine replication shares the basic characteristics of classic SMR, namely, requests are ordered and then deterministically executed. In order to achieve good performance, only the partitions involved in a request must order and execute the request. Consequently, the partitions involved in the execution of a request must be known *before* the request is ordered and executed. Prior work has considered classes of applications in which this information is readily available (e.g., key-value stores, file systems). In this chapter, we share our experiences with developing complex applications with partitioned state machine replication. By complex we mean applications in which the partitions involved in a request cannot be easily identified a priori.

This chapter introduces DynaTree, a scalable and highly available B+tree over partitioned state machine replication. B+tree is a self-balancing tree data structure that preserves sorting order and guarantees lower bounds for accessing data. DynaTree employs the scalable state machine replication model proposed by Le et al. [51]. This model partitions the application state and replicates each partition for availability. It maintains a location oracle with a global view of the application state to help clients locate them in partitions. By having the set of partitions involved in the execution of a requests, clients atomically multicast requests to the destination partitions where the requests get executed. For executing a request, the model ensures that all the states accessed in the request are gathered in one partition before execution. Atomic multicast ensures that requests are properly ordered across partitions [26].

The S-SMR model provides scalability and fault tolerance at the cost of additional requirements: clients must identify the data and partitions accessed in a request before the request is executed. This poses some challenges in the case

of complex data structures, such as B+trees. For example, to insert a key in the tree, a client must know in advance whether the insert will lead to a split and the nodes and partitions involved in the insertion. To satisfy this requirement, clients in DynaTree lazily cache inner nodes of the tree. Therefore, clients first traverse the cached tree to find the appropriate nodes and then issue the request to the involved partitions for execution. This scheme, however, introduces additional complications. Since the client cache may be stale, partitions must verify the validity of the cached information before executing a request. In case a partition finds that the client cache is stale, it informs the client to update the cache and re-try the request.

In addition to proposing a novel scalable and fault-tolerant distributed B+tree key-value store, we discuss the inherent challenges involved in designing a distributed data structure in partitioned state machine replication. We fully implemented DynaTree and assessed its performance extensively. Our results show that DynaTree scales read and update requests with the number of partitions, and outperforms a well-established database that relies on a B-tree, deployed in high-availability mode.

The reminder of the chapter is structured as follows. Section 3.1 introduces DynaTree and explains the challenges involved. Section 3.2 discusses implementation details and presents experimental results. Section 3.3 concludes the chapter.

3.1 General idea

DynaTree implements a distributed B+tree. The tree consists of a set of nodes with variable number of child nodes within some pre-defined range. Tree nodes may split into two half-size nodes to maintain the pre-defined range. In a B+tree, leaf nodes store key-value pairs and inner nodes store key-pointers. When a node is full, it is split into two half-size nodes. With the exception of the root node, which may contain a single key, all other nodes contain $k < n < 2k - 1$ keys, where n is the node size and k is the minimum size of a node [10].

DynaTree distributes tree nodes in a number of partitions with the goal of increasing performance. Yet, distributing tree nodes raises challenges in executing tree operations. Searching for a key, for instance, is not an obvious task while nodes are distributed. In the following, we demonstrate how DynaTree deal with executing operations while tree nodes are distributed.

DynaTree is composed of three main components: the client process, the server process, and the oracle process. Figure 3.1 shows different components

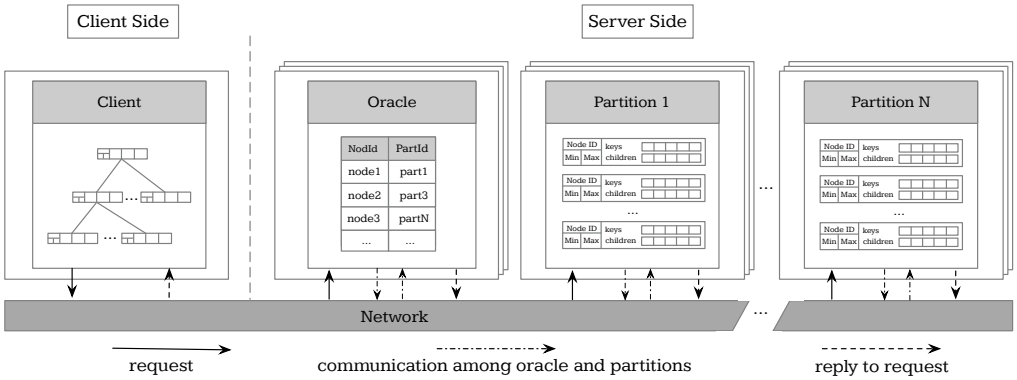

Figure 3.1. The client cache, the oracle map, and the distributed tree nodes in DynaTree.

and how they are connected.

The client provides an interface with search, update and insert operations. Clients cache internal B+tree nodes. Whether a client searches for a key or inserts a new key in the tree, it traverses the tree in its cache from the root down to the appropriate leaf. After finding the leaf, the client issues a request to the partitions to execute the request. While traversing the tree, the client either has the node in its cache or reads the required nodes from the partitions.

Each server process contains a subset of tree nodes. A partition receives client requests if it is in the destination of the request. A partition is in the destination of a request when the request involves nodes of the tree from that partition. The partitions deliver and execute requests respecting their realtime order so they contain the most updated version of the objects for each request execution to guarantee strong consistency (i.e., linearizability).

The oracle is responsible for storing a location map from tree nodes to partitions. The oracle plays the role of a directory to guide clients to find nodes in partitions. This allows clients to forward requests to the partitions that are involved in the execution of the request.

In the following, we first discuss client cache and its importance in the execution of requests. Then we discuss tree operations and explain the corresponding algorithms.

3.1.1 Client cache

In DynaTree, clients store internal tree nodes and thus are able to traverse the tree locally rather than reading nodes from the partitions each time. This cache

Algorithm 1 Data structures

1: The tree contains the following information:
2: $rootId$: The root node of the tree
3: MAX-SIZE: Maximum size of a node
4: $cache$: Client cache of internal nodes
5: $storage$: List of tree nodes that a partition owns
6: $childParent$: A map of nodes to their parents
7: $ancestorlist$: List of ancestors of the node
8: $numReservedObjIds$: Number of requested object Ids asked by request
9: $reservedObjIds$: The oracle reserves object Ids declared in numReservedObjIds and append to the request
10: $result$: Result of looking up a key in node; contains $value$ if the key is available
11: $RootSplit()$: Splits the root of the tree
12: $NodeSplit()$: Splits a tree node
13: Each tree node contains the following information:
14: nid: Node ID
15: $size$: Node size
16: $keys$: Keys stored in the node
17: $values$: Values associated with the keys
18: $children$: Pointer to child nodes (if any)
19: $isLeaf()$: Returns true if the node is leaf
20: $isRoot()$: Returns true if the node is root
21: $lookup(key)$: Looks up the key in node
22: $update(key, value)$: Updates the value of a key
23: $insert(key, value)$: Inserts the key/value pair
24: $split(parent, node)$: Splits and move half of keys into $node$; updates the $parent$ node
25: $isInFenceKeys(key)$: true if key is in node's fence keys

becomes invalid when the structure of the tree changes (e.g., when a node splits). Upon executing a request, the partition notifies the client to invalidate its cache and try again when the client's cache is not valid.

We use fence keys to validate client requests at partitions [72]. Fence keys are essentially two integers determining the range of the keys a node is responsible for, even though the node may not contain all keys in the range (i.e., a key never inserted). Fence keys optimize the cache invalidation by decreasing the number of unnecessary invalidations. While splitting nodes, their fence keys are changed. DynaTree guarantees continuous, non-overlapping ranges for the nodes in each tree level in the presence of concurrent node split requests.

A client reads the inner nodes from the partitions and stores them for traversing the tree. The lazy replicated nodes in the client cache may become stale. For example, when a client traverses its stale cache to find a key, it finds leaf T that should be looked up for the key. So, the client issues a request for searching node

Algorithm 2 Reading a node from cache or partitions

```
1:  GetNode(nid, parentid, key)                                  {Client side}
2:    if cache.contains(nid) then
3:      return cache.get(nid)                                    {read node from local cache}
4:    <nid, node> ← command(READ, nid)
5:    if node.isInFenceKeys(key) then
6:      cache.add(nid, node)                                     {add node to client's cache}
7:      childParent.add(nid, parentid)
8:      return node
9:    else
10:     parent ← childParent.get(nid)
11:     InvalidateNode(parent.nid)                               {cache invalidation}
12:     return null
13: InvalidateNode(nid)                                          {Client side}
14:   node ← cache.get(nid)
15:   for child in node.children do
16:     InvalidateNode(child)
17:   cache.remove(node)
18: Read(nid)                                                    {Server side}
19:   node ← storage.get(nid)
20:   return node
```

T for the provided key. The partition that receives the request first check if the request is valid. The request is valid if the key is in the fence keys of node T. If node T has split and the key is not in the range of its fence keys anymore, the request is not valid, and therefore, the partition asks the client to invalidate its cache and try again. Otherwise, the partition searches for the key in T and sends back the response to client.

Cache invalidation is optimized by invalidating one node at a time. When a request performing an operation on node T is asked to retry, the algorithm invalidates node T's parent, called T'. The retry asked by the partition implies that the node T' has changed. By reading the parent node again, the client will likely find the correct child next time. In case reading the parent node also results in a retry, the invalidation proceeds one level up. This can go up to the root node where the client clears its cache and starts reading the root node again.

Algorithm 2 shows the steps executed by clients to read a tree node. The GetNode method starts by searching the client's cache for a node. In case the client does not contain a node, it issues a request to read the node from the partitions. We invoke a DynaStar command by specifying the operation and the arguments passed to the command (line 3). The request is later processed by the

Algorithm 3 Searching for a key

```
 1: Search(key)                                                    {Client Side}
 2:   node ← SearchUtil(key)                                       {find the appropriate leaf}
 3:   <RESPONSE,value> ← command(SEARCH, node.nid, key)
 4:   if RESPONSE = SEARCHFOUND then
 5:     return value                                               {key is available in the tree}
 6:   else if RESPONSE = SEARCHNOTFOUND then
 7:     return null                                                {key is not available}
 8:   else if RESPONSE = SEARCHRETRY then
 9:     parent ← childParent.get(node.nid)
10:     InvalidateNode(parent)                                     {invalidate parent node}
11:     return Search(key)
12: SearchUtil(key)                                               {Client Side}
13:   node ← GetNode(rootId, null, key)                           {read the root node}
14:   while !node.isLeaf() do
15:     <nid, result, childid> ← node.lookup(key)
16:     node ← GetNode(childid, nid, key)
17:     if node = null then                                       {retry traversing the tree}
18:       return SearchUtil(key)
19:   return node
20: Search(nid, key)                                              {Server Side}
21:   node ← storage.get(nid)
22:   if !node.isInFenceKeys(key) then                            {check fence keys}
23:     return <SEARCHRETRY>
24:   result ← node.lookup(key)
25:   if result.isAvailable() then
26:     return <SEARCHFOUND, result.value>
27:   else
28:     return <SEARCHNOTFOUND>
```

server-side logic in the destination partition (18-20). Here, the partition simply returns the node asked by the request. The client checks if the node is valid. Otherwise, the client invalidates its cache and tries again (5-12). Algorithm 1 summarizes the fields and methods in the algorithms.

3.1.2 Search and update

Algorithm 3 presents the steps to search for a key. Updating the value of a node follows a similar execution path, where the client provides the new value for the node. The client starts searching for a key by reading the root node (line 13) and looks up this node to find the next node to read (15). It continues reading nodes and looking them up down to the leaf (14-18).

After finding the appropriate leaf, the client asks the corresponding partition to search for the key (3). The partition validates the request before execution (22-23). The validation is necessary since the client cache may be stale. The partition uses the leaf's fence keys to find out if the request is valid. If so, the partition is able to search the leaf and return the result to the client (25-28).

3.1.3 Insert

Algorithm 4 details the insert operation. The client starts the insert request by traversing the tree in its cache. This leads to a leaf which is the node for inserting the key. Next, the client issues a request for inserting the key (lines 4-5). When a partition delivers the request, it checks if the insertion is valid. In case the request is not valid due to stale client cache, the client is asked to invalidate its cache and try again (9-10). The client cache is stale when it asks to insert a key in a node out of the node's fence keys.

In case the insert operation is valid, the node is looked up to see if it already contains the key. In this case, the algorithm only needs to update the value of the key (12-14). If the node does not contain the key and the node is not full, the partition inserts the key in the tree (15-17). In the last case, where the node is full, the partition needs to split the node before inserting the key. Therefore, the partition notifies the client that the node is full (18-19). This case involves additional communication steps and is discussed in the next section.

3.1.4 Splitting nodes

In the B+tree algorithm, a tree node splits when its size hits the node's maximum size. While splitting, the node is divided into two nodes, each containing half of the key-value pairs. To split a node, the request needs to involve the node and its parent. The parent node is necessary because a separator key for the new node is inserted in the parent node.

Splitting a node is a more complex operation than the other operations. There are three main reasons for this complexity. First, splitting a node involves creating a new node. The oracle has to be aware of all objects in the system. So, the oracle has to be involved in the request. Second, the split operation can lead to further splits. When a node needs to split while its parent node is full, the operation needs to cascade the split up to the parent node. Third, clients may try to split a node concurrently. The algorithm needs to check if concurrent requests have changed the node before applying changes. In the following, we explain how we deal with these cases.

Algorithm 4 Inserting a key-value pair

```
1:  Insert(key, value)                                              {Client side}
2:    node ← SearchUtil(key)
3:    <RESPONSE, nid, key, value, ancestorlist> ←
4:        command(INSERT, node.nid, key, value)
5:    InsertRespone(RESPONSE,nid,key,value,ancestorlist)

6:  Insert(nid, key, value)                                         {Server side}
7:    node ← storage.get(nid)
8:    if !node.isInFenceKeys(key) then
9:      return <INSERTRETRY>
10:   result ← node.lookup(key)
11:   if result.isAvailable() then                                  {update the value}
12:     node.update(key, value)
13:     return <UPDATED>
14:   else if node.size < MAX-SIZE then                             {insert new key}
15:     node.insert(key, value)
16:     return <INSERTED>
17:   else if node.size = MAX-SIZE then                             {split needed}
18:     return <RETRYSPLIT,nid,key,value,null>

19: InsertResponse(RESPONSE, nid, key, value, ancestorlist)
20:   if RESPONSE ∈ {INSERTED,UPDATED} then                         {Client side}
21:     return true                                                 {success}
22:   else if RESPONSE = INSERTRETRY then                           {stale cache}
23:     parent ← childParent.get(nid)
24:     InvalidateNode(parent)
25:     return Insert(key, value)
26:   else if RESPONSE = RETRYSPLIT then                            {split needed}
27:     if ancestorlist.isEmpty() then
28:       parent ← childParent.get(nid)
29:       ancestorlist.add(parent)
30:       numReservedObjIds ← 1
31:     else if !ancestorlist.last().isRoot() then
32:       ancestor ← childParent.get(ancestorlist.last())
33:       ancestorlist.add(ancestor)
34:       numReservedObjIds ← 1
35:     else
36:       numReservedObjIds ← 2
37:     command.allocate(numReservedObjIds)
38:     <RESPONSE, nid, key, value, ancestorlist> ←
39:         command(INSERTSPLIT,nid,key,value,ancestorlist)
40:     InsertRespone(RESPONSE,nid,key,value,ancestorlist)
```

Algorithm 5 Splitting a node

```
1:  Split (nid, key, value, ancestorlist)                          {Server side}
2:    node ← storage.get(nid)
3:    if !node.isInFenceKeys(key) then
4:      invalidNode ← node.nid
5:      return <SPLITRETRY, invalidNode>
6:    result ← node.lookup(key)
7:    if result.isAvailable() then
8:      node.update(key, value)                                    {update the value}
9:      return <UPDATED>
10:   if node.size < MAX-SIZE then
11:     node.insert(key, value)                                    {insert the new key-value pair}
12:     return <INSERTED>
13:   for i=ancestorlist.size−1; i>=0; i−− do
14:     ancestor ← ancestorlist[i]
15:     if !ancestor.isInFenceKeys(key) then
16:       invalidNode ← ancestor                                   {return the invalid node}
17:       return <SPLITRETRY, invalidNode>
18:   for i=0; i<ancestorlist.size; i++ do
19:     ancestor ← ancestorlist[i]                                 {check if ancestors need split}
20:     if ancestor.size < MAX-SIZE then
21:       for j=ancestorlist.size−1; j>i; j−− do
22:         ancestorlist.remove(j)
23:   reservedObjIds ← command.getReservedObjIds()
24:   lastNode ← nodes.get(ancestorlist.last())
25:   if lastNode.size < MAX-SIZE or lastNode.isRoot() then
26:     if lastNode.isRoot then
27:       RootSplit(lastNode, reservedObjIds)
28:     pos ← ancestorlist.size−1
29:     for pos; pos>=0; pos−− do
30:       NodeSplit(pos, node, ancestorlist, reservedObjIds)
31:     node.insert(key, value)                                    {insert the new key-value}
32:     return <INSERTED>
33:   else                                                         {upmost parent is full}
34:     return <RETRYSPLIT,nid,key,value,ancestorlist>
```

The split operation starts when a client receives RETRYSPLIT as the result of an insert request (lines 38-40 in Algorithm 4). The client starts a new request by adding the node id, the key and the value. There is one more field appended to the arguments list called ancestorlist, which contains the ancestors of the node. Ancestor nodes are added to the list one at a time. When a partition asks the client to retry an insertion with split, the client adds its parent node to the list. If the parent node is also full and needs to split, the client retries by adding the parent of the parent node to the list. The list contains all ancestors of the leaf when the root needs to split.

The Split operation is shown in Algorithm 5. The algorithm starts by checking whether the request is valid (lines 3-5). Next, it checks whether the key has been inserted concurrently. In this case, it is enough to update the key's value (6-9). The node might split with a request from another client. In this case, the key can be inserted into the node without an additional split (10-12).

For the other cases, we ensure that modifications to the tree deal with concurrent requests to maintain a consistent tree. Starting from line 13, the algorithm verifies that the insertion is valid. Next, the algorithm verifies whether ancestor nodes need to split. We clarify this verification with an example. Assume that there are two splits needed before insertion, which means that there are two items in the ancestorlist. However, it is possible that the immediate parent of the node has split with concurrent requests from a different client. In this case, we avoid splitting the parent node again.

In the last step, the algorithm splits the nodes. The invocation of the split method finishes by moving half of the key-value pairs to the new node and by adding the first element of the new node to the parent node. Now, the algorithm is able to insert the key in the leaf node (31-32).

3.2 Evaluation

In this section, we evaluate the performance of DynaTree. In particular, we investigate the scalability of tree operations with the growing number of partitions. Our prototype is written in Java. The source code is publicly available.[1]

Experimental environment We conducted all experiments on a cluster with two types of nodes: (a) Forty nodes (HP SE1102), equipped with two Intel Xeon L5420 processors running at 2.5 GHz and with 8 GB of main memory, and (b)

[1] https://github.com/meslahik/dynatree

Forty eight nodes (Dell SC1435), 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.

3.2.1 BerkeleyDB High Availability

BerkeleyDB [85] is a well-known embedded key-value store that provides high performance data management service to applications. BerkeleyDB Java Edition is a native Java implementation that we use in our evaluation as a baseline. BerkeleyDB JE uses a B-tree as its underlying data structure. We use BerkeleyDB with high availability mode enabled. High availability mode adds replication with master-slave model. It provides fault tolerance and increases the performance of BerkeleyDB under certain workloads. All changes in data have to be performed by the single read-write replica and then propagated to the read-only replicas. We configure the read-write replica to wait for acknowledgements from all read-only replicas to ensure strong consistency (i.e., linearizability) and make it comparable to the strong properties offered by DynaTree.

3.2.2 Workloads

In all experiments, unless stated otherwise, the tree is initially populated with 100K key-values pairs. The tree node's minimum size is 100 keys. This results in a initial tree of height 4. The tree is configured to maintain three replicas per partition. Keys and values are 4-byte integers and taken from integer's positive range, chosen randomly using an uniform distribution. Each experiment lasts two minutes; we ignore the first and last 15 seconds. We report peak throughput, achieved by increasing the number of clients to saturate the system, and contention-free latency, by configuring the experiments with a single client. In both setups, DynaTree and BerkeleyDB, we configured the system so that there is enough memory to keep all data in memory.

3.2.3 Node size

We assess the effect of tree node size on the peak throughput. There is a performance tradeoff involving the tree node size. A large node results in fewer splits, which can improve the throughput of insertions. However, a large node

Node min size	10	50	100	200	1000
Throughput (cps)	7858	19077	31443	20108	15866

Table 3.1. Peak throughput versus tree node size

leads to expensive serialization and deserialization costs when nodes are moved from one partition to another. The nodes are moved between partitions while splitting a node or when a client fetches an inner node while traversing the tree. The results reveal the tradeoff between the node size and the throughput. Table 3.1 shows the peak throughput for 8 partitions versus tree node minimum size with insert workload. Motivated by these results, we have configured all other experiments with a tree node minimum size equal to 100.

3.2.4 Search scalability

In this experiment, we investigate the ability of DynaTree to scale with the increasing number of partitions. Figure 3.3 shows the throughput and latency of DynaTree and the BerkeleyDB-HA for 1 to 16 partitions, the maximum number of partitions we can accommodate in our experimental environment.

Since BerkeleyDB is fully replicated, for a workload with only search requests, replicas can individually respond the requests. Therefore, BerkeleyDB scales almost linearly with the number of replicas for search requests. DynaTree also scales linearly for search requests, though it incurs higher overhead than BerkeleyDB, due to ordering requests before execution, which explains its lower peak throughput than BerkeleyDB's. Moreover, since BerkeleyDB replicas have a full copy of the data, the execution of a search in BerkeleyDB is local to a replica with no additional roundtrips to fetch tree nodes by clients, which results in lower latency than DynaTree.

These performance advantages come with a cost though. In the BerkeleyDB setup, each replica must contain the entire data set. In particular, for the configuration with 16 partitions, BerkeleyDB demands 16 times as much memory in each partition/replica as DynaTree. In case servers do not have enough resources to keep all data in memory, they will rely on expensive I/O to read data from disk. DynaTree does not suffer from this drawback as it partitions the data.

3.2.5 Update scalability

We now examine the throughput under an update-intensive workload. In the beginning of each experiment, 100K keys, chosen randomly from a uniform dis-

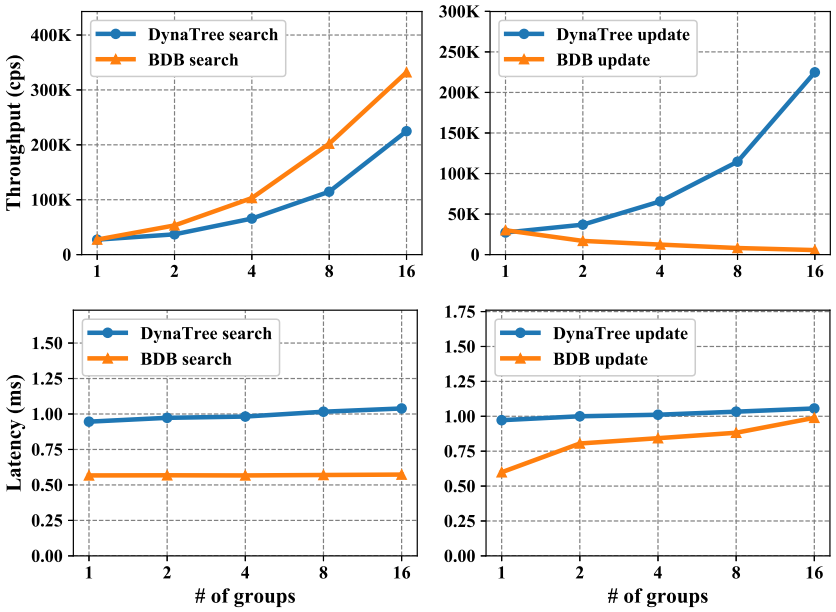

Figure 3.2. Peak throughput and contention-free latency for search and update workloads in DynaTree and BrekeleyDB.

tribution, have been inserted in the tree. Then, clients in a closed loop choose a key from the inserted keys and update their value. The number of keys show the system behavior in the presence of contention.

BerkeleyDB scales poorly with updates as each update involves all replicas. In DynaTree, however, updates scale linearly with the number of partitions. This is due to the fact that updates do not change the structure of the tree. Thus, updates are single-partition requests to update the key's value.

The latency graph in Figure 3.3 shows how the two systems behave when the number of partitions increases. The difference between the latency of requests in one partition/replica configuration is due to the fact that in DynaTree, the requests pass the ordering process. This implies some delays even for requests submitted to one partition. However the latency does not increase with the number of partitions since update operation is a single-partition request. In BerkeleyDB, the latency increases when there are more replicas. The master replica has to wait for acknowledgements from all replicas. Therefore, the more replicas in the system, the more master has to wait for the completion of updates.

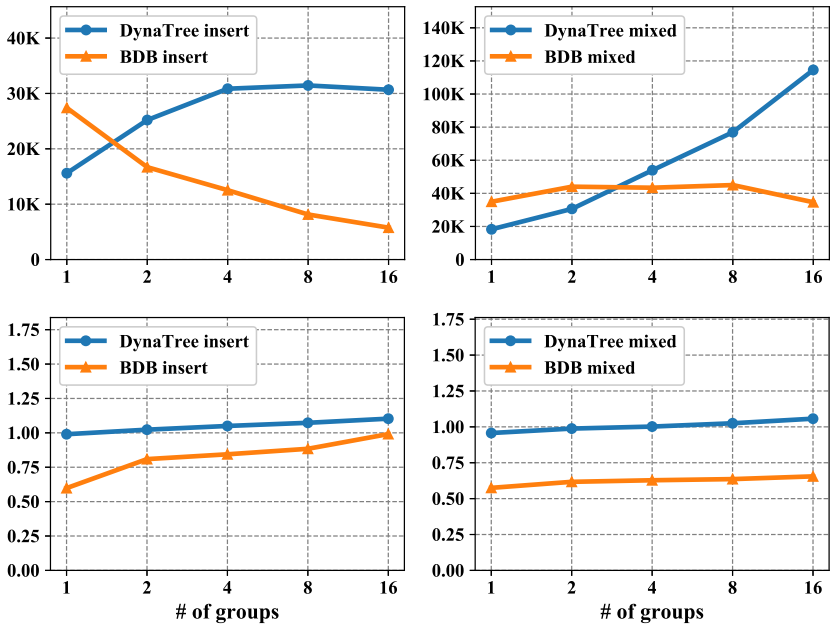

Figure 3.3. Peak throughput and contention-free latency for insertion and mixed workloads in DynaTree and BrekeleyDB.

3.2.6 Insert scalability

In order to show how DynaTree scales out for the 100% insert workload, we conducted experiments where we insert keys randomly, chosen from a uniform distribution. Figure 3.3 shows the peak throughput and contention-free latency of both DynaTree and BerkeleyDB as we vary the number of partitions and replicas, respectively.

Insertions scale well up to four partitions in DynaTree. Performance stops growing when the oracle is saturated with the high number of node creations. Each node, when it is full, does not accept a new key and asks the client to retry the request by splitting the node. Each split request involves the oracle for the creation of the new node. When the oracle gets saturated, it cannot handle more requests and the throughput stops growing. However, performance does not decrease with the number of partitions.

BerkeleyDB has good performance with one partition but loses performance with the growing number of partitions. Each replica added to the system has to receive all insertions, and the read-write replica waits for acknowledgements from all other replicas before responding to the client. This explains why the throughput for insertion decreases with the number of replicas.

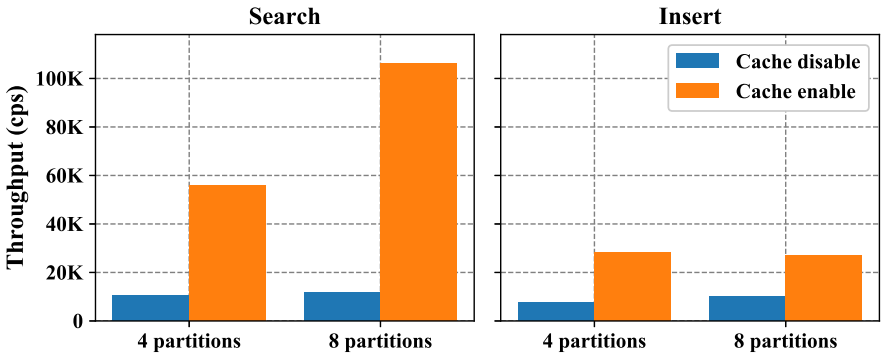

Figure 3.4. Effect of client cache on throughput.

3.2.7 Mixed workload

In this experiment, we assess the performance of DynaTree and BerkeleyDB with a mixed workload consisting of 80% of search requests, 15% of update requests, and 5% of insert requests.

In the mixed workload, shown in Figure 3.3, BerkeleyDB's performance increases from one to two partitions, then remains stable up to eight partitions, after which it decreases. The expensive updates and inserts affect the mixed workload even with 5 percent of inserts. DynaTree scales well up to 16 partitions. The reason is that DynaTree scales linearly with searches and updates, and the rate of inserts in the mixed workload is within the range that the oracle can handle. Both BerkeleyDB and DynaTree experience little variation in latency with the number of replicas and partitions, respectively.

3.2.8 Client cache impact

The client cache plays an important role in DynaTree. Thanks to cached data, clients can identify the partitions involved in the execution of a request before the request is executed by the servers. However, requests based on outdated cached data may lead to expensive retries. One question that arises is whether the client cache indeed improves performance.

When the tree structure changes, cached data must be invalidated and clients may need to retry requests based on stale data. Retries are expensive since clients need to interactively rebuild their cached tree, starting from an up-to-date tree node, or the tree root, in the worst case. Notice that while the client cache reduces the probability of retrying a request, it does not eliminate retries due to concurrent accesses to common parts of the tree.

We mitigate the effect of cache invalidation by invalidating the client cache one level at a time rather than invalidating the whole cache. Since the changes in the tree are propagated bottom-up, it is probable that only a small part of a branch has been modified.

Figure 3.4 shows the performance of DynaTree with and without the client cache. We conducted experiments with search-only and insert-only requests in deployments with 4 and 8 partitions. With no cache stored locally, clients start with fetching root node for each traversal. This affects both search and insert requests considerably, with direct implications on performance. As the graphs show, the insertion workloads, which expect high frequency of cache invalidation, do gain performance improvement from the client cache. Search workloads gain even higher performance improvement due to no cache invalidation.

3.3 Conclusion

In this chapter, we presented DynaTree, a distributed B+tree that is both scalable and fault-tolerant. DynaTree provides an architectural design for a B+tree whose nodes are distributed among a number of partitions in a partitioned state machine replication system. It is shown that building a complex data structure such as a B+tree in partitioned SMR implies a number of challenges. We discussed those challenges and presented distributed algorithms for tree operations that handle concurrent tree modifications while ensuring strong consistency. The results show that both read and update operations scale linearly with the number of partitions. Moreover, insert operations do not lose performance with the growing number of partitions.

Chapter 4

RDMA-based Atomic Multicast

Today's online services are expected to operate uninterruptedly despite server failures. Some services are also latency-critical and require to operate at microsecond scale to handle ever-increasing demand. Services that match these expectations are deemed highly available and scalable. Many years of research in dependable distributed systems have deepened the understanding of how to design systems that can tolerate failures. A golden rule is that abstractions can significantly reduce complexity, and avoid design and programming errors.

In order to both scale performance and tolerate failures, service state is typically sharded and each shard is replicated (e.g., [7; 27; 51]). Atomic multicast is a group communication abstraction that generalizes atomic broadcast [22; 46] by allowing requests to be propagated to groups of processes (in this case shards) with reliability and order guarantees. Intuitively, all correct processes addressed by a request must deliver the request and processes must agree on the order of delivered requests. Over the years, many atomic multicast algorithms have been designed for the message-passing system model (e.g., [14; 20; 25; 30; 42; 80]).

This chapter presents RamCast, the first atomic multicast protocol tailor-made for the shared-memory model. Our motivation is practical: recent years have seen widespread development and adoption of Remote Direct Memory Access (RDMA) technology. RDMA extends the traditional send and receive communication primitives with read and write operations on shared memory. RDMA's shared-memory primitives provide inter-node communications with microsecond latency. In addition, RDMA offers the possibility for a process to safeguard its memory by specifying which processes can read or write which regions of its memory. This guarantee is quite powerful. In particular, if every process revokes the write permission of other processes before writing to shared memory, then a process that writes successfully knows that it executed in isolation, without hav-

ing to take additional steps (e.g., reading the memory). Protocols can leverage this property to optimize performance [2].

In order to deliver performance that largely outperforms the most efficient message-passing atomic multicast protocols, RamCast combines ideas from Skeen's genuine atomic multicast algorithm [19], a blocking algorithm that has been used as the basis for fault-tolerant atomic multicast protocols (e.g., [25; 42]), the leader-follower replication model, explored by atomic multicast and broadcast protocols (e.g., [1; 9; 42; 57]), and RDMA technology, recently used to boost the performance of distributed systems (e.g., [2; 58; 59; 82]). RamCast can order messages multicast to a single group after 2 RDMA write delays, at the leader process, and order messages multicast to multiple groups after 3 RDMA write delays, at both the leader and followers. As a reference, the most delay-efficient message-passing atomic multicast algorithm orders messages after 3 communication delays at the leader process and after 4 communication delays at the followers [42]. We have implemented and evaluated RamCast under various conditions. We show that RamCast outperforms current state-of-the-art atomic multicast protocols, increasing throughput by up to 3.7× and reducing latency by up to 28×.

The remainder of the chapter is structured as follows. Section 4.1 provides a short description of the ideas that have inspired RamCast. Section 4.2 details the RamCast protocol by describing its normal behavior, in the absence of failures, and how it handles failures. Section 4.3 presents our prototype, and Section 4.4 details its performance. Section 4.5 concludes the chapter.

4.1 General idea

In this section, we recall the problem statement (§4.1.1) and the building blocks that inspired RamCast (§4.1.2) and an initial experiment that evaluates the performance of RDMA primitives (§4.1.3).

4.1.1 Problem statement

RamCast implements atomic multicast as defined in Section 2.3 We assume partitions, as defined Section 2.1, contain $n = 2f + 1$ processes, where f is the maximum number of faulty processes per partition. The assumption about disjoint partitions has little practical implication since it does not prevent collocating processes that are members of different groups on the same machine. Yet,

it is important since atomic multicast requires strong assumptions when groups intersect [44]. A set of $f + 1$ processes in group g is a *quorum* in g.

We require atomic multicast to be *genuine* [44]: an atomic multicast algorithm is genuine if in any run in which a message m is multicast, then for every process p that participates in ordering m, p is the process that multicasts m or $p \in g$ and $g \in m.dst$.

4.1.2 Building blocks

RamCast leverages two ideas, Skeen's atomic multicast algorithm [19] and Protected Memory Paxos [2]. Skeen's algorithm orders messages multicast to multiple processes consistently but it does not tolerate failures. Protected Memory Paxos takes advantage of RDMA permissions to improve the efficiency of Paxos [68]. Like Paxos, it implements atomic broadcast (i.e., it assumes a single group of processes).

Skeen's atomic multicast

In Skeen's algorithm, there is one process per group, and each process assigns unique timestamps to multicast messages based on a logical clock [66]. The correctness of the algorithm stems from two basic properties: (i) processes in the destination of a multicast message first assign tentative timestamps to the message and eventually agree on the message's final timestamp; and (ii) processes deliver messages according to their final timestamp. These properties are implemented as follows.

(i) To multicast a message m to a set of processes, p sends m to the destinations. Upon receiving m, each destination updates its logical clock, assigns a tentative timestamp to m, stores m and its timestamp in a buffer, and sends m's timestamp to all destinations. Upon receiving timestamps from all destinations in $m.dst$, a process computes m's final timestamp as the maximum among all received tentative timestamps for m.

(ii) Messages are delivered respecting the order of their final timestamp. A process p delivers m when it can ascertain that m's final timestamp is smaller than the final timestamp of any messages p will deliver after m (intuitively, this holds because logical clocks are monotonically increasing).

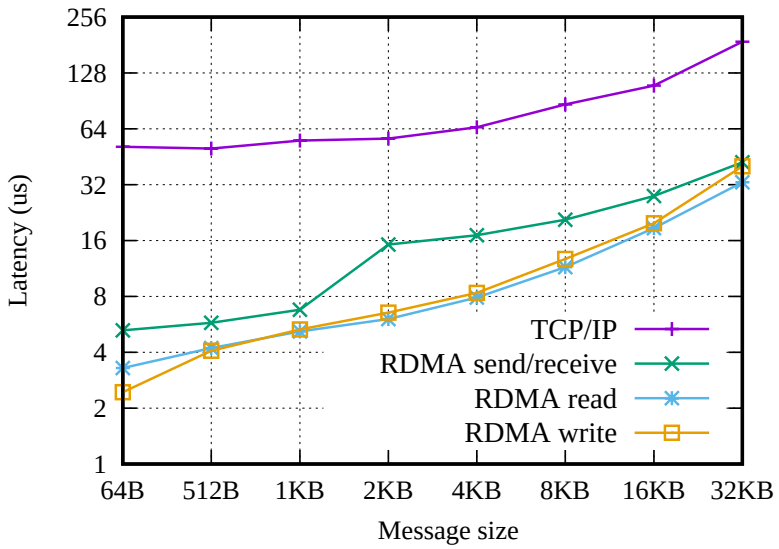

Figure 4.1. Performance comparison between communication primitives.

Protected Memory Paxos

In Paxos [68], to order a message m, the leader proposes m in a consensus instance. In the normal case, where there is a single leader, the followers accept the proposed message and reply to the leader. In Protected Memory Paxos, the followers grant exclusive write permission to their memory to the leader. If a new leader takes over, then it revokes the permission of the previous leader. To order m, the leader writes m in the memory of the followers. If the leader succeeds in writing the message in the memory of a quorum of followers, then no other leader took over, and the message is ordered.

Just like Paxos, to ensure that the new leader makes decisions that are consistent with the decisions of the previous leader, each leader associates a *round* to its proposed message. Rounds are unique across the system. When process q becomes leader, upon suspecting the failure of the current leader p, q must pick a round bigger than p's round. Process q then proceeds in two steps. First, q needs to acquire permission from a quorum of processes, which it does by contacting all processes and providing its chosen round. Processes grant write permission to q if the provided round is bigger than the round of the process that currently holds the write permission. Second, q must check whether other processes have already accepted any values. If so, q must propose the value that has been accepted in the largest round; otherwise, q can propose a new value.

4.1.3 Performance of RDMA

Designing an efficient RDMA-based atomic multicast protocol is not trivial, since RDMA's communication primitives vary substantially in performance [58; 82]. Figure 4.1 compares the latency of TCP/IP to RDMA's send/receive and read-/write operations (setup details are presented in Section 4.4.2). RDMA primitives largely outperform TCP/IP communication because they bypass the network stack. RDMA shared-memory primitives deliver performance superior to message-passing primitives, although the advantage depends on the message size. In our environment, RDMA read and write operations have similar performance, unless the write operation can "inline" data (i.e., with 64-byte messages in our case) [82]. This happens because the interface adapter has the data available and does not need to retrieve data from the memory.

RamCast uses remote writes only and avoids remote reads. There are two reasons for this design. First, most writes are small (e.g., acknowledgments) and can be inlined. Second, a process detects a write in shared memory with busy-polling reads, and reading from a process's own memory is faster than reading from the memory of another process. Consequently, process p can read process q's write more efficiently when q issues a (remote) write on p's memory and p issues busy-polling reads on its own memory.

4.2 RamCast design and architecture

In this section, we present RamCast's design and algorithms. We start with an overview of RamCast (§4.2.1), then detail its data structures (§4.2.2) and algorithms in the absence of failures (§4.2.3) and in the presence of failures (§4.2.4). We argue for the correctness of RamCast (§4.2.5) and conclude with a few extensions to the protocol (§4.2.6).

4.2.1 Design

Figure 4.2 depicts the various components and memory layout of RamCast. Processes within each group coordinate using the leader-follower model [1; 9; 42; 57]. Each server process has a fixed-size buffer per client, analogously to other RDMA-based systems [1; 31; 87; 103]. A buffer in RamCast is divided into two parts, a message buffer M, and a timestamp buffer T. Message buffer M is a shared memory region that can be read and written by any processes, including the client process the buffer is associated with. Timestamp buffer T is protected and can only be written by the leader of each group; the buffer can be read by

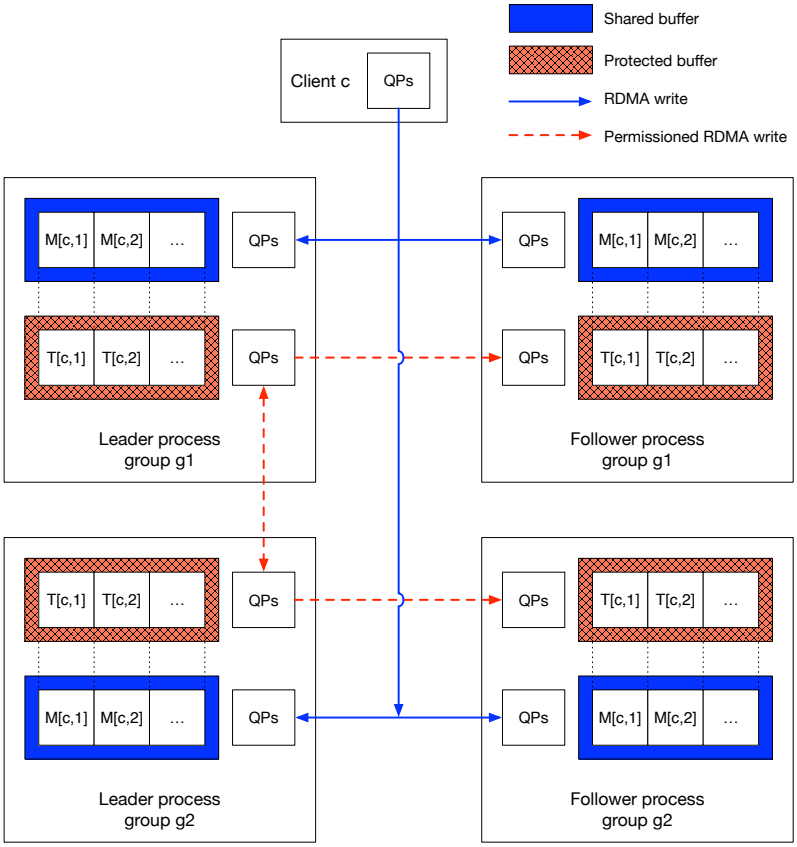

Figure 4.2. RamCast's memory layout.

any processes. Each slot in M, with a multicast message m, has a corresponding slot in T, with m's timestamp.

Clients keep a copy of the remote head and tail pointer of their buffer at each server. A client increases the remote tail after writing to the shared memory. The server process updates the head pointer on the client buffer after handling the message. Each process p periodically polls the memory cell at the head position of each connected QPs to detect new messages.

RamCast consists of the following main components:

- *Memory management.* This component handles the shared buffer and the protected buffer (detailed in Section 4.2.2). While all processes have read and write access to the shared buffer of a process, only the leader of each group has write permission to the protected buffer of a process.

- *RDMA communication.* This component provides functions to read and write remote memory, used by the normal execution component, and to

send and receive messages, used by the failure handling module.

- *Normal execution.* The normal protocol execution (detailed in Section 4.2.3) is invoked when there is a sole leader per group with support of at least a majority of processes in its group. A leader is responsible for proposing the group's timestamp for a new multicast message, and propagating timestamps from other groups to followers of its own group.

- *Failure handling.* Upon detecting the failure of a leader, processes in a group elect a new leader. The new leader must ensure that its execution is consistent with the execution of the previous leader (detailed in Section 4.2.4).

- *Leader election.* RamCast requires processes to detect a slow or crashed leader, and elect a new leader. Leader election is not assumed to be perfect: the protocol ensures safety despite multiple leaders in a group. To ensure progress, though, eventually every group should elect a single operational and stable leader [2; 68]. Stable leader election can be implemented in the partially synchronous model [3].

4.2.2 Data structures

Algorithm 6 presents the data structures used by processes in RamCast. Every server process has a shared buffer M per client c, where slot $M[c,i]$ contains the i-th message msg multicast by client c, the groups dst the message is addressed to, and an address vector ptr, where $ptr[g,p] = j$ means that at process p in group g message msg is stored in slot $M[c,j]$. The address vector is used by a process to know where to write in the memory of another process addressed by the message. Servers compute the message's timestamp tmp, based on the timestamps proposed by the leader of each destination group, and the acknowledgements from the members of the leader's group, stored in vector ack. A multicast message state $stat$ can be null ($\perp$), pending a final timestamp (MCAST), assigned a final timestamp (ORDERED) or delivered (DONE).

To compute a message's timestamp, each server process has a protected buffer T, where the i-th slot $T[c,i]$ matches the corresponding slot $M[c,i]$ in the shared buffer M associated with c. The slot contains a timestamp vector tmp and a round vector rnd, each one with an entry per group g: $tmp[g]$ contains the timestamp proposed by the current leader in g in round $rnd[g]$. Timestamps and rounds are tuples $\langle cnt, pid \rangle$, where cnt is a scalar and pid is a process identifier,

Algorithm 6 Data structures

1: Each server has a shared buffer M and a protected buffer T per client c; each slot in M stores a multicast message; the corresponding slot in T stores the message's timestamps

2: Each slot $M[c,i]$ contains the following information:
3: msg: the message m multicast by client c
4: dst: destination groups m is addressed to
5: $ptr[1..k, 1..n]$: for each g in destination, slot with msg at processes in g; $null$ if g is not in the message's destination
6: tmp: the timestamp of m, initially $\langle 0,0\rangle$
7: $ack[1..k, 1..n]$: for each g in destination, acknowledgment of timestamp in $T[c,i].tmp[g]$ from processes in g
8: $stat$: state of m: $\perp$ (initially), MCAST, ORDERED or DONE

9: Each entry $T[c,i]$ contains the following information:
10: $tmp[1..k]$: timestamp proposed by leader of group g
11: $rnd[1..k]$: the round of g's leader, initially $\langle 0,0\rangle$

12: Each client c has vector $cptr[1..k, 1..n]$, with the next available slot in buffers M and T per group g and process p

13: Each server p at group g also has:
14: $clock$: logical timestamp counter at p, initially $\langle 0,p\rangle$
15: $round$: the round of p, when leader, initially $\langle 0,p\rangle$
16: $Leader[1..k]$: the leader at each group
17: $Round[1..k]$: last accepted round at each group

unique across the system. It follows that $\langle cnt, pid\rangle > \langle cnt', pid'\rangle$ iff $cnt > cnt'$, or $cnt = cnt'$ and $pid > pid'$. We further assume that $time(\langle cnt, pid\rangle) = cnt$.

To multicast a message, a client writes the message in the shared buffer of each process in the groups addressed by the message. To know in which entry the multicast message must be written, the client keeps an address vector $cptr$ with an entry per group and per process in the group.

Process p's local state includes a $clock$, used to compute logical timestamps, p's current round, used when p is the leader of the group, vector $Leader$ with p's view on the current leader of each group, and vector $Round$, where entry $Round[g]$ contains the largest round p has accepted from $Leader[g]$.

4.2.3 Normal execution

RamCast is optimized for the normal case, when a message is addressed to groups whose leaders are operational and stable. The normal execution proceeds in three steps. In step 1, the client writes a multicast message in the memory of all destination processes. In step 2, the leader of each destination group pro-

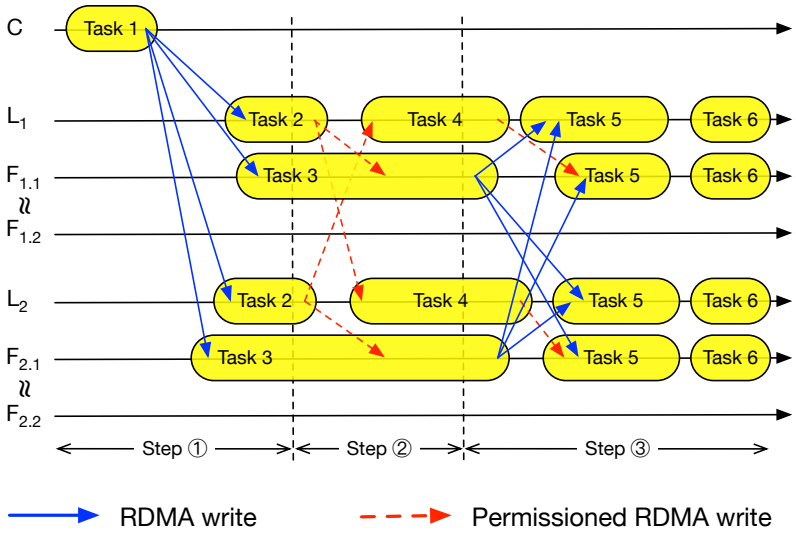

Figure 4.3. Normal execution of RamCast (i.e., group leaders are operational and stable). We show the steps at one follower per group only to avoid cluttering.

poses and writes a timestamp for the message in the memory of its followers and other leaders. In step 3, the leaders propagate the timestamp written by other leaders, and the followers acknowledge the proposed timestamps. The message is delivered after step 3. Algorithm 7 presents RamCast's normal execution, and Figure 4.3 illustrates one normal execution. We explain next the behavior of each one of the tasks in Algorithm 7.

- *Task 1.* To multicast message m, client c first calculates the next available slot in the buffer of every process addressed by m. Then, c invokes the $Relay$ procedure, which copies the message, its destination, and the address vector for m in the message buffer of every process addressed by m.

- *Task 2.* Once leader process L in group g reads m from its message buffer, it computes a group-wise timestamp for m and writes the proposed timestamp in the protected timestamp buffer of all follower processes in the leader's group g, and all leader processes of other groups in the destination of m. If a remote write is denied, then L ends the task since another process in g became leader.

- *Task 3.* When a follower detects that a multicast message has been assigned a timestamp by the group leader, it updates its clock. This is done to ensure

Algorithm 7 Normal case (stable leader)

1: Client c multicasts message m to groups in $m.dst$ as follows:
2: for each h in $m.dst$, for each q in h ***{Task 1}***
3: $cptr[q] \leftarrow cptr[q] + 1$
4: $Relay(c, m, m.dst, cptr)$

5: Server p in group g executes as follows:
6: **when** $\exists c, i : M[c,i].stat = \text{MCAST}$ **and** $p = Leader[g]$ ***{Task 2}***
7: $clock \leftarrow clock + 1$
8: for each follower q in g **and** each leader q in $M[c,i].dst$
9: $j \leftarrow M[c,i].ptr[q]$
10: write$(q, T[c,j].tmp[g], \langle clock, p \rangle)$
11: write$(q, T[c,j].rnd[g], round)$
12: **if** write denied **then** end task

13: **when** $\exists c, i : M[c,i].stat = \text{MCAST}$ **and** $T[c,i].rnd[g] = Round[g]$ ***{Task 3}***
14: $clock \leftarrow max(clock, time(T[c,i].tmp[g]))$
15: for each h in $M[c,i].dst$, for each q in h
16: $j \leftarrow M[c,i].ptr[q]$
17: write$(q, M[c,j].ack[p], Round[g])$

18: **when** $\exists c, i, h : M[c,i].stat = \text{MCAST}$ **and** $T[c,i].rnd[h] = Round[h]$ **and** $h \neq g$ ***{Task 4}***
19: $clock \leftarrow max(clock, time(T[c,i].tmp[g]))$
20: for each follower q in g
21: $j \leftarrow M[c,i].ptr[q]$
22: write$(q, T[c,j].tmp[h], T[c,i].tmp[h])$
23: **if** write denied **then** end task

24: **when** $\exists c, i, h : M[c,i].stat = \text{MCAST}$ **and** $\exists$ quorum Q in h: for each q in Q:
$M[c,i].ack[q] = Round[h]$ ***{Task 5}***
25: $M[c,i].tmp \leftarrow max(M[c,i].tmp, T[c,i].tmp[h])$
26: **if** for each group h in $M[c,i].dst$: $\exists$ quorum Q in h: for each q in Q:
$M[c,i].ack[q] = Round[h]$ **then**
27: $M[c,i].stat \leftarrow \text{ORDERED}$

28: **when** $\exists c, i : M[c,i].stat = \text{ORDERED}$ **and** $\nexists d, j : M[d,j].stat \in \{\text{ORDERED}, \text{MCAST}\}$
and $M[d,j].tmp < M[c,i].tmp$ ***{Task 6}***
29: deliver m
30: $M[c,i].stat \leftarrow \text{DONE}$

31: **procedure** $Relay(c, msg, dst, ptr)$
32: for each h in dst: for each q in h
33: write$(q, M[c, ptr[q]].msg, msg)$
34: write$(q, M[c, ptr[q]].dst, dst)$
35: write$(q, M[c, ptr[q]].ptr, ptr)$
36: write$(q, M[c, ptr[q]].stat, \text{MCAST})$

that timestamps from a group are monotonically increasing (necessary in case the process becomes leader). Then, the follower acknowledges that it has read this timestamp by writing the round used by the leader in its entry in the *ack* vector of every process in the destination of the message. The follower detects the leader proposed timestamp by checking whether the round of the timestamp matches the round associated with the leader. This assumes that both the timestamp and the round have been updated by the leader. We discuss in Section 4.3 how we ensure this with RDMA.

- *Task 4.* When the leader L in g reads a timestamp written by a leader in another group, L updates its local clock with the timestamp, to ensure that any future proposed timestmaps will be bigger, and writes the read timestamp in the memory of each one of its followers. This task is executed by the leader of a group only, since only the leader is updated with timestamps from the leader of another group (see Task 2). The reason why only the leader of a group is updated is to ensure that any timestamps assigned by the leader are consistent with any other timestamps assigned by the group.

- *Task 5.* When a message has a timestamp proposed by a leader from each destination group of the message, and a quorum of processes in each destination group agrees with the proposed timestamp, the message becomes ordered.

- *Task 6.* A process delivers an ordered message when it can assert that no other messages can be assigned a smaller timestamp.

In the normal case, a message multicast by a client to multiple groups is delivered by the leaders and the followers of the addressed groups after three RDMA write delays (see Figure 4.3). In Section 4.2.6, we discuss how messages addressed to a single group can be delivered by the group's leader after two RDMA write delays.

4.2.4 Handling failures

RamCast handles the failure of a leader using a mechanism similar to Paxos. As a consequence, it can tolerate multiple processes that believe to be leader in a group without violating safety. In order to ensure progress, however, eventually there should be only one operational leader process per group. When a process becomes leader, it needs to catch up with the previous leader. In the following, we describe how the newly elected leader does this. The procedure uses both

shared memory and message passing for communication. In RDMA, message passing is less efficient than shared memory, but it reduces complexity, as we do not have to handle concurrent accesses to shared memory. Since failures are hopefully rare, we consider that trading performance for simplicity is acceptable.

- *Task 7.* When a process that will become the next leader of the group suspects the current leader, it determines its *first undecided slot (FUS)* per client in its shared buffers. A slot is undecided if its state is equal to MCAST. Then, the new leader chooses a round and sends a catch-up message to every server process in the system. Since slot i in the new leader's buffer may correspond to a different slot at another process, the new leader must convert its FUS into one that is meaningful for the contacted process (line 7).

- *Task 8.* A process p will consider a catch-up message from new leader q in group h if q has picked a round bigger than the current round for h at p. This is a requirement from Paxos, to ensure that a new leader will not decide on a value different than a previously decided value. If the catch-up message can be considered, then p revokes permissions to the previous leader, grants permission to the shared buffer T to q, collects all information requested by q, and sends it to q. Finally, p updates h's round and leader.

- *Task 9.* When the new leader receives responses for a catch-up request from a quorum of processes in a group, it handles each entry i for every client c received as follows. First, the process selects the response with the largest round. From Paxos, this ensures that if a timestamp has been chosen, it can only be the one with the largest round. The next steps depend on whether the process received the responses from its own group or not. If the process received the responses from its own group, then it picks the timestamp in the selected response, if any, or picks a timestamp using its own clock. In either case, the process proposes the picked timestamp to all other members of its group and the leaders of the other involved groups. If the process received the responses from another group, then it forwards the timestamp in the selected response to the followers in its group.

We also consider the case of faulty clients, who may fail to update all destinations of a multicast message.

- *Task 10.* When a process detects the failure of a client, it relays all the messages multicast by the faulty client that have not been ordered yet. This means that only messages in the MCAST state need to be relayed.

Algorithm 8 Handling failures and suspicions

```
 1: when suspect Leader[g] and p is g's next leader                          {Task 7}
 2:   for each c do
 3:     FUS[c] ← i, where M[c,i] is the first undecided entry
 4:   round ← ⟨time(round)+1, p⟩
 5:   for each h in Γ
 6:     for each q in h
 7:       for each c: xFUS[c] ← M[c,FUS[c]].ptr[h,q]
 8:       send (CATCH_UP, xFUS, round) to q

 9: when receive (CATCH_UP, FUS, round) from q in h and round > Round[h]    {Task 8}
10:   revoke previous permissions and grant permission to q
11:   pend ← ∅
12:   for each c do
13:     let j be the last entry in M such that M[c,j] ≠ ⊥
14:     for i in FUS[c]..j do
15:       if h ∈ M[c,i].dst then
16:         pend ← pend ∪ (c, i, M[c,i].msg, M[c,i].dst, M[c,i].ptr, T[c,i].tmp[g],
                                                                  T[c,i].rnd[g])
17:   send (MY_STATE, pend) to q
18:   Round[h] ← round
19:   Leader[h] ← q

20: when receive (MY_STATE, pend) from quorum Q in h,
        including p's response if g = h                                     {Task 9}
21:   bag ← union of all received pend from h
22:   let maxts be the largest timestamp tmp in bag
23:   clock ← max(clock, time(maxts))
24:   for each (c,i,−,−,−,−,−) in bag
25:     let (c,i,msg,dst,ptr,tmp,rnd) in bag be such that ∄(c,i,−,−,−,−,rnd′) in bag
              and rnd′ > rnd
26:     if g = h then
27:       if rnd > 0 then
28:         t ← tmp
29:       else
30:         clock ← clock + 1
31:         t ← ⟨clock, g⟩
32:       for each q in g and each leader q in dst
33:         write(q, T[c,ptr[q]].tmp[g], t)
34:         write(q, T[c,ptr[q]].rnd[g], round)
35:         if write denied then end task
36:     else
37:       for each q in g
38:         write(q, T[c,ptr[q]].tmp[h], tmp)
39:         if write denied then end task

40: when suspect client c                                                   {Task 10}
41:   for each i such that M[c,i].stat = MCAST
42:     Relay(c, M[c,i].msg, M[c,i].dst, M[c,i].ptr)
```

4.2.5 Correctness

In this section, we argue that RamCast implements atomic multicast, as defined in Section 2.3.

Proposition 1 *(Uniform integrity) For any message m, every process p delivers m at most once, and only if p is a destination of m and m was previously multicast.*

PROOF: Process p delivers m at Task 6 if m's state is ORDERED. After delivering m, p sets m's state to DONE, and thus m cannot be delivered more than once.

Let c be the client that multicasts m to groups in dst, and let p be in group g. From Task 6, p only delivers m if it is in p's M buffer and m's state is ORDERED. Message m's state is set to ORDERED in Task 5 if its current state is MCAST. A message's state is set to MCAST in procedure $Relay$, which is invoked in two cases: (a) by client c upon multicasting m (Task 1) to groups in dst, in which case $g \in dst$; or (b) by some process q that suspects c (Task 10), has m in its buffer in state MCAST, and g is a destination of m. In case (b), m was written in q's buffer either (b.1) directly by c or (b.2) indirectly by some other process. In any case, there is some process r such that m is included in r's buffer by c. It follows from Task 1 that p is a destination of m and m was multicast by client c. □

Lemma 1 *If all correct processes in the destination of an atomically multicast message m have m in their M buffer in the* MCAST *state, then they eventually set m to the* ORDERED *state.*

PROOF: Let m be addressed to groups in dst and q be a correct process addressed by m. We claim that for each $h \in dst$, q will have a timestamp for h that is acknowledged by a quorum of processes in h. By the leader election oracle and the fact that each group has a majority of correct processes, group h eventually has a stable correct leader l. Either (a) l executes Task 2 and proposes its clock value as h's timestamp or (b) l executes Task 7 to replace a suspected leader. In (b), l sends a CATCH_UP message to all processes and will receive for each group $g \in dst$ the timestamp proposed in g, if any, and the corresponding acknowledgements from processes in g (Task 8). For the case where $h = g$, l will pick the timestamp decided by a previous leader or choose one if no timestamp has been decided (Task 9). Thus, in both cases (a) and (b), the leader writes the chosen timestamp in the M buffer of each process in h and in the leaders of other groups in dst. From Task 3, every follower in h will acknowledge this timestamp

in the buffer of each process in the destination of m. From Task 4, when l has a timestamp from $g \neq h$, l writes the timestamp in the buffer of its followers, which concludes the claim. Therefore, eventually q has a timestamp for every group in dst, can compute m's final timestamp, and set m's state as ORDERED.

Lemma 2 *If a correct process p has an atomically multicast message m in its M buffer in the* ORDERED *state, p eventually delivers m.*

PROOF: Assume for a contradiction that q does not deliver m. Thus, there is some message m' in the buffer such that $m \neq m'$, m''s timestamp is smaller than m's timestamp, and m''s state is not DONE.

We first show that any message added in the buffer after m becomes ORDERED has a timestamp bigger than m's timestamp. Message m only becomes ordered after it has timestamps from all groups in m's destinations dst. When q reads a timestamp x for m from some group in dst, q updates its clock such that it contains the maximum between its current value and x. Since the next event that q handles for a message m'' will increment its clock, it follows that m'' will have a timestamp bigger than x.

We now show that every message that contains a timestamp smaller than m's final timestamp ts is eventually delivered and its state set to DONE. To see why, let m' be the message with the smallest timestamp in the buffer. Thus, such a message is eventually delivered and its state set to ORDERED. Eventually, m will be the message in the buffer with smallest timestamp and therefore delivered, a contradiction. We conclude then that q eventually delivers m. □

Proposition 2 *(Validity) If a correct client c multicasts a message m, then eventually every correct process p in m's destination dst delivers m.*

PROOF: Upon multicasting m, c relays m to groups in dst (see Task 1). The Relay procedure then copies m to the M buffer of every correct process p in groups in dst and sets its state to MCAST. From Lemma 1, it follows that every correct process p set m's state to ORDERED. From Lemma 2, p eventually delivers m. □

Proposition 3 *(Uniform agreement) If a process p delivers a message m, then eventually all correct processes q in m's destination dst deliver m.*

PROOF: For process p to deliver m, from Task 6, p has a timestamp for every group h in dst in the M buffer such that ts is the largest among these timestamps.

Moreover, there is no message m' in the buffer such that $m \neq m'$, $ts < y$, where y is a timestamp assigned to m', and m' is not ordered.

We first show by contradiction that q eventually has m in its M buffer. Let c be the client that multicasts m. If c is correct then, c writes m in q's buffer, so consider that c fails before it can write m in q's buffer. Since p delivers m, it has a quorum of acknowledges from each group in dst. Any quorum includes at least one correct process, which from Task 10, eventually suspects c and relays m to all processes in dst, including q, a contradiction.

It follows from Lemma 1 that q eventually sets the state of m to ORDERED in its buffer, and from Lemma 2 that q eventually delivers m. □

Proposition 4 *(Uniform prefix order) For any two messages m and m' and any two processes p and q such that $\{p, q\} \subseteq dst \cap dst'$, where dst and dst' are the groups addressed by m and m', respectively, if p delivers m and q delivers m', then either p delivers m' before m or q delivers m before m'.*

PROOF: The proposition trivially holds if p and q are in the same group, so assume p is in group g and q is in group h and suppose, by way of contradiction, that p does not deliver m' before m nor does q deliver m before m'. Without loss of generality, suppose that m's timestamp ts is smaller than m''s timestamp ts'.

We claim that q inserts m into the M buffer before delivering m'. In order for m (respectively, m') to be delivered by p (resp., q), p's (resp., q's) M buffer must contain a timestamp ts_g from group g and ts_h from group h (resp., ts'_g from group g and ts'_h from group h).

From Task 2 (or Task 9 if some process has suspected the leader), the leader l in group g must have included the timestamp ts_g for message m and ts'_g for message m' in p's M buffer and both timestamps have been acknowledged by a quorum of processes in group g. Assume that the leader l has written ts_g before ts'_g to the M buffer of every follower in group g and the leader l_h in group h. From Task 2, we have $ts_g < ts'_g$. Therefore, from Task 4, l_h will write to the M buffer of every follower in group h, including q, both ts_g for message m and ts'_g for message m'.

Consequently, from the claim, q delivers m before m' since $m.ts < m'.ts$, a contradiction that concludes the proof. □

Proposition 5 *(Uniform acyclic order) Let relation $<$ be defined such that $m < m'$ iff there exists a process that delivers m before m'. The relation $<$ is acyclic.*

PROOF: Suppose, by way of contradiction, that there exist messages $m_1, ..., m_k$ such that $m_1 < m_2 < ... < m_k < m_1$. From Task 6, processes deliver messages following the order of their final timestamps. Thus, there must be processes p and q such that the final timestamps they assign to m_1, ts_p and ts_q, satisfy $ts_p < ts_q$, a contradiction since both p and q have the same timestamps for each group in dst in Task 6. □

Theorem 1 *RamCast implements atomic multicast.*

PROOF: This follows directly from Propositions 1 through 5. □

4.2.6 Extensions

We now discuss how to speed up the execution of messages multicast to a single group of processes and how to reuse entries in the client buffers (i.e., essentially, how to turn the data structures into circular buffers).

Since only one process at a time can hold permission to write in the timestamp buffer of processes, if a leader manages to write its proposed timestamp for a multicast message (Task 2) in a quorum of processes, it knows that the timestamp proposed has been accepted by the followers and can change the message's state to ORDERED. Thus, at the leader the message is ready to be delivered without the acknowledgements from the followers. We use this optimization to speed up the delivery of single-group messages at the leader.

A client can recycle a buffer slot when the slot will not be needed by any processes. This is the case when all message destinations have delivered the message (i.e., message state is DONE). Therefore, periodically, all message destinations inform the client about the slot with their *Last Delivered Message (LDM)*. The client then computes the *Last Stable Group Message (LSGM)* as the lowest LDM received in the group. The client can safely update the pointer to the tail of its buffer to the LSGM. This procedure, although simple, requires feedback from all processes in a group. To tolerate failures, processes must checkpoint their state. When $f + 1$ processes in a group have checkpointed a state that includes the i-th slot, then the group's LSGM can be updated to i.

4.3 Implementation

We implemented a prototype of RamCast in Java using jVerbs (DiSNI library) version 2.1,[1] an open-source user-level networking library developed by IBM that supports RDMA communication [98]. jVerbs offers low latencies to applications running inside a Java Virtual Machine by exposing RDMA network hardware resources directly to the JVM. The source code of RamCast is publicly available.[2]

In RamCast, we applied a number of optimizations to further decrease latency and improve performance. When establishing the connections between hosts, we use two-sided operations (e.g., send and receive) to exchange memory addresses, and use the one-sided writes for data transfer. As the two-sided operation is only used for control information at the start up and in the case of failures, this procedure does not affect performance of normal execution. The one-sided operation for the actual transfer makes the overall data transfer efficient. In RDMA, writes and sends with payloads below a limit specified by devices may be written to the work request (WR) as inlined data, thus the RNIC does not need to fetch that payload via a DMA read. In RamCast, we inline all writes whose payload is lower than the inline limit (i.e., 64 bytes) [82].

Normally, the RNICs actively poll a completion event (CE) from the CQ to ensure a write resides in remote memory. Polling CE is time consuming as it involves synchronization between the RNICs on both sides of a CQ [103]. Thus, for multi-group messages, we employ *selective signaling* [58] to reduce this overhead by only checking for a CE after pushing a number of writes. When using selectively signaled writes with requests of size n, up to $n-1$ consecutive operations can be unsignaled, i.e., a CE will not be pushed for these operations. Note that if an operation ended with an error (e.g., a leader's write permission is revoked), it will generate a CE even if it was supposed to use unsignaled completion.

In a shared memory context, when a process reads entries that are updated by another process, it is important that the reader process does not read incomplete data that has not been fully updated by the writer process, (e.g., processes in RamCast continually monitor their shared buffer for new messages and may be reading an incomplete entry). We resolve this issue by adding an extra canary value at the end of each entry, as used in previous works [1; 31; 53; 58; 103]. Before writing a message to a remote host, a process in RamCast adds the checksum of the entry to the end of the entry. A remote process always first checks the checksum value and waits for the checksum to match the entry.

[1] `https://github.com/zrlio/disni`

[2] https://github.com/longle255/libRamcastV3

4.4 Experimental evaluation

In this section, we discuss the evaluation rationale (§4.4.1), describe the experimental environment (§4.4.2), and present the results of experiments we conducted (§4.4.3–4.4.5).

4.4.1 Evaluation rationale

We conducted three sets of experiments. In the first set (§4.4.3), we seek to understand the effects of message size on RamCast's performance. In the second set (§4.4.4), we compare RamCast's performance to WBCast's, an efficient message-passing atomic multicast protocol. As we will see, RamCast largely outperforms WBCast in both throughput and latency. Even though both protocols are assessed in the same environment, RamCast's advantage is a result of RDMA's efficient write operations (used by RamCast) when compared to message-passing operations (used by WBCast). In the third set of experiments (§4.4.5), we compare RamCast's "inherent performance" (i.e., in the absence of contention and queueing effects) to high-performance atomic broadcast protocols that rely on RDMA technology (APUS and Mu) or bypass the network stack (Kernel Paxos).

In the following, we briefly comment on these protocols and their configuration in the experimental study. We provide more details about each protocol in Section 6.1.

White-Box Atomic Multicast (WBCast) [42] is a genuine atomic multicast protocol that delivers exceptional performance, thanks to some algorithmic optimizations. WBCast provides a C-language implementation that uses libevent for communication.[3] We extended the code to include additional statistics information. We include WBCast in our evaluation because it is currently the best-performing message-passing atomic multicast protocol.

APUS is a general-purpose atomic broadcast protocol that implements Paxos. As part of the execution, nodes store ordered messages on stable storage (e.g., SSD). In order to ensure a fair comparison among the various protocols, which store messages in main memory only, we configured APUS with a RAM disk storage instead.

Mu [2] implements Protected Memory Paxos. It was designed to replicate micro services and optimizes atomic broadcast in one important aspect: by co-locating clients and the Paxos's leader on the same host. As a consequence, a broadcast message can be ordered after one RDMA write delay (i.e., done by

[3] `https://github.com/imdea-software/atomic-multicast`

the leader to place the message in the memory of the followers). As described in Section 4.1.2, this is enough to ensure that the message is ordered. Unfortunately, co-locating clients and leaders on the same host is not possible in atomic multicast: the motivation and scalability of atomic multicast stem from the fact that one can create multiple groups, each one operating independently. We consider Mu in our evaluation since it is the best-performing RDMA-based atomic broadcast protocol.

Kernel Paxos [37] is a Multi-Paxos implementation that improves the performance of the original libpaxos library.[4] The main idea is to reduce system calls by running Paxos logic in the Linux kernel, bypassing the network stack, and avoiding the TCP/IP stack. We used the original code[5] and deployed a single group with three replicas. We compare RamCast to Kernel Paxos because both systems avoid the overhead of the communication stack.

4.4.2 Environment and configuration

We conducted all experiments in CloudLab [32] with two sets of nodes: (a) R320 nodes, equipped with one eight-core Xeon E5-2450 processor running at 2.1GHz, 16 GB of main memory, and a Mellanox FDR CX3 NIC; and (b) XL170 nodes, equipped with one ten-core Intel E5-2640v4 processor running at 2.4GHz, 64 GB of main memory, and a Mellanox ConnectX-4 NIC. A 10 Gbps network link with around 0.1ms round-trip time connects all nodes running Ubuntu Linux 18.04 with kernel 4.15 an Oracle Java SE Runtime Environment 11. In all experiments, clients and servers are independent processes. Clients submit requests in a closed-loop, that is, a client multicasts a message to servers and waits for a response before multicasting the next message. In all RamCast experiments, clients measure latency as the interval between the multicast of a message and the response received from the first server in each group addressed by the message. In all protocols, each group has 3 processes with in-memory storage.

4.4.3 The impact of message size

In this experiment, conducted on XL170 nodes, we measure RamCast throughput and latency for different message sizes. For each message size, we increase the number of clients until the system is saturated (i.e., throughput increases minimally with the number of clients). Figure 4.4 shows that up to 4KB messages,

[4] `https://bitbucket.org/sciasciad/libpaxos`

[5] `https://github.com/esposem/Kernel_Paxos`

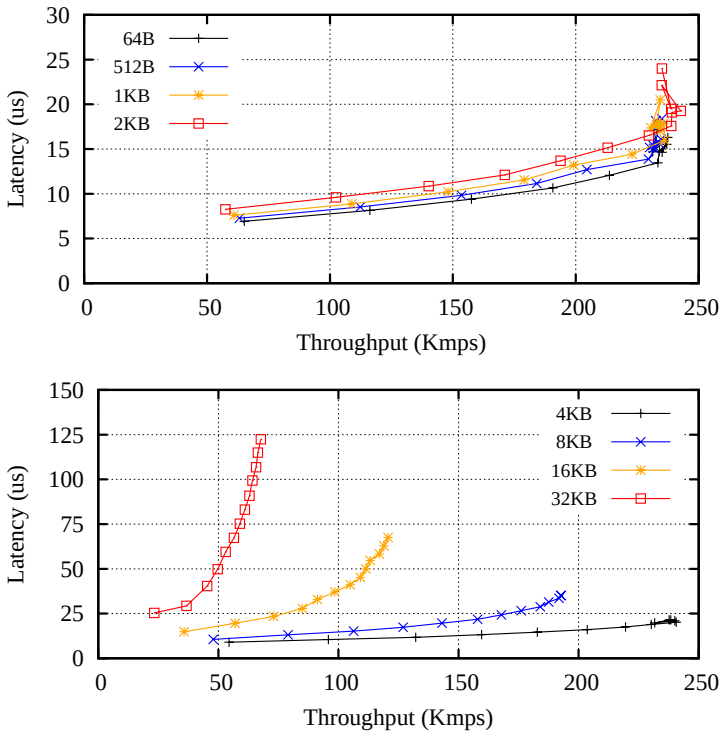

Figure 4.4. RamCast performance with different message sizes: 64B to 32 KB, throughput versus latency.

the impact of message size on the system throughput is negligible, with nearly 250 thousand messages delivered per second. As the message size increases past 4KB, the maximum throughput decreases with 70 thousand messages per second for 32KB messages. The latency cumulative distribution function (CDF) in Figure 4.5 exhibits minimum latency variation for messages with up to 2KB, around 8 microseconds at 95^{th} percentile. At 4KB messages, the latency slightly goes up to around 10 microseconds.

4.4.4 The performance of atomic multicast

The next set of experiments assess RamCast behavior in scenarios with up to 8 groups of 3 replicas each, deployed on XL170 nodes. The first experiment comprises executions in which clients multicast single-group 64-byte messages in setups with 1, 2, 4, and 8 groups.

Figure 4.6 (top) shows the aggregated throughput results when the system is saturated. The results show that the throughput of both RamCast and WBCast

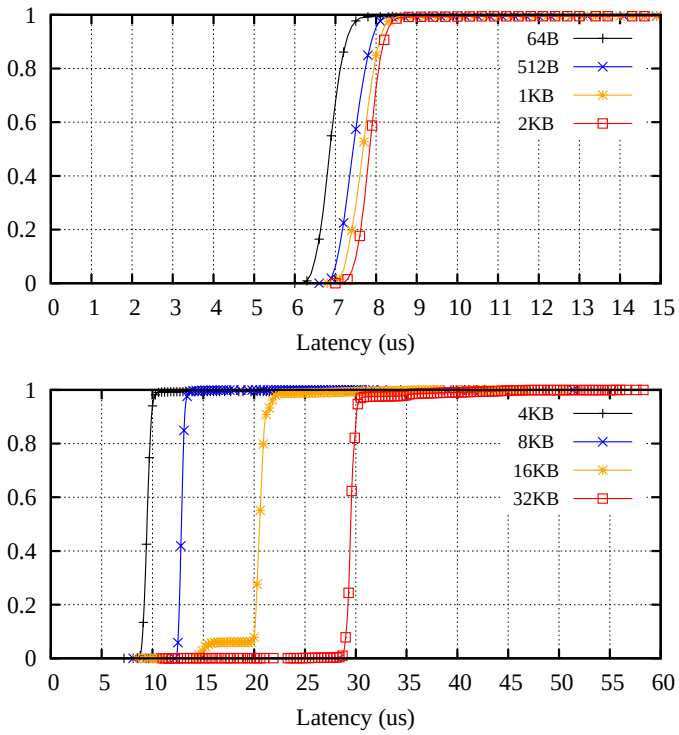

Figure 4.5. RamCast performance with different message sizes: latency cumulative distribution function for a single client.

grow linearly with the number of groups for single-group messages. RamCast outperforms WBCast, however, by a factor of 3.6× in all configurations. Since groups do not exchange any information when dealing with single-group messages, the latency CDF is similar for all configurations, no matter the number of groups in the system, as depicted in Figure 4.6 (middle and bottom). RamCast's efficient single-group multicast (see §4.2.6) together with RDMA's high performance writes grant RamCast a 28× median latency advantage to WBCast (i.e., ∼7 us against ∼200 us).

The next experiment evaluates the protocols with multi-group messages of 64 bytes addressed to all the groups. This is the most stressful case for a genuine atomic multicast protocol, since to order a multicast message, all groups addressed by the message must interact. Therefore, the more groups addressed by a message, the lower the expected performance. RamCast's maximum throughput is greater than WBCast's in every configuration with 233, 145, 80, and 40 thousand messages per second for 1, 2, 4, and 8 destination groups against 63, 50, 35, and 27 thousand for WBCast, as shown in Figure 4.7 (top). The values

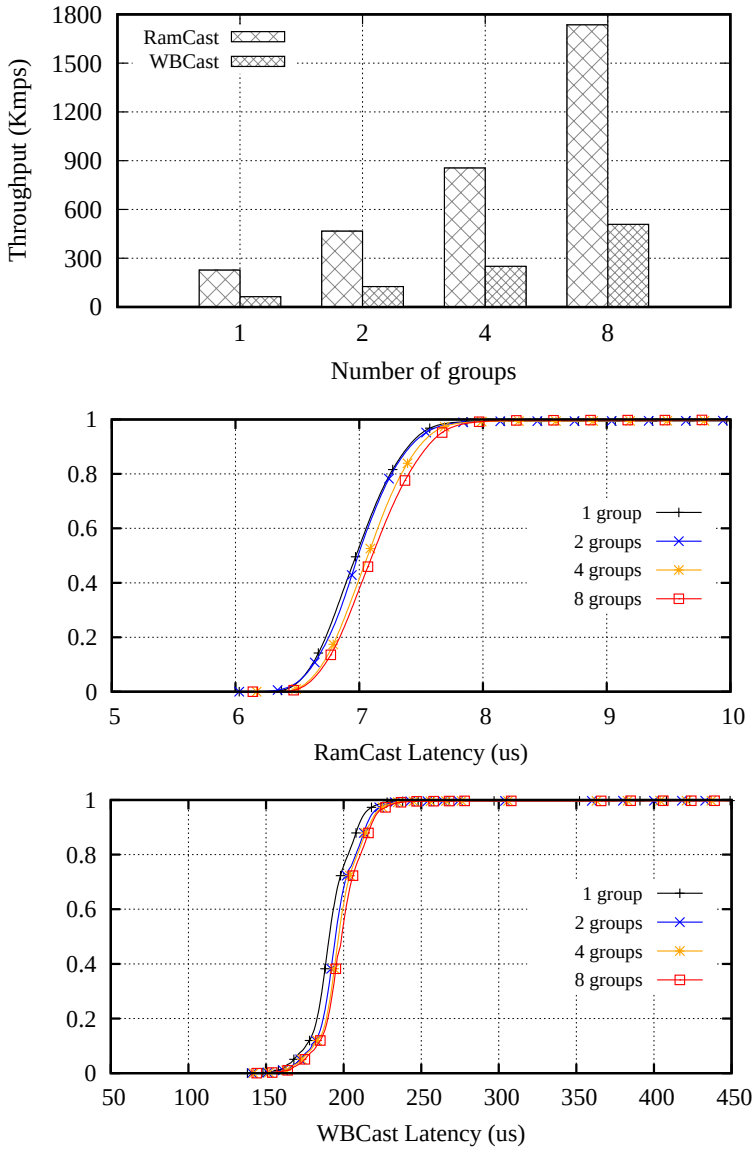

Figure 4.6. Performance of atomic multicast when messages are multicast to a single group. We show throughput (top) and latency cumulative distribution function with one client for RamCast (middle) and WBCast (bottom).

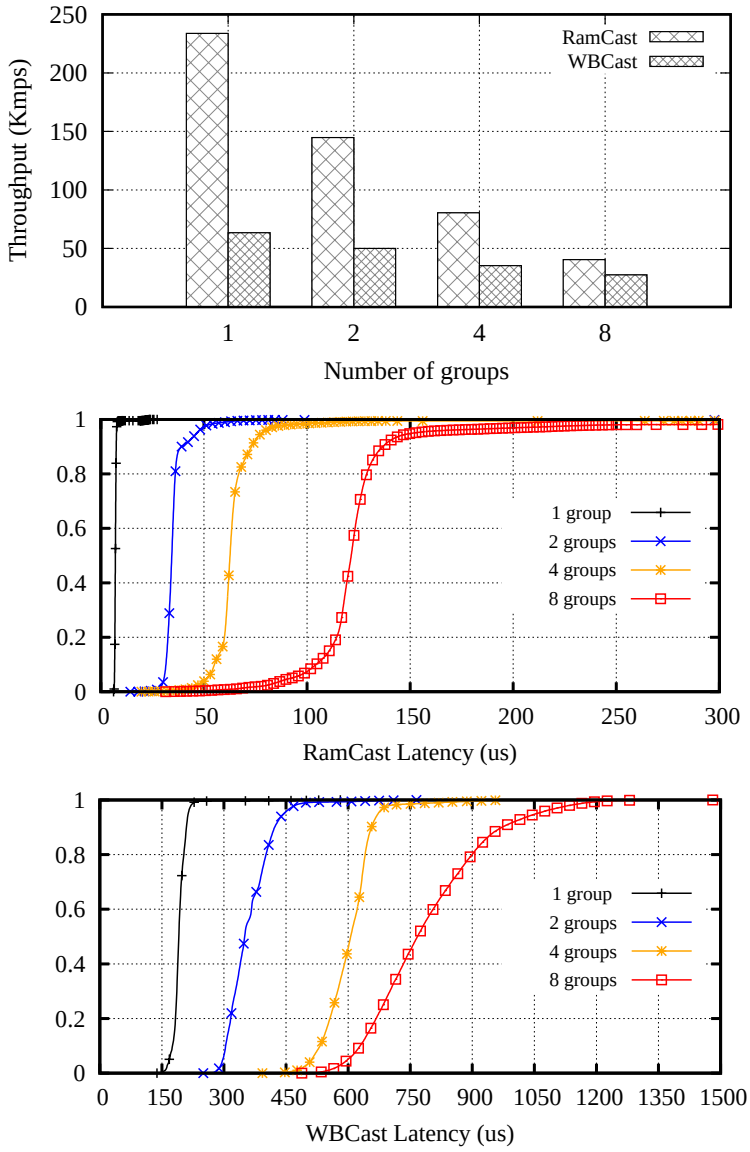

Figure 4.7. Performance of atomic multicast when messages are multicast to all groups. We show throughput (top) and latency cumulative distribution function with one client for RamCast (middle) and WBCast (bottom).

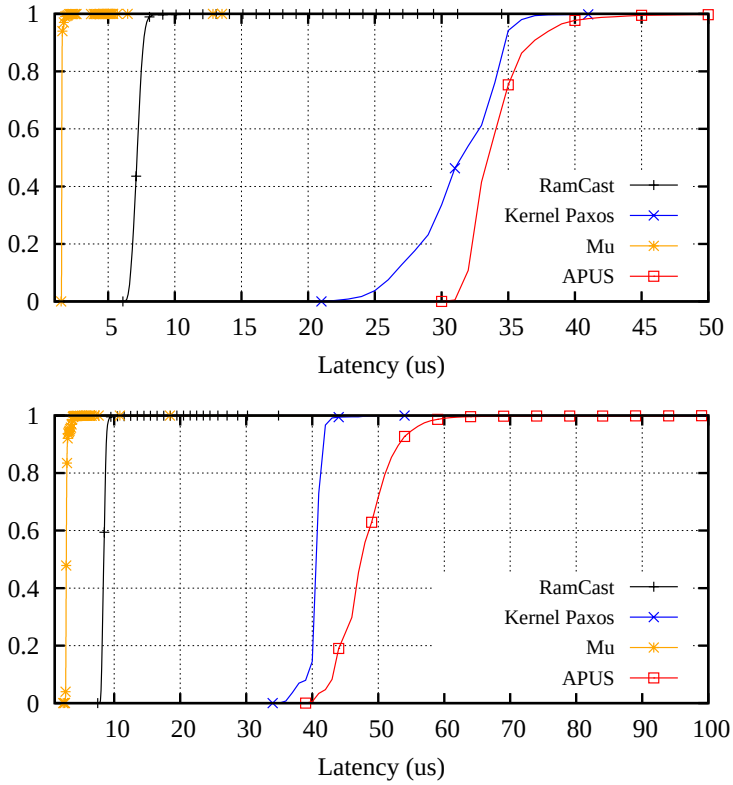

Figure 4.8. Latency cumulative distribution function for RamCast and atomic broadcast protocols with a single client: 64-byte messages (top) and 1K-byte messages (bottom).

correspond to improvements of 3.7×, 2.9×, 2.3× and 1.5×, respectively.

The difference is more expressive when we consider the latency for a single client, i.e., when both protocols are contention-free. Figure 4.7 (middle) shows that the latency CDF for RamCast with values of 8, 46, 78 and 150 microseconds for 1, 2, 4, and 8 destination groups if we consider the 95^{th} percentile. The equivalent values for WBCast, as depicted in Figure 4.7 (bottom), are 214, 445, 673, and 1055 microseconds, representing 20× to 7× slower delivery times when compared to RamCast's.

4.4.5 RamCast's inherent performance

We now compare RamCast to atomic broadcast protocols using a single group of three replicas, and 64-byte and 1-kilobyte messages, on R320 nodes. Figure 4.8 shows a similar trend for both message sizes. Mu's co-location of clients and

leader on the same host (with the resulting single RDMA write delay) significantly pays off: 4.8× and 3.1× reduction in the median latency with respect to RamCast for 64-byte and 1-kilobyte messages, respectively. However, co-locating the clients and the leaders on the same host hampers atomic multicast scalability (see §4.4.1). When compared to APUS, RamCast reduces the median latency by 4.7× and 5.6×, with messages with 64-byte and 1-kilobyte messages, respectively. Compared to to Kernel Paxos, the improvements are in the range of 4.4× and 4.7×.

4.5 Conclusion

Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available strongly consistent distributed systems. This chapter presents RamCast, the first genuine atomic multicast protocol tailor-made for the shared-memory model. In addition to introducing a novel algorithm that leverages the permission mechanism of RDMA's write operations to reduce the number of communication steps, we also have implemented and evaluated the protocol under a large range of parameters. The results show that RamCast outperforms a state-of-the-art message-passing genuine atomic multicast protocol and atomic broadcast protocols that optimize communication and rely on comparable assumptions.

Chapter 5

RDMA-based partitioned SMR

One of the main challenges that S-SMR systems face is executing multi-partition requests, that is, requests that span more than one partition. Existing solutions range from weakening the consistency of multi-partition requests (e.g., [21]) to fully implementing SMR's strong consistency [8; 49]. In DynaStar [51], for instance, a state-of-the-art S-SMR system, after ordering a request, replicas in the partitions involved in the request migrate the data needed to execute the request to the replicas of a single partition, so that these replicas can execute the request. Unfortunately, this data exchange during request execution results in substantial overhead.

This chapter presents Heron, the first scalable state machine replication system on shared memory. Heron delivers scalable throughput through state partitioning and microsecond latency by careful use of RDMA primitives. Heron relies on an RDMA-based atomic multicast protocol to consistently order requests within and across partitions, described in Chapter 4. It executes multi-partition requests using a combination of different strategies. First, replicas coordinate when executing requests to ensure linearizability and use a dual-versioning technique that keeps two versions of every object to account for concurrent access to data. Coordination encompasses a majority of replicas in each partition involved in a request. While coordinating with a majority of replicas (instead of all) avoids blocking due replica failures, it creates the possibility of laggers, slow replicas that do not keep up. Heron uses a simple heuristic, waiting for an additional small delay, to reduce the probability of laggers. Finally, laggers resort to an efficient state synchronization protocol to update their state.

We extensively evaluate Heron by considering its inherent coordination latency and performance in TPCC workloads. We found that Heron adds very low latency of around 3 microseconds for coordinating executions in a workload in

which requests involve 4 partitions. Heron is able to execute complex TPCC single-partition requests in 19 microseconds and multi-partition requests in 35 microseconds. The performance evaluation shows more than an order of magnitude performance improvement when compared to state-of-the-art message-passing based S-SMR systems. We also study Heron's state synchronization protocol and show that lagging replicas can be swiftly brought back to date. Heron is able to recover a replica that has lost recent updates of several Kilobytes in less than 30 microseconds.

The remainder of the chapter is structured as follows. Section 5.1 discusses the challenges involved in Heron's design, describes its algorithm in detail, and argues about its correctness. Section 5.2 presents our prototype, and Section 5.3 evaluates its performance. Section 5.4 concludes the paper.

5.1 General idea

In this section, we discuss the challenges involved in Heron's design (§5.1.1), present Heron in detail (§5.1.2), and argue about its correctness (§5.1.3).

5.1.1 Main design and challenges

In Heron, application state is partitioned (or sharded), for performance, and each partition is replicated, for high availability [13; 27; 41; 50; 78]. Clients use atomic multicast to propagate requests to the partitions involved in the request. A partition is involved in the request if the request reads or writes an object in the partition. Replicas execute requests using local data (i.e., reading and writing objects stored at the replica) and remote data (i.e., reading objects stored at replicas in other partitions by means of RDMA). This scheme introduces challenges in the execution of requests, in the coordination between replicas, and in how replicas within a partition keep their state synchronized. In the following, we comment on how Heron faces these challenges. Hereafter, we assume that the execution of a request has a reading phase, during which a replica reads local and remote objects without updating any objects, and a writing phase, during which the replica updates local objects. Once the replica starts the writing phase, it does not read any objects.

Request execution

Single-partition requests are executed as in classic state machine replication: replicas of a partition execute requests deterministically in the same order, using local data only. We now explain the rationale for using remote reads only in Heron when executing multi-partition requests.

In general, there are two solutions to the problem of executing a multi-partition request: (a) one partition, among the partitions involved in the request, executes the request, and (b) all involved partitions execute the request. In the first solution, to execute a request, the *active partition* reads local and remote objects, and updates its local objects and the remote objects stored in the other partitions involved in the request, the *passive partitions*. In the second solution, all involved partitions execute the request, after reading local and remote objects, and update only local objects (i.e., there is no remote write to update objects in other partitions).

The first solution saves computing resources, as requests are executed by the replicas of the active partition only. However, replicas in the active partition compete with each other to update remote objects in the passive partitions. Therefore, one must handle read-write conflicts (i.e., remote reads versus remote writes) and write-write conflicts (i.e., due to remote writes). In the second solution, replicas in all involved partitions execute the request, but update local objects only. Therefore, every replica in a partition involved in the request issues local and remote reads, but only local writes. As a consequence, there are only read-write conflicts.

Heron adopts the second solution to avoid write-write conflicts. Heron handles read-write conflicts with a coordination mechanism that strives to keep replicas synchronized, described next, and a dual-versioning technique that keeps two versions of every object. When executing a request, replicas read the most recent version of the object and update the older version. Each version is tagged with the timestamp of the request that creates the version. To determine the most recent version of an object, a replica compares timestamps and chooses the version with the largest timestamp.

Replica coordination

Multi-partition requests require replicas of the partitions involved in the request to coordinate. First, before a replica r_i executes a request R, r_i coordinates with replicas in other partitions involved in R to ensure that remote reads issued by r_i to these replicas will be consistent, that is, they reflect all requests that precede

R. Second, after r_i has executed R, r_i coordinates with replicas in other partitions involved in R to ensure that remote reads issued by these replicas to r_i will be consistent, that is, they do not reflect requests that come after R.

Although the coordination used by Heron is analogous to barriers, instead of waiting for every replica of each partition involved in a request, a replica waits for a majority of replicas in each partition. This ensures that no replica remains blocked in case of replica failures, as each partition has a majority of correct servers. To ensure reading consistent values, replicas read remote values from replicas that they have heard from in the coordination phase before execution.

Coordinating with a majority of replicas only, however, may leave a replica in a partition behind a majority of replicas in its partition, a lagger. In this case, the lagger may not be able to execute a multi-partition request because it may not be able to consistently read the value of remote objects, because the replicas that store the object have already moved to a later request. In Heron, a lagger needs to transfer a consistent state from other replicas in its partition, as described in the next section.

To reduce the probability that a replica r_i lags behind, after coordinating with a majority of replicas in another partition, replicas wait an additional small delay to allow r_i to catch up, should r_i be slower than a majority of replicas in its partition. We show experimentally that waiting for a small fraction of the time needed to execute a multi-partition request is enough to practically avoid laggers.

State synchronization

When a replica realizes that it lags behind other replicas in its partition, the replica requests a state transfer to the other replicas in its partition. A replica finds out that it is lagging behind when it reads remote objects with timestamps higher than the timestamp of the request the replica is currently executing. As explained in the previous section, a replica that lags behind other replicas needs to update its state from the state of other replicas in its partition because the replica cannot retrieve consistent object values.

To communicate state transfer requests, Heron replicas maintain the State Transfer Memory. The State Transfer Memory is an array of RDMA-registered buffers of size equal to the number of replicas in the partition. Each array entry stores two values: *req_tmp* and *status*. *req_tmp* is the timestamp of the request that the replica failed to execute. *status* is the stage of state transfer protocol: 0 when there is no state transfer in execution at the replica and 1 when the replica has requested a state transfer. In addition, replicas maintain a log record

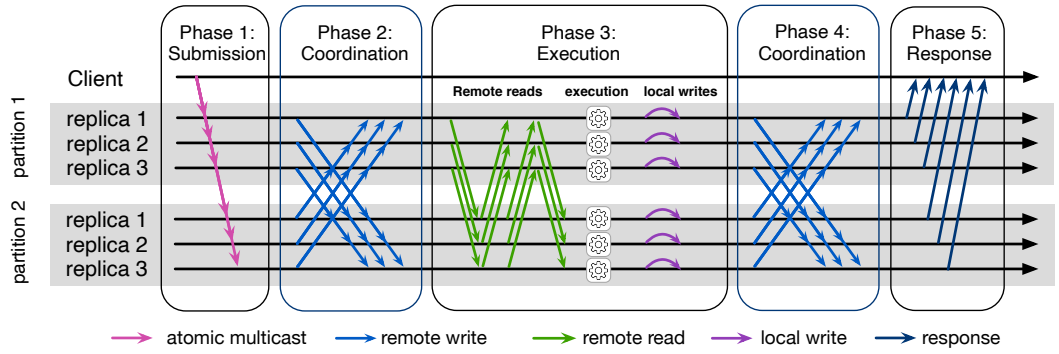

Figure 5.1. The lifespan of multi-partition requests in Heron.

of updated values while executing requests in normal execution. This log is used during state transfer to reduce the objects that must be synchronized.

5.1.2 Detailed algorithm

Processes in Heron use unique timestamps assigned by the atomic multicast protocol to infer the order of delivered messages. Timestamps are stored in $m.tmp$, to every delivered message m, such that for any two messages m and m', if $m < m'$ then $m.tmp < m'.tmp$. Algorithm 9 shows the coordination logic of Heron (see also Figure 5.1). Clients submit a request by atomically multicasting it to the destination partitions (line 3). Upon delivery of a request, a server process p first checks if the request must be skipped, which is the case when a client has received state updates through state transfer after a failure (lines 6–7). In case the request is single partition, process p skips coordinations and executes the request right away (lines 8–10). Otherwise, p executes the coordination phase by writing coordination messages on processes involved in the request and waiting for coordination messages from a majority of processes in each involved partition (lines 11–14). Next, the request is executed (lines 15–16). This includes reading states locally and remotely and writing new values locally using *read_objects* and *write_objects* procedures. In phase 4, p goes through another round of coordination, similar to phase 2 (lines 17–20). Finally, p responds to the client (lines 21–22).

Algorithm 10 presents the procedures to perform object reads and writes. The procedures are invoked in Algorithm 9 from the *exec_callback* method. The *read_objects* procedure reads values for objects in *read_set*. For each object in *read_set*, process p finds out the object's partition by querying an application-defined partitioning method (lines 2–3). Next, for remote objects that the parti-

Algorithm 9 Coordination

```
1:  Client to issue request r
2:    /* phase 1: multicast */
3:    multicast(r, r.dest)

4:  Process p in group g to execute request r
5:    upon delivery of request r do
6:      if r.tmp ≤ last_req then return                    {skip execution}
7:      else last_req ← r.tmp
8:      if r.dest.size = 1 then                            {no coordination required}
9:        response ← exec_callback(r)
10:       return response to client
11:     /* phase 2: coordination */
12:     for all h ∈ r.dest, for each q ∈ h do
13:       write_coord(q, p, ⟨r.tmp, 1⟩)
14:     wait until ∀h ∈ r.dest, ∃ majority of q ∈ h :
              coord_mem[h][q].tmp = r.tmp
15:     /* phase 3: execution */
16:     response ← exec_callback(r)
17:     /* phase 4: coordination */
18:     for all h ∈ r.dest, for each q ∈ h do
19:       write_coord(q, p, ⟨r.tmp, 2⟩)
20:     wait until ∀h ∈ r.dest, ∃ majority of q ∈ h :
              (coord_mem[h][q].tmp = r.tmp and
              coord_mem[h][q].state = 2) or
              coord_mem[h][q].tmp > r.tmp
21:     /* phase 5: response */
22:     return response to client
```

Variables:

$r.tmp$: request timestamp

$r.dest$: destination partitions request r is addressed to

$last\_req$: timestamp of the last executed request, initially 0

$coord\_mem[h][q]$: coordination memory entry for process q in partition h; each entry consists of a request timestamp and a state (1: delivered request, 2: finished execution), initially $\langle 0, 0 \rangle$

Methods:

$multicast(r, r.dest)$: atomically multicasts request r to $r.dest$

$write\_coord(q, p, v)$: remote write v to process p's entry in the $coord\_mem$ of process q.

$exec\_callback(r)$: application's execute callback method; it includes $read\_objs$ and $write\_objs$ procedures to perform reads and writes

Algorithm 10 Execution

```
 1: Procedure read_objects(r, read_set):
 2:   for oid in read_set do
 3:     h ← query_mapping(oid)
 4:     /* retrieve object address in remote processes */
 5:     if h ≠ g and ∀q ∈ h, !obj_map.contains(⟨oid, q⟩) then
 6:       for all q in h do
 7:         query_obj_addr(q, oid)
 8:       while not heard from majority of processes in h do
 9:         q, addr ← wait()
10:         obj_map.put(⟨oid, q⟩, obj_addr)
11:     /* read values */
12:     if h = g then                                          {local values}
13:       val ← obj_list.get(oid)
14:       r.set_value(oid, val)
15:     else
16:       while true do
17:         q ← rand_proc(h, r)
18:         addr ← obj_map.get(⟨oid, q⟩)
19:         if addr is null then
20:           continue
21:         result, val1, val2 ← remote_read(q, addr)
22:         if result is RDMA_EXCEPTION then                   {failed process}
23:           continue
24:         val ← null
25:         if val1.tmp < r.tmp and val1.tmp > val2.tmp then
26:           val ← val1
27:         else if val2.tmp < r.tmp then
28:           val ← val2
29:         if val is null then                                {state loss}
30:           invoke state transfer protocol
31:           return
32:         else r.set_value(oid, val)
33: Procedure write_objects(write_set):
34:   for ⟨oid, val⟩ in write_set do
35:     obj_list[oid] ← val
```

Variables:

obj_list: set of local objects

obj_map: map of $\langle oid, q \rangle$ to the address of object oid in process q

Methods:

$rand\_proc(h, r)$: choose a random process from h that coordinated in phase 2 for request r

$query\_mapping(oid)$: query the partition that stores object oid

$query\_obj\_addr(q, oid)$: query address of oid in the memory of process q

$remote\_read(q, addr)$: remotely read object from address $addr$ in memory of q

tion does not know the memory address where the object is stored, p queries the object location from processes in partition h and waits to hear from at least a majority of processes (lines 4–10). This ensures that process p knows the memory location of the object in at least one correct process from the remote partition.

Having the memory address of objects, process p is able to read object values for local objects (lines 11–14) and remote ones (lines 15–32). For remote reads, p randomly chooses a remote process in partition h (line 17). To ensure reading consistent values, the selected process must be among the ones that p has heard from in phase 2. If the object location in the selected process is unknown, p chooses another process (lines 18-20). Otherwise, p reads the object value. If the remote process has failed, p will find out about the failure through RDMA exceptions for the read operation and choose another process (lines 22–23).

To ensure linearizability, p must read valid object values only (lines 24–32). If p is not a lagger, then it finds the object value by choosing the valid value among the two available (duplicated) instances. The valid instance is one with smaller timestamp than the current request's timestamp and is the maximum among the two. If such a value is not found (i.e., both values have timestamp equal or bigger than request's timestamp), it implies that p must initiate the state transfer protocol. Otherwise, the value is valid and the request can be executed. After executing a request, a new object version is created (lines 33–35).

Algorithm 11 presents Heron's state transfer protocol. A replica initiates state transfer by remotely writing in a pre-assigned entry in the memory of all replicas in the partition (lines 2–4). Upon reading a state transfer request, a replica is deterministically selected for performing state transfer (line 10). Then, the states to be synchronized are specified (line 12) and the replica synchronizes the states (lines 13–15). At the end of the synchronization, the replica informs the other replicas in the partition about the completion of state transfer by updating the *request id* and *status* values in their memory (lines 16–17). *request id* specifies the last request that its state modifications are synchronized. Finally, p updates its $last\_req$ field to prevent executing earlier requests (line 6). In case the selected replica is suspected to have failed, another one is selected for state transfer (lines 18–19, line 9–10).

5.1.3 Correctness

In this section, we argue that Heron produces linearizable executions: For any execution σ of Heron, there is a total order π on client requests that (i) respects the semantics of the requests, as defined in their sequential specifications, and (ii) respects the real-time precedence of requests [8; 49].

Algorithm 11 State transfer

```
1:  Process p in group g to initiate recovery
2:    upon state transfer invocation on request r do
3:      for all q in g do
4:        write_state_transfer(q, ⟨r, 1⟩)
5:      wait on state_sync_mem[p].status to become 0
6:      last_req ← state_sync_mem[p].rid

7:  Process p in group g to handle recovery
8:    upon state transfer request from process q for request r do
9:      while true do
10:       proc ← deterministically choose a process
11:       if proc = p then
12:         objects ← log.get_objects(r.tmp, last_req)
13:         for obj in objects do
14:           addr ← obj_map.get(q, obj.id)
15:           remote_write(q, addr, obj)
16:         for all proc in g do
17:           write_state_transfer(proc, ⟨last_req, 0⟩)
18:       else
19:         wait on state_sync_mem[q].status to become 0 unless timeout expires
```

Variables

$state\_sync\_mem[q]$: state sync memory entry for process q; each entry consists of a request id rid and $status$

log: log of objects written/updated while executing requests

Methods

$write\_state\_transfer(q, \langle r, s \rangle)$: write state transfer request on process q for request r with status s

$get\_objects(r1.tmp, r2.tmp)$: returns objects written/updated from request $r1$ to request $r2$ (included)

$remote\_write(q, addr, v)$: remotely write value v to $addr$ on the memory of process q

Let π be a total order of requests in σ that respects $\prec$, the order atomic multicast induces on requests. To argue that π respects the semantics of requests, let C_i be the i-th request in π and p a process in partition x that executes C_i. We claim that when p executes C_i, all read operations issued by p as part of C_i result in values that reflect all requests that precede C_i and no value created by a request that succeeds C_i. We prove the claim by induction on i. For the base step, request C_0, the claim trivially holds for local reads, as objects are initialized correctly. Assume that p successfully reads an object from process q in partition y. Since p only accepts the remote read if the timestamp of the value read is smaller than the timestamp of C_0, p knows that q has not executed any later request that modifies the object read.

For the inductive step, assume the claim holds for $C_0, ..., C_{i-1}$. If p reads a local object, then the claim holds from the inductive hypothesis. Assume that p reads a remote object from process q in partition y. There are two cases to consider. When p reads the object, (a) q has already executed every request that precedes C_i, and (b) q has not executed any requests that succeed C_i. For (a), from the algorithm, p only issues a remote read operation for an object stored on q if q coordinated with p in phase 2. For (b), as in the base step, p only accepts the remote read if the timestamp of the value read is smaller than the timestamp of C_i. Thus, q did not execute any later request that modifies the object read by p when p reads the object from y.

We now argue that π respects the real-time precedence of requests in σ. Assume that C_i ends at a client before C_j starts at a client. 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_k and C_l be two consecutive requests in $C_j \prec ... \prec C_i$, where $C_k \prec C_l$. Thus, there is some partition x involved in C_k and C_l such that servers in x deliver first C_k and then C_l. Since servers execute one request at a time in the order they are delivered, it follows that C_k is executed before C_l by servers in x, and it cannot be that C_l ends before C_k starts. From a simple induction, it cannot be that $C_j \prec C_i$, and so, either $C_i \prec C_j$; or neither $C_i \prec C_j$ nor $C_j \prec C_i$.

5.2 Implementation

We implemented a prototype of Heron in Java. We use an open-source user-level library developed by IBM for RDMA communication [98] called jVerbs (DiSNI library v2.1).[1] jVerbs offers low latency overhead to applications running Java

[1] https://github.com/zrlio/disni

by exposing RDMA network hardware resources directly to the Java Virtual Machine. Heron relies on RamCast,[2] our shared-memory atomic multicast primitive for ordered delivery of requests introduced in Chapter 4. Heron's source code is publicly available.[3]

5.2.1 TPCC implementation

We implemented a Java version of TPCC that runs on top of Heron. TPCC is an established standard for evaluating the performance of storage and database systems. TPCC defines a transactional workload for a database system in a wholesale supplier company. The company has a possibly variable number of distributed warehouses (Warehouse table). Each warehouse has 10 districts (District table) and each district services 3,000 customers (Customer table). Warehouses maintain a stock of 100,000 items (Item and Stock tables). The customer orders (Order and New-Order tables) are also stored per order item (Order-Line table), and a history of customers orders are maintained (History table). There are 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%).

Each row in TPCC tables is an object in Heron. To allow processes to access remote objects, these objects must be stored in memory regions that are registered by RDMA device. Currently, Java does not support Value Types [89]. This prevents us from using Java List, for example, to store remotely accessible array of objects. One workaround is to store the data in Java's ByteBuffer. The serialized data can then be stored in RDMA-registered memories for remote access. Accessing serialized tables, locally or remotely, involves deserializing the data to retrieve values and serializing again in the case of data modification. The data in two tables, Stock and Customer, are stored serialized. These are tables that are accessed by remote processes while executing TPCC requests. Other tables are stored in memory using Java HashMap since they are not accessed remotely.

Each Heron partition stores one TPCC warehouse. The Warehouse and Item tables are replicated in all partitions, since they are not updated in the benchmark. Other tables are warehouse-specific and replicated in one partition. As shown in Figure 5.1, there is no remote writes while executing requests. This allows our TPCC implementation to partially execute transactions in some partitions. Partial execution refers to avoiding computations that results in modifica-

[2]https://github.com/longle255/libRamcastV3
[3]https://github.com/meslahik/heron

tion of objects that are not stored locally in the partition.

5.3 Evaluation

In this section, we motivate our experimental study (§5.3.1), describe the experiment's environment (§5.3.2), and discuss the results of our evaluation (§5.3.3-§5.3.5).

5.3.1 Roadmap

We seek to answer the following questions through three sets of experiments:

1. Performance (§5.3.3): What is the overall performance and scalability of Heron while running complex transactions (i.e., TPCC)? How does Heron's shared memory model compare to message passing-based scalable SMR systems?

2. Latency (§5.3.4): What is the latency of Heron's coordination? What is the latency of running TPCC transactions on Heron?

3. State transfer (§5.3.5): How long does it take for Heron to recover a left-behind replica? How to determine the efficient cut-off time for coordination?

5.3.2 Environment and configuration

We conducted all experiments in CloudLab [33] in XL170 nodes. Each node is equipped with one ten-core Intel E5-2640v4 processor running at 2.4GHz, 64 GB of main memory, and a Mellanox ConnectX-4 NIC. A 25-Gbps network link with around 0.1ms round-trip time connects all nodes running Ubuntu Linux 18.04 with kernel 4.15 and Oracle Java SE Runtime Environment 11. In all experiments, clients and servers are independent processes with in-memory storage. Clients submit requests in a closed-loop, that is, a client submits a request to servers and waits for a response before submitting the next request. Unless stated otherwise, each partition has 3 replicas. In all Heron experiments, clients measure latency as the interval between submitting a request and the response received from one server in each partition addressed by the request.

Figure 5.2. Performance of RamCast, Heron, TPCC, and TPCC local with increasing number of partitions.

5.3.3 Performance

The performance of Heron

Figure 5.2 shows the maximum throughput of 4 sets of TPCC experiments as we increase the number of warehouses from 1 to 16. In the first three sets, the ratio of single- and multi-partition requests is given by TPCC. In the last set of bars (Tpcc local), all requests are local. For 1WH experiments, Heron skips coordination since there is only one partition in the system.

The first set of bars shows the performance of RamCast, without coordination and execution. RamCast sports a close-to-linear scalability as we increase the number of warehouses. This is a promising result that sets the stage for fast coordination and execution. The second set of bars represents the performance of Heron with null requests. This helps understand the cost of coordination in Heron, without the overhead of request execution. From 1WH to 2WH, performance does not increase due to the overheard of coordination needed in 2WH. Performance increases by factors of 1.49x, 1.88x, and 1.61x thereafter. The third set of bars shows the performance of TPCC on Heron. As before, the performance of TPCC is the same for 1WH and 2WH. Performance for 4WH, 8WH, and 16WH increases by the factors of 1.54x, 1.74x, 1.50x, respectively.

In the above experiments, the performance improvement from 8 to 16 partitions is less pronounced than from 4 to 8 partitions. We attribute this to the network infrastructure of our testbed. According to Cloudlab documentation [24], XL170 nodes are connected via an experimental link to Mellanox switches in groups of 40 servers. Each of the groups' experimental switches are then connected to another Mellanox switch at 5x100Gbps. This means that above 40 nodes, there are always requests that go beyond the Top-Of-Rack switch to reach destination, with no bandwidth guarantees.

Finally, as a sanity check, we consider a workload with local-only TPCC trans-

Table 5.1. Scalability factor of different configurations.

Experiment	1 → 2	2 → 4	4 → 8	8 → 16
RamCast	1.37	1.70	2.22	1.83
Heron	1.05	1.49	1.88	1.61
TPCC	0.98	1.54	1.74	1.50
Local TPCC	1.90	1.98	2.09	2.01

actions. We modify the TPCC client code so that requests do not access objects in other partitions. In this case, we expect linear scalability since there in no cross-partition requests. The forth set of bars confirms this expectation while executing local TPCC workload.

Table 5.1 shows the scalability factor of different experiments. The scalability factor measures the increase in throughput from one configuration to another. Ideal scalability (scalability factor 2.0) is only possible in the absence of cross-partition requests, as in TPCC local. Likewise, limited improvements, if any, can be achieved from 1WH to 2WH setups. In all other cases, performance improves by a factor of at least 1.49x.

Heron vs. DynaStar

We now compare Heron to DynaStar, a message-passing scalable state machine replication system [51]. We choose DynaStar because it closely matches Heron execution model, it supports both single- and multi-partition requests, it has been shown to outperform other related systems, and it is available as opensource. Figure 5.3 shows peak performance and latency of both systems when executing TPCC as we increase the number of warehouses. In the 16WH configuration, we ran out of machines for DynaStar to deploy enough clients to saturate the system, which resulted in lower throughput and latency than expected. The performance results show that Heron outperforms DynaStar by an order of magnitude in all configurations considered. Heron improves performance by 17x in the 1WH experiment, up to 27x in the 16WH experiment. The latency results show that Heron has substantially lower latency than DynaStar which has 43.9x, 68.3x, 69.7x, and 72.0x higher latency than Heron for 1WH to 8WH, respectively.

There are three reasons for Heron's impressive performance. First, Heron directly benefits from efficient RDMA verbs, avoiding expensive message-passing primitives (i.e., no overhead with context switches and communication protocol stacks). This impacts both the coordination and the execution of application requests. Second, in Heron, multi-partition requests read remote objects

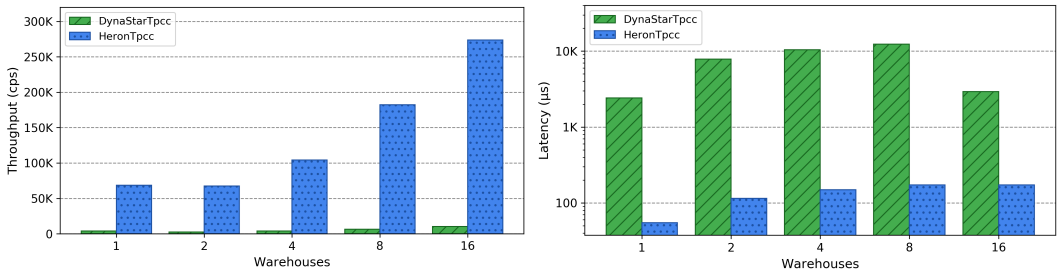

Figure 5.3. Performance and latency of Heron vs. DynaStar.

through RDMA verbs, while in DynaStar, the execution of a multi-partition request involves rounds of message exchanges to move objects from one partition to another. Third, Heron benefits from a carefully designed execution path. Optimizations include a manually (de)serialization of objects rather than using a serializer library, and storing strings as byte buffers as (de)serialization of Java Strings is quite expensive.

5.3.4 Latency

Latency without contention

Figure 5.4 shows the breakdown of the average latency when one client submits TPCC New Order requests in a closed loop. We consider a workload with a single client to avoid queuing effects due to contention. The breakdown shows the latency footprint of three stages of running a request on Heron. In this workload, Heron's coordination constitutes only about 2 microseconds of the whole latency of 35.4 microseconds, while ordering and execution take 18 and 16 microseconds, respectively.

We further study the latency of Heron for requests that target a fixed number of partitions. For that, we modify TPCC NewOrder transactions so that they access objects in the specified number of partitions. In the 1WH workload, there is no cross-partition requests: all requests are local and there is no coordination. In the 4WH workload, requests always target 4 partitions, accessing at least one object in each of these partitions.

From 1WH to 4WH, all stages of running a request become more expensive. For the ordering, the slight increase in latency is due to the higher number of partitions in the destination of the request. For the execution, the additional latency comes from the fact that more remote objects must be read per request. Coordination latency never goes above 3 microseconds in all workloads.

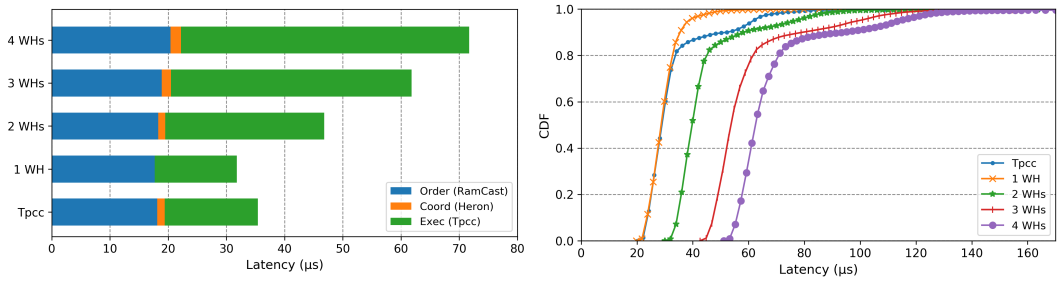

Figure 5.4. Heron's latency for single- and multi-partition requests with 1 client: breakdown of average latency (left) and cumulative distribution function (CDF) (right).

The CDF graph in Figure 5.4 reveals more insights about the latency of request execution. For 1WH, all requests are local so latency experiences little variation, with some outliers that constitute about 8% of the requests. In TPCC, about 10% of requests are multi-partition. This results in similar latencies as in 1WH workload for about 82% of the requests. Then, the outliers of single-partition requests show up until about 90% of latency values. Multi-partition requests show even higher latencies. A similar interpretation applies to latencies for other workloads.

The latency of TPCC transactions

Figure 5.5 shows the average latency of various TPCC transaction types. For each transaction type, one client submits that transaction type in a closed loop. The bars differentiate between latencies for single- and multi-partition transactions that expand to multiple partitions (New Order and Payment transactions). The blue bars show the average latency of single-partition transactions. The green bars show the additional latency added by multi-partition requests.

New Order and Payment transactions are heavy transactions. OrderStatus and Delivery transactions are local, light-weight transactions and their latencies are as low as 16.5 and 17.6 microseconds, respectively. StockLevel is a heavy local transaction that accesses items in the last 20 orders. StockLevel transactions are expensive because they access many items in a serialized table (i.e., Stock table), and the data must be deserialized, modified, and stored back serialized.

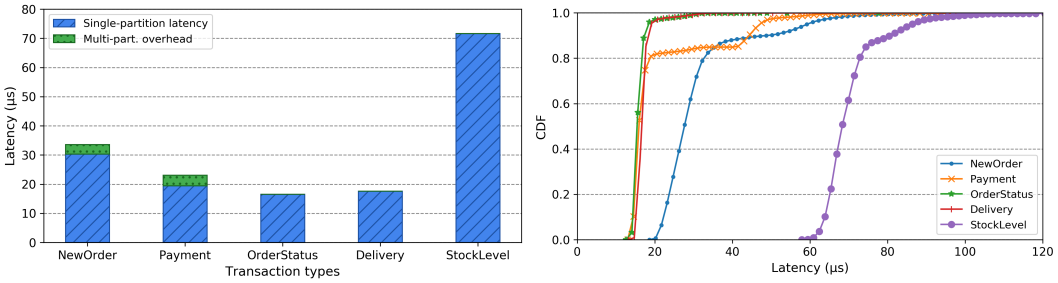

Figure 5.5. Latency of TPCC transactions: average latency of single- and multi-partition transactions (left) and cumulative distribution function (CDF) (right).

5.3.5 State transfer

The impact on latency of "waiting for all"

We first measure the impact of tentatively waiting for all replicas when coordinating. Table 5.2 shows the percentage of delayed transactions and the average delay in microseconds in four different configurations: 2 and 4 partitions, and 3 and 5 replicas per partition. A transaction is delayed at a replica if when the replica checks for a majority of coordination messages in its data structures, it does not already have messages from all replicas. The average delay is the amount of time the replica needs to wait to have coordination messages from all replicas, if the transaction is delayed. An important observation from the results is that very few transactions need to be delayed, in the worst case 8%, and the delay per transaction is a fraction of the average latency of a transaction. Since clients wait for a reply from each partition involved in a request, the perceived increase in latency by the client is given by the maximum delay among the partitions involved in the request. Moreover, only the second coordination phase needs to use this additional delay in order to keep replicas in sync.

In all configurations, the percentage of delayed transactions increases with the partition id, while the average delay decreases. This happens because each replica updates the coordination data structure in other replicas involved in a request in order from the smallest replica id to the largest replica id in the smallest partition id, then proceeds to the next partition id and so on. As a result, a replica in the first partition id (among those involved in the request) has higher chances of finding all coordination messages when it checks its data structure than replicas in partitions with higher id. However, the average delay decreases in replicas with larger id because it takes longer for these replicas to have all

Table 5.2. Delay of transactions due to waiting for all rather than a majority of replicas during coordination.

2 Partitions				
	3 replicas per partition		5 replicas per partition	
	max throughput: 53,340 tps average latency: 35.7 μs		max throughput: 42,658 tps average latency: 45 μs	
partition id	delayed transactions	avg delay	delayed transactions	avg delay
#1	1%	5.3 μs	2%	18.6 μs
#2	8%	4 μs	4%	9.3 μs
4 Partitions				
	3 replicas per partition		5 replicas per partition	
	max throughput: 92,808 tps average latency: 41.3 μs		max throughput: 73,724 tps average latency: 52.2 μs	
partition id	delayed transactions	avg delay	delayed transactions	avg delay
#1	1%	29.6 μs	3%	16 μs
#2	3%	11.8 μs	3%	11.1 μs
#3	3%	6.9 μs	3%	5.4 μs
#4	4%	2.1 μs	4%	8.8 μs

coordination messages, and so, the increase in latency is not so substantial as in replicas in partitions with smaller id.

State transfer latency

Figure 5.6 shows the latency of the state transfer for TPCC tables in logarithmic scale. For each transfer size, we show average latency (bars) and the standard deviation (whiskers). The standard deviation in all cases shows minimal deviation from the average latency except for the "Protocol" experiment. For state transfer, the data is transferred through RDMA writes with payloads of 32KBs (which has better performance than smaller payload sizes for the same amount of data [71]).

The "Protocol" bar shows the latency of state transfer for a null application, when no data is transferred. This represents the overhead of Heron's state transfer protocol without data exchange, and it amounts to two RDMA writes (i.e., one by the replica that requests the state transfer and the other by the replica that responds to this request). The next bars show the state transfer with various data sizes for two scenarios. The two scenarios differentiate between state transfer of serialized and non-serialized data. We chose 64KB data size as a representative

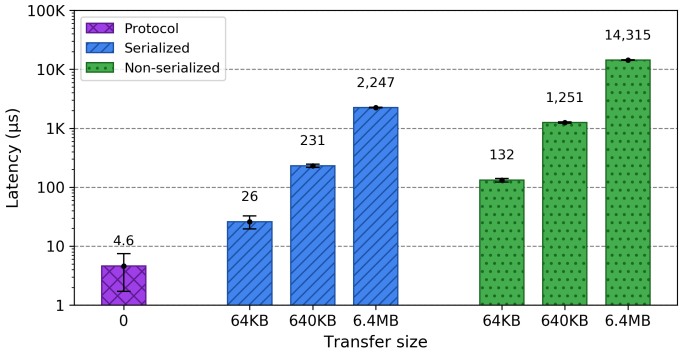

Figure 5.6. Latency of state transfer. Protocol shows latency of state transfer protocol without transferring any data. Other bars show state transfer for various data sizes.

of a small data size that must be transferred during the state sync, while 640KB and 6.4MB stands for state sync when 1% and 10% of a default TPCC table (i.e., Stock table) is modified and needs to be synced.

In the first scenario, we assume that only serialized data (e.g., TPCC Stock table) is transferred. In this case, state transfer includes writing the missing data to recipient's memory where the outdated data resides. The figure shows that for 64KB of data, it takes 26 microseconds for Heron to perform the state syncronization. The latency increases linearly with the data size for 640KB and 6.4MB of data, as expected.

In the second scenario, non-serialized data is transferred (e.g., TPCC Item table). In this case, state transfer includes serializing the data and remotely writing the data in a part of the receiver's memory. The receiver then deserializes the data and updates the application states accordingly. The results show that (de)serialization has a considerable degrading effect on the latency.

The time needed by a replica to catch up depends on how much it lags behind. If the replica misses a single request, then it will catch up in tens of microseconds, depending on how much data was updated in the missed request. In the worst case, upon recovering from a failure, a replica needs to transfer the complete state from another replica. In our prototype, a warehouse stores 137.69 MB worth of data, 105.3MB serialized and 32.39MB non-serialized.[4] This amounts to a transfer time of 109.4ms (36.9ms serialized, 72.5ms non-serialized).

[4]This represents some point during the execution, as some tables in TPCC increase constantly. The changes in size are minimum though and do not impact the state sync time significantly.

5.4 Conclusion

Microsecond latency applications are becoming the de facto standard for latency-critical services. This chapter presents Heron, the first scalable state machine replication system that targets microsecond latency applications. Heron's contributions include a novel shared-memory algorithm for coordinating linearizable execution of requests and a state synchronization protocol that recovers lagging replicas very quickly. We have implemented Heron and extensively evaluated its performance. The results show that Heron provides microsecond latency for coordinating strongly consistent executions and achieves more than tenfold improvement in the throughput of TPCC workloads in comparison to its competitors.

Chapter 6

Related work

In this chapter, we review selected publications in the research areas considered in this dissertation. The proposed systems build on prior works on atomic multicast protocols (§6.1), scalable state machine replication (§6.2), RDMA-based systems (§6.3), and distributed B-Tree algorithms (§6.4).

6.1 Atomic multicast

Atomic multicast is a well-studied problem. Skeen's algorithm (described in Section 4.1.2) is possibly the first atomic multicast algorithm. Even though it is not fault-tolerant, it is genuine: processes only communicate if they are in the destinations of the messages. Later timestamp-based genuine atomic multicast algorithms implemented fault-tolerant versions of Skeen's protocol. FastCast [25] speeds up the delivery of messages by overleaping some parts of the protocol (i.e., the order proposed by the leader and the consensus needed to decide on the proposed order). In good runs, FastCast delivers multi-group messages in 4 communication steps. White-Box Atomic Multicast [42] further improves latency with a protocol that combines Paxos and a fault-tolerant version of Skeen's protocol. White-Box Atomic Multicast delivers multi-group messages in 3 communication steps at the leaders of the involved groups and 4 communication steps at the followers. RamCast [71] improves on White-Box Atomic Multicast in that both leaders and followers can deliver a multi-group message in 3 communication steps.

Ring-based protocols [14; 30; 80] proposed a different approach to high throughput by propagating messages along a predefined ring overlay and ensuring atomic multicast properties by relying on this topology. However, ring-based algorithms are non-genuine: involved processes communicate with pro-

cesses outside the destination groups to deliver messages. The time complexity of these algorithms is proportional to the number of destination groups.

6.2 Scalable SMR

State machine replication (SMR) [62; 65; 66; 68; 91; 92] provides a general method for implementing fault-tolerant and strongly consistent distributed systems. In its essence, a combination of total order and deterministic execution of requests. Various attempts have been made to increase the performance of state machine replication. Kapritsos and Junqueira [60] propose to divide the ordering of requests between different clusters to optimize the ordering as it is fundamental for SMR. S-Paxos [17] avoids overloading the leader process of Paxos [69], which would otherwise turn it into a bottleneck.

Some works propose a multi-threaded, yet deterministic implementation of SMR. In [91], the receipt and dispatching of requests are handled concurrently, while requests are executed sequentially. In CBASE [65], deterministic execution is guaranteed, although requests that are safe to execute concurrently (e.g., read-only requests) are executed in parallel. In Eve [62], requests are tentatively executed in parallel and then replicas verify whether they reached a consistent state. In the case of disagreement, commands are rolled back and re-executed sequentially.

Some database replication systems target high throughput by relaxing consistency, that is, they do not ensure linearizability. In deferred-update replication [23; 63; 94; 95], replicas immediately commit read-only transactions, which may result in non-linearizable executions. Such systems ensure serializability [12] or snapshot isolation [73], which do not take into account real-time order of various requests among different clients. Services that require linearizability cannot be implemented with such techniques.

State partitioning has been investigated to make linearizable systems scalable [13; 27; 41; 50; 78]. In a partitioned state machine system, there are two general solutions to handle multi-partition requests. The first solution is to weaken the guarantees of requests that access multiple partitions. Facebook's Tao [21] is a distributed data store that explicitly favors efficiency and availability over consistency. Scatter [41] is a scalable key-value store that is linearizable within a given key but not linearizable for multi-key application transactions. Calvin [101] is a transaction scheduling layer over non-transactional storage systems that provides strong consistency and ACID transactions. Its deterministic locking order allows high throughput for multi-partition requests.

Another solution is to provide strong consistency guarantees at the cost of a more complex execution path for requests that involve multiple partitions. Marandi et al. [78] propose a variant of SMR in which data items are partitioned but requests have to be totally ordered and with the limitation that a partition cannot access objects in other partitions. S-SMR [13] maintains a statically partitioned state machine. Upon delivery of a multi-partition request, a copy of accessed states are exchanged across partitions to execute the request. Partitions coordinate their execution to prevent request interleaves that violate strong consistency. DS-SMR [50] extends S-SMR by allowing state migration across partitions in order to reduce multi-partition commands. Although DS-SMR implements repartitioning, it does not perform well in scenarios where the state cannot be perfectly partitioned. DynaStar [51] improves on DS-SMR by employing a graph partitioning technique to place states that are accessed together frequently in one partition. In DynaStar, multi-partition requests are executed by moving states to one partition, among those involved in the request, which then executes the request. Heron [36] largely outperforms DynaStar and similar message-passing systems through its shared-memory algorithm which reduces the latency of coordinating linearizable executions to some microseconds.

6.3 RDMA systems

Remote Direct Memory Access (RDMA) allows servers to directly access the memory of a remote server. Over the years, RDMA has become an active area of research for its high throughput, low latency, and low CPU overhead. RDMA is supported through three architectures, Infiniband [86], RoCE [11], and iWRAP [88], that share a common API. RDMA has previously been studied and deployed in a range of distributed services such as RPCs [99], key/value stores [31; 58; 82; 104], databases [18; 52], and distributed file systems [53; 74; 100; 105].

DaRPC [99] is an RDMA-based RPC framework that aims at saturating the network and the CPUs within a multi-core system. Pilaf [82] is a distributed in-memory key-value store that restricts the use of RDMA to read-only requests and handles all other requests through messaging. FaRM [31] proposes a distributed computing platform that exposes memory of a cluster of machines as a shared address space and provides the transactional interface for applications to access the shared memory. It implements a key-value store on top and uses RDMA reads for GETs and RDMA writes for PUTs. HERD [58] focuses its design on reducing network round trips through using RDMA writes and involving server CPUs for executing the requests. Clients use one-sided RDMA writes to

relay requests, including GETs, to servers which poll per-client buffers to process requests. NVFS [53] leverages byte-addressable NVM and RDMA network to provide a novel design of Hadoop distributed file system. Octopus [74] is a distributed, shared persistent memory file system that redesigns the file system internal mechanisms by a combination of NVM and RDMA features and avoiding the isolation of file system and network layers. Kalia et al. [59] claim that low-level details are significantly important for RDMA system design and propose optimization guidelines to enhance the performance of RDMA system. We have applied many of the mentioned best practices in the implementation of RamCast and Heron.

RDMA's shared memory model is fundamentally different from TCP's message passing model. This requires revising the design and implementation of distributed building blocks such as consensus and group communication. DARE [87] is a crash-tolerant replication protocol that proposes a novel algorithm based on RDMA for replicating states. The consensus leader responds to read requests and replicates requests to its follower with RDMA one-sided write operations. DARE makes use of RDMA permission semantics when changing leaders. APUS [102] is another leader-based consensus protocol based on RDMA networks. It intercepts inbound socket calls on the leader host and denotes these calls as a consensus request, so it does not require modifying applications for integration. A leader process executes the request and replicates the log entry on followers using RDMA writes. Derecho [54] is a library that allows structuring applications into shards and replicating them. Updates occur with a variation of Paxos, while queries exploit a new form of snapshot isolation. The dynamic membership tracking uses virtual synchrony. Even though Derecho organizes processes into subgroups and shards, it does not offer any abstraction that provides total order for operations involving multiple shards. Mu [1] implements Protected Memory Paxos [2], a consensus algorithm that, in normal execution, uses one RDMA write to replicate a consensus request. Mu colocates the client and the leader roles of Paxos for optimizing latency and makes use of memory protection semantics of RDMA for leader change. Velos [43] extends Mu and proposes a leader-based consensus algorithm that relies solely on one-sided RDMA verbs.

6.4 B-Trees

The organization and maintenance of large ordered indexes based on B-tree date back to the 70s [10]. This was later followed by efforts to introduce concurrency in the execution of B-tree operations. Investigation into the performance of con-

current tree algorithms [56; 97] showed that B-link trees [72] provide the best performance for most operations. A distributed dictionary based on a distributed B-link tree was introduced later [55]. An extension of B+Tree in P2P approaches for multi-dimensional information proposed in [16].

There are several studies investigating distributed B-trees. Boxwood [75] studies the possibility of having a high-level scalable data structure as the fundamental storage infrastructure. It is shown that there is no universal abstraction that fits all needs. Mitchel et al. [83] introduced a cell distributed B-tree store. Their model explores the possibility of using the potential network capabilities when the processor becomes the bottleneck. HyperDex [35] is another distributed data store that provides a new search primitive for retrieving objects by secondary attributes. It statically maps objects to servers according to object values. Objects are duplicated to increase the performance of queries for an index with the cost of slower update operations. Aguilera et al. [4] implemented a distributed B-tree using Sinfonia [6], a distributed data sharing service. They use distributed transactions to make changes to B-tree nodes. The overall throughput of the proposed system is limited due to a large number of aborts in their model. Sowell et al. [96] extended the previous work to unify online and analytics systems. Their model is a multi-version tree with snapshots to increase the performance at the cost of weaker consistency guarantees. None of these implementations are open source, which prevented us from comparing them with DynaTree presented in Chapter 3. Aguilera et al. [5] introduced a distributed balanced tree in the core of their storage engine for Yesquel. A balanced tree differs from a B-tree to some extent, for example, by providing load balance rather than size balance. Yesquel does not scale for insert operations. Even though Yesquel's design supports replication and it is open source, the available implementation does not include replication, which prevented comparison to DynaTree.

Chapter 7

Conclusion

Many modern online applications require performance scalability and high availability while operating at low latency. Designing systems that combine scalability and fault tolerance where service latency is within a few microseconds, however, is challenging. For that, we targeted scaling state machine replication, a popular approach to high availability. State-machine replication provides configurable fault tolerance and strong consistency among independent replicas, so that users of a replicated service are unaware that multiple copies exist. However, since all replicated nodes must execute the same sequence of commands, performance scalability is limited. With this in mind, we studied scalable state machine replication systems by developing a practical complex application on top of such a system. Next, we designed, implemented, and extensively evaluated solutions, over the shared memory system model, to reduce the latency of such systems and improve their performance. Our solutions delivered promising performance improvements over message-passing solutions.

Two main objectives shaped the flow of our research: Our first goal was to study the implications of developing distributed applications over S-SMR systems. To this end we studied the challenges involved in developing a complex data structure like B+Tree and devised solutions to overcome them. Our findings and observations are mostly general, and they can be applied to other data structures and applications that seek high performance.

As our second goal, we strove for improving the performance of S-SMR systems and reducing their latency in order to serve latency-critical applications. We first looked into reducing the latency of atomic multicast primitive. Considering the high potential of RDMA technology for low latency communication, we developed a novel shared-memory atomic multicast algorithm that reduces the number of communication delays to 3 communications. Next, we revised

S-SMR system design to operate on the shared-memory model to coordinate and execute distributed operations using one-sided RDMA primitives to prevent object migration. In the next section, we briefly overview our findings and lessons learnt throughout this study.

7.1 Research assessment

This dissertation presents three contributions: (i) a distributed B+Tree algorithm that is modeled over scalable state machine replication, (ii) a novel atomic multicast algorithm that operates and communicates using shared memory model, and (iii) a new scalable state machine replication system that uses shared memory model to improve the performance of S-SMR systems. In the following, we review and discuss the most important aspects of these contributions.

DynaTree. Most scalable and fault-tolerant distributed systems use sharding and replication as their primary mechanisms. Recent studies on scalable state machine replication systems have shown promising results for executing distributed requests while maintaining expected consistency of SMR systems. We look into the difficulties of developing complex applications for S-SMR systems while developing DynaTree, a distributed B+tree that is developed using state-of-the-art S-SMR systems. It is demonstrated that constructing a complex data structure like B+tree in partitioned SMR entails a number of difficulties. Before a request is executed, clients must identify the data and partitions accessed. This is required to ensure that only the necessary partitions are involved in the execution of a request. To meet this requirement, DynaTree clients cache the tree's inner nodes in a lazy manner. Partitions verify the validity of the cached information before executing a request. The key contribution of DynaTree is to present a practical development of a distributed application over S-SMR systems. DynaTree satisfies the requirement of identifying objects accessed during execution of a request. We describe the design and implementation of DynaTree, and present a detailed performance evaluation of the tree operations using various workloads.

RamCast. Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available strongly consistent distributed systems. In order to decrease the performance of atomic multicast, we use RDMA technology, which provides low-latency communication through its shared memory primitives. Skeen's original atomic multicast algorithm [19] is revised and extended to support the shared memory model and sustain server crashes. The

result is RamCast, the first genuine atomic multicast protocol customized for the shared-memory model. RamCast leverages the permission mechanism of RDMA's write operation to reduce the number of communication steps. RamCast delivers single-destination requests after 2 RDMA write delays, at the leader process, and multi-destination requests after 3 RDMA write delays, at both the leader and followers. The algorithm has been proved correct, implemented, and extensively evaluated under a large range of parameters.

Heron. For latency-critical services, microsecond latency is becoming the standard. Recent proposals that have extended state machine replication with sharding, to overcome its limited performance scalability, do not perform well due to the data migration needed to execute multi-partition requests. This data exchange during request execution results in substantial overhead. To reduce the latency of executing multi-partition requests in such systems, we introduce Heron, the first scalable state machine replication system that targets microsecond latency applications. Heron uses RDMA read primitive to obtain the value of remote objects instead of migrating them between partitions. Heron also provides a state synchronization protocol that recovers replicas that are lagging behind and therefore have lost the values of remote objects. Heron has been implemented and its performance has been thoroughly evaluated.

7.2 Future directions

The main objective of this thesis is to investigate strongly consistent replicated systems that provide scalable performance. We discuss some of the possible future research directions.

Dynamic partitioning for scalable state machine replication. Recent research on S-SMR systems based on message passing [50; 51] have shown that a dynamic partitioning scheme helps to improve the performance of S-SMR systems. The performance results reveal that the performance of a partitioned system heavily depends on the partitioning of the data. In order to scale, most requests must involve a single shard, and the load must be balanced across shards. A dynamic partitioning scheme puts states commonly accessed together in the same partition, which significantly improves scalability. One solution is to build a workload graph on-the-fly and use an optimized partitioning of the workload graph to decide how to move state variables efficiently. The importance of locality is also present in RDMA-based communications, in which accessing local memory is 23× faster than remote memory [31]. It is worth studying efficient

techniques for dynamic partitioning data in shared-memory S-SMR systems.

RDMA-based distributed key-value stores. Over the past few years, several studies have targeted the development of efficient key-value stores using RDMA. These systems use RDMA primitives very different from each other to perform put and get operations. The lack of powerful RDMA operations, as mentioned in [58], is the main challenge for all these systems. An RDMA operation can only read or write a remote memory location. More complex operations are not possible, such as dereferencing and following a pointer in remote memory. Pilaf [82] restricts the use of RDMA to read-only requests and handles all other requests through messaging. FaRM [31] implements a key-value store on top of its distributed computing platform, which performs (several) RDMA reads to retrieve remote values. In contrast, HERD [58] found that the delay of RDMA writing is lower than that of reading. It focuses its design on reducing network round trips through RDMA writes and involving server CPUs for retrieving values written remotely on the local memory. As one of the directions for expanding this research path, one can study the proposed systems and compare their performance for various workloads and scenarios.

RDMA-based Byzantine Consensus In the context of Byzantine consensus protocols, RDMA has received little attention. Several replication protocols based on RDMA have been proposed, but none tolerate Byzantine failures. One can build up a consensus protocol on the existing BFT abstractions to tolerate malicious behavior.

Bibliography

[1] Aguilera, M. K., Ben-David, N., Guerraoui, R., Marathe, V. J., Xygkis, A. and Zablotchi, I. [2020]. Microsecond consensus for microsecond applications, *OSDI*.

[2] Aguilera, M. K., Ben-David, N., Guerraoui, R., Marathe, V. and Zablotchi, I. [2019]. The impact of rdma on agreement, *PODC*.

[3] Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H. and Toueg, S. [2001]. Stable leader election, *DISC*.

[4] Aguilera, M. K., Golab, W. M. and Shah, M. A. [n.d.]. A practical scalable distributed b-tree, *VLDB 2008* .

[5] Aguilera, M. K., Leners, J. B. and Walfish, M. [n.d.]. Yesquel: scalable sql storage for web applications, *SOSP 2015, Monterey, CA, USA, October 4-7*.

[6] Aguilera, M. K., Merchant, A., Shah, M. A., Veitch, A. C. and Karamanolis, C. T. [n.d.]. Sinfonia: a new paradigm for building scalable distributed systems, *SOSP 2007, Stevenson, Washington, USA, October 14-17*.

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

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

[9] Barret, P., Hilborne, A., Bond, P., Seaton, D., Verissimo, P., Rodrigues, L. and Speirs, N. [1990]. The delta-4 extra performance architecture (xpa), *FTCS*.

[10] Bayer, R. and McCreight, E. M. [1972]. Organization and maintenance of large ordered indexes, *Acta Informatica* **1**(3): 173–189.

[11] Beck, M. and Kagan, M. [2011]. Performance evaluation of the rdma over ethernet (roce) standard in enterprise data centers infrastructure, *DC-CaVES*.

[12] Bernstein, P. A., Hadzilacos, V. and Goodman, N. [1987]. *Concurrency control and recovery in database systems*, Vol. 370, Addison-wesley Reading.

[13] Bezerra, C. E. B., Pedone, F. and van Renesse, R. [n.d.]. Scalable state-machine replication, *DSN 2014, Atlanta, GA, USA, June 23-26*.

[14] Bezerra, C. E., Cason, D. and Pedone, F. [2015]. Ridge: high-throughput, low-latency atomic multicast, *SRDS*, IEEE, pp. 256–265.

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

[16] Bianchi, S., Felber, P. and Potop-Butucaru, M. G. [2009]. Stabilizing distributed r-trees for peer-to-peer content routing, *IEEE Transactions on Parallel and Distributed Systems* **21**(8): 1175–1187.

[17] Biely, M., Milosevic, Z., Santos, N. and Schiper, A. [2012]. S-paxos: Offloading the leader for high throughput state machine replication, *2012 IEEE 31st Symposium on Reliable Distributed Systems*, IEEE, pp. 111–120.

[18] Binnig, C., Crotty, A., Galakatos, A., Kraska, T. and Zamanian, E. [2015]. The end of slow networks: It's time for a redesign, *arXiv preprint arXiv:1504.01048* .

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

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

[21] 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, *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pp. 49–60.

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

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

[24] Cloudlab [2021]. Hardware specification.
URL: *http://docs.cloudlab.us/hardware.html*

[25] Coelho, P. R., Schiper, N. and Pedone, F. [2017]. Fast atomic multicast, *DSN*.

[26] Coelho, P. R., Schiper, N. and Pedone, F. [n.d.]. Fast atomic multicast, *DSN 2017, Denver, CO, USA, June 26-29*.

[27] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P. et al. [2013]. Spanner: Google's globally distributed database, *ACM Transactions on Computer Systems (TOCS)* **31**(3): 1–22.

[28] Cowling, J. and Liskov, B. [2012]. Granola: Low-overhead distributed transaction coordination, *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX, Boston, MA, USA.

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

[30] Delporte-Gallet, C. and Fauconnier, H. [2000]. Fault-tolerant genuine atomic multicast to multiple groups., *OPODIS*, Citeseer.

[31] Dragojević, A., Narayanan, D., Castro, M. and Hodson, O. [2014]. Farm: Fast remote memory, *NSDI*.

[32] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S. and Mishra, P. [2019a]. The design and operation of CloudLab, *USENIX-ATC*.

[33] Duplyakin, D., Ricci, R., Maricq, A., Wong, G., Duerig, J., Eide, E., Stoller, L., Hibler, M., Johnson, D., Webb, K., Akella, A., Wang, K., Ricart, G., Landweber, L., Elliott, C., Zink, M., Cecchet, E., Kar, S. and Mishra, P. [2019b]. The design and operation of CloudLab, *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pp. 1–14.

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

[35] Escriva, R., Wong, B. and Sirer, E. G. [n.d.]. Hyperdex: a distributed, searchable key-value store, *SIGCOMM '12*.

[36] Eslahi-Kelorazi, M., Le, L. H. and Pedone, F. [2022]. Heron: Scalable state machine replication on shared memory, *In review*.

[37] Esposito, E. G., Coelho, P. and Pedone, F. [2018]. Kernel paxos, *SRDS*, IEEE.

[38] Fischer, M. J. [1983]. The consensus problem in unreliable distributed systems (a brief survey), *International conference on fundamentals of computation theory*, Springer, pp. 127–140.

[39] 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.

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

[41] Glendenning, L., Beschastnikh, I., Krishnamurthy, A. and Anderson, T. [2011b]. Scalable consistency in scatter, *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pp. 15–28.

[42] Gotsman, A., Lefort, A. and Chockler, G. [2019]. White-box atomic multicast, *DSN*.

[43] Guerraoui, R., Murat, A. and Xygkis, A. [2021]. Velos: One-sided paxos for rdma applications, *arXiv preprint arXiv:2106.08676* .

[44] Guerraoui, R. and Schiper, A. [2001]. Genuine atomic multicast in asynchronous distributed systems, *Theor. Comput. Sci.* **254**(1-2): 297–316.

[45] 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.

[46] Hadzilacos, V. and Toueg, S. [1993]. Fault-tolerant broadcasts and related problems, *in* S. J. Mullender (ed.), *Distributed Systems*, Addison-Wesley, chapter 5, pp. 97–145.

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

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

[49] Herlihy, M. and Wing, J. M. [1990b]. Linearizability: A correctness condition for concurrent objects, *ACM Trans. Program. Lang. Syst.* **12**(3): 463–492.

[50] Hoang, L. L., Bezerra, C. E. B. and Pedone, F. [n.d.]. Dynamic scalable state machine replication, *DSN 2016, Toulouse, France, June 28 - July 1*.

[51] Hoang, L. L., Fynn, E., Eslahi-Kelorazi, M., Soule, R. and Pedone, F. [n.d.]. Dynastar: Optimized dynamic partitioning for scalable state machine replication, *ICDCS 2019*.

[52] Huang, B., Jin, L., Lu, Z., Yan, M., Wu, J., Hung, P. C. and Tang, Q. [2019]. Rdma-driven mongodb: An approach of rdma enhanced nosql paradigm for large-scale data processing, *Information Sciences* **502**: 376–393.

[53] Islam, N. S., Rahman, M. W., Jose, J., Rajachandrasekar, R., Wang, H., Subramoni, H., Murthy, C. and Panda, D. K. [2012]. High performance rdma-based design of hdfs over infiniband, *SC*, IEEE.

[54] Jha, S., Behrens, J., Gkountouvas, T., Milano, M., Song, W., Tremel, E., Renesse, R. V., Zink, S. and Birman, K. P. [2019]. Derecho: Fast state machine replication for cloud services, *ACM Transactions on Computer Systems (TOCS)* **36**(2): 1–49.

[55] Johnson, T. and Colbrook, A. [n.d.]. A distributed data-balanced dictionary based on the b-link tree, *Proceedings of the 6th International Parallel Processing Symposium, Beverly Hills, CA, USA, March 1992*.

[56] Johnson, T. and Shasha, D. E. [n.d.]. A framework for the performance analysis of concurrent b-tree algorithms, *Symposium on Principles of Database Systems, April 2-4, 1990, Nashville, Tennessee, USA*.

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

[58] Kalia, A., Kaminsky, M. and Andersen, D. G. [2014]. Using rdma efficiently for key-value services, *SIGCOMM*.

[59] Kalia, A., Kaminsky, M. and Andersen, D. G. [2016]. Design guidelines for high performance {RDMA} systems, *USENIX-ATC*.

[60] Kapritsos, M. and Junqueira, F. P. [n.d.]. Scalable agreement: Toward ordering as a service., *HotDep 2010*.

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

[62] Kapritsos, M., Wang, Y., Quema, V., Clement, A., Alvisi, L. and Dahlin, M. [2012b]. All about eve:{Execute-Verify} replication for {Multi-Core} servers, *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pp. 237–250.

[63] Kobus, T., Kokocinski, M. and Wojciechowski, P. T. [2013]. Hybrid replication: State-machine-based and deferred-update replication schemes combined, *2013 IEEE 33rd International Conference on Distributed Computing Systems*, IEEE, pp. 286–296.

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

[65] Kotla, R. and Dahlin, M. [2004b]. High throughput byzantine fault tolerance, *International Conference on Dependable Systems and Networks, 2004*, IEEE, pp. 575–584.

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

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

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

[69] Lamport, L. [n.d.]. Paxos made simple, fast, and byzantine, *OPODIS 2002, Reims, France, December 11-13*.

[70] Le, L. H. [2020]. *Scaling State Machine Replication*, PhD thesis, Universita della Svizzera italiana, Switzerland.

[71] Le, L. H., Eslahi-Kelorazi, M., Coelho, P. and Pedone, F. [2021]. Ramcast: Rdma-based atomic multicast, *Proceedings of the 22nd International Middleware Conference*, pp. 172–184.

[72] Lehman, P. L. and Yao, S. B. [1981]. Efficient locking for concurrent operations on b-trees, *ACM Trans. Database Syst.* **6**(4): 650–670.

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

[74] Lu, Y., Shu, J., Chen, Y. and Li, T. [2017]. Octopus: an rdma-enabled distributed persistent memory file system, *USENIX-ATC*.

[75] MacCormick, J., Murphy, N., Najork, M., Thekkath, C. A. and Zhou, L. [n.d.]. Boxwood: Abstractions as the foundation for storage infrastructure, *OSDI 2004, San Francisco, California, USA, December 6-8*.

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

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

[78] Marandi, P. J., Primi, M. and Pedone, F. [2011a]. High performance state-machine replication, *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, IEEE, pp. 454–465.

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

[80] Marandi, P. J., Primi, M. and Pedone, F. [2012b]. Multi-ring paxos, *DSN*, IEEE.

[81] Marandi, P., Primi, M. and Pedone, F. [2011b]. 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.

[82] Mitchell, C., Geng, Y. and Li, J. [2013]. Using one-sided {RDMA} reads to build a fast, cpu-efficient key-value store, *USENIX-ATC*.

[83] Mitchell, C., Montgomery, K., Nelson, L., Sen, S. and Li, J. [n.d.]. Balancing CPU and network in the cell distributed b-tree store, *USENIX ATC 2016, Denver, CO, USA, June 22-24*.

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

[85] Oracle [2020]. Berkeley db.
URL: *https://www.oracle.com/database/technologies/related/berkeleydb.html*

[86] Pfister, G. F. [2001]. An introduction to the infiniband architecture, *High performance mass storage and parallel I/O* **42**(617-632): 102.

[87] Poke, M. and Hoefler, T. [2015]. Dare: High-performance state machine replication on rdma networks, *HPDC*.

[88] Rashti, M. J. and Afsahi, A. [2007]. 10-gigabit iwarp ethernet: comparative performance analysis with infiniband and myrinet-10g, *IPDPS*, IEEE, pp. 1–8.

[89] Rose, J., Goetz, B. and Steele, G. [2014]. Java value types.
URL: *http://cr.openjdk.java.net/ jrose/values/values-0.html*

[90] 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.

[91] Santos, N. and Schiper, A. [n.d.]. Achieving high-throughput state machine replication in multi-core systems, *ICDCS 2013*.

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

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

[94] Sciascia, D., Pedone, F. and Junqueira, F. [2012]. Scalable deferred update replication, *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, IEEE, pp. 1–12.

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

[96] Sowell, B., Golab, W. M. and Shah, M. A. [n.d.]. Minuet: A scalable distributed multiversion b-tree, *VLDB 2012* .

[97] Srinivasan, V. and Carey, M. J. [1993]. Performance of B+ tree concurrency algorithms, *VLDB J.* **2**(4): 361–406.

[98] Stuedi, P., Metzler, B. and Trivedi, A. [2013]. jverbs: ultra-low latency for data center applications, *SoCC*.

[99] Stuedi, P., Trivedi, A., Metzler, B. and Pfefferle, J. [2014]. Darpc: Data center rpc, *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–13.

[100] Stuedi, P., Trivedi, A., Pfefferle, J., Stoica, R., Metzler, B., Ioannou, N. and Koltsidas, I. [2017]. Crail: A high-performance i/o architecture for distributed data processing., *IEEE Data Eng. Bull.* **40**(1): 38–49.

[101] 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*, pp. 1–12.

[102] Wang, C., Jiang, J., Chen, X., Yi, N. and Cui, H. [2017a]. Apus: Fast and scalable paxos on rdma, *Proceedings of the 2017 Symposium on Cloud Computing*, pp. 94–107.

[103] Wang, C., Jiang, J., Chen, X., Yi, N. and Cui, H. [2017b]. Apus: Fast and scalable paxos on rdma, *SoCC*.

[104] Wei, X., Shi, J., Chen, Y., Chen, R. and Chen, H. [2015]. Fast in-memory transaction processing using rdma and htm, *SOSP*.

[105] Wu, J., Wyckoff, P. and Panda, D. [2003]. Pvfs over infiniband: Design and performance evaluation, *ICPP*, IEEE.