Scaling Strongly Consistent Replicated Systems

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
Leandro Pacheco de Sousa

under the supervision of
Fernando Pedone

February 2023

Dissertation Committee

Cesare Pautasso	Università della Svizzera italiana
Patrick Eugster	Università della Svizzera italiana
Pascal Felber	Université de Neuchâtel
Rüdiger Kapitza	Technische Universität Braunschweig
Tony Cortes	Universitat Politécnica de Catalunya

Dissertation accepted on 8 February 2023

Research Advisor	PhD Program Director
Fernando Pedone	**Walter Binder and 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.

Leandro Pacheco de Sousa
Lugano, 8 February 2023

To my family and friends

It is looking at things for a long time that ripens you and gives you a deeper meaning.

Vincent Van Gogh

Abstract

With the recent rise of cloud infrastructure, geographically distributed applications have become a reality. Application developers can easily place servers close to end users, even if those users are spread around the globe. Still, designing applications that work correctly and perform well at such a large scale is a difficult endeavor, involving many tradeoffs. One such tradeoff is that between consistency and availability. While relaxing application guarantees might allow for better performance and availability, deciding which and how guarantees should be relaxed is a complex task, subject to application semantics and user expectations. On the other hand, strong consistency allows for developers to focus on core business logic and end users to interact with a more transparent system.

Reliability in a distributed application is achieved through replication. Full replication, where each server stores the whole application state, does not scale, as every server needs to apply every update. By partitioning the application state and letting only a subset of servers store it, performance can scale if the workload allows. Atomic multicast is a communication abstraction that can serve as a fundamental building block for partitioned applications. It allows for requests to be reliably sent to one or more groups of destinations, and ensures a partial ordering of deliveries, a property fundamental to consistent and scalable systems.

In this thesis, we focus on exploiting the atomic multicast abstraction to build strongly consistent geographically distributed applications. To that end, we first explore the design of a geographically scalable file system, GlobalFS. We then focus on the design of a latency efficient atomic multicast protocol, PrimCast. Finally, we propose linearizable atomic multicast, a stronger version of atomic multicast.

GlobalFS is a POSIX-like distributed file system that provides strong consistency guarantees and scales geographically by allowing for fast local operations while still providing consistent operations over the whole system. GlobalFS builds upon atomic multicast, providing four execution modes which are then used to execute each file system operation. We describe and implement a prototype of GlobalFS which we then use to validate the approach in a global deploy-

ment on Amazon’s EC2.

PrimCast is an atomic multicast protocol that allows for message delivery in three communication steps at any destination. PrimCast is genuine, that is, only sender and destinations take steps to deliver a message, a critical property in large scale deployments. We present the complete algorithm and proof of correctness for PrimCast. We also show how loosely synchronized clocks can be used to reduce the convoy effect that further delays messages under high system load. We implement a prototype of PrimCast and evaluate its performance under various scenarios.

Linearizable atomic multicast is an atomic multicast that provides linearizability for applications without the need for extra coordination among replicas. We show why classic atomic multicast by itself is not enough to ensure linearizability, describe a stronger ordering property that fixes the issue and show how a classic atomic multicast protocol can be modified to provide the stronger property.

Acknowledgements

I hereby express my gratitude to everyone who in any way supported me in the long journey of writing this thesis. Many people have contributed, directly or indirectly, to the work here presented.

I would like to, first of all, thank my advisor, Fernando Pedone, for the guidance and patience throughout the years it took me to complete my PhD. This thesis would definitely not exist without his support, encouragement and insights. I also would like to thank professors Cesare Pautasso, Patrick Eugster, Pascal Felber, Rüdiger Kapitza and Tony Cortes for the time dedicated to reviewing this thesis and for their invaluable feedback.

I'm extremely grateful to all the people who have been part of the Distributed Systems group at USI during my time there. In particular, to Daniele, Eduardo, Paulo, Sam and Enrique, with whom I've spent the most time with, my genuine thanks: your help and friendship were fundamental to me. To all the others (in a tentative chronological order): Amir, Alex, Parisa, Ricardo, Daniel, Tu, Long, Odorico, Edson, Tarcisio, Pietro, Mojtaba, Theo; Thank you for all our conversations over coffee, lunch or beer, I won't forget you.

I would like to thank my parents, Valdete and Antonio Carlos, for all their love and support. I simply would not be here if not for them.

Last but not least, to all the friends I've made during my years at Lugano: thank you for all the time we spent together.

Publications

L. Pacheco, D. Sciascia, F. Pedone. Parallel Deferred Update Replication. In *2014 IEEE 13th International Symposium on Network Computing and Applications*, NCA '14, pages 205–212. IEEE, 2014.

L. Pacheco, R. Halalai, V. Schiavoni, F. Pedone, E. Rivière, P. Felber. GlobalFS: A Strongly Consistent Multi-Site File System. In *2016 IEEE 35th Symposium on Reliable Distributed Systems*, SRDS '16, pages 147–156. IEEE, 2016.

S. Benz, L. Pacheco, F. Pedone. Stretching Multi-Ring Paxos. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 492–499. 2016.

T. Jepsen, L. Pacheco, M. Moshref, F. Pedone, R. Soulé. Infinite Resources for Optimistic Concurrency Control. In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, NetCompute '18, pages 26–32. 2018.

L. Pacheco, F. Dotti, F. Pedone. Strengthening Atomic Multicast for Partitioned State Machine Replication. In *11th Latin-American Symposium on Dependable Computing*, LADC '22. 2022.

Contents

Figures

Tables

Chapter 1

Introduction

Over the last few years, the ubiquity of cloud infrastructure has made the deployment of distributed systems commonplace. Large cloud providers such as Google and Amazon allow for users to distribute their computing resources all over the world. The motivation for doing so is clear: disasters might cause entire datacenters to go down and placing data close to end users is fundamental to achieving a responsive service. Deploying computing resources across multiple continents has never been easier, yet, designing systems that work well when deployed at such a global scale remains a challenge.

1.1 The tradeoff between strong and weak consistency

The design of distributed systems in general, and global distribution in special, involves many tradeoffs. One such tradeoff is between consistency and availability [42]. A system with strong consistency properties is more general and easier to reason about, providing clear semantics and guarantees to the user. On the other hand, weakening the guarantees provided by the system might allow for more performance and availability; as an example, a server temporarily partitioned from the system (e.g., due to a latency spike) might be allowed to produce inconsistent results instead of simply being unavailable. Deciding which and how guarantees should be relaxed can be a complex task, subject to application semantics and user expectations. Thus, in many situations, providing strong consistency can simplify the life of both application developers and end users. Developers can focus on the core logic of their business while end users can interact with a system with simpler semantics. In this thesis, we focus on the problem of providing strong consistency at a global scale.

1.2 Abstractions for scalable replicated applications

In order to tolerate failures, the components of a distributed system need to be replicated. For a replicated system to scale, the workload needs to be distributed among the replicas in the system. One common approach for such distribution is state partitioning, also known as sharding. In partitioned systems, each data item is stored by only a subset of the servers in the system. If an operation only accesses data in a single partition, servers from other partitions do not need to participate in its execution. For operations that read or write data from multiple partitions, some form of synchronization is needed to ensure that concurrent operations execute correctly. One way to ensure correctness consists in establishing a total order of execution among operations; the issue with this is that, with increasing load, totally ordering operations becomes a bottleneck. A more scalable alternative relies in establishing a partial order among operations; this is typically achieved by some form of ad hoc two-phase commit protocol. In this thesis, we instead focus on the atomic multicast abstraction. Atomic multicast is a communication protocol that allows a process to send a message to only a subset of the processes in the system. Atomic multicast ensures a partial ordering of message deliveries in the system, proving scalable strong ordering guarantees.

Distributed applications, instead of implementing ad hoc protocols to achieve replication, synchronization and scalability, typically rely on some form of distributed storage that provides the interface and guarantees needed by the application. Given the different needs of distributed applications, distributed storage can take many forms: relational and non-relational databases, key-value storage, coordination systems, distributed caches and many others. One such storage abstraction is that of the file system. File systems provide a familiar interface to both end users and application developers, and can greatly simplify the task of integrating existing applications into a distributed environment. In this thesis, we explore the design of a strongly consistent, geographically distributed file system, exploiting the partial ordering of atomic multicast for scalability.

1.3 Research contributions

This thesis provides three major contributions:

GlobalFS. We show how atomic multicast can be used as the fundamental building block for a geographically distributed file system. GlobalFS is a POSIX-like file system that can provide low latency for single-region local commands while allowing for consistent global operations across regions. File system operations are executed in one of four execution modes: (1) single-partition operations, (2) multi-partition uncoordinated operations, (3) multi-partition coordinated operations, and (4) read-only operations. We describe the implementation of GlobalFS and validate its design in a global deployment over Amazon's EC2. This contribution has been published in the 35th Symposium on Reliable Distributed Systems (SRDS '16). [77]

PrimCast. The design and implementation of GlobalFS relies on a non-genuine atomic multicast protocol, Multi-Ring Paxos [68]. A protocol is genuine if only sender and destination take steps to deliver a message. Genuineness is a fundamental property for scalability and low latency in global deployments. Even though Multi-Ring Paxos can be made to work at a global scale [12], it was originally designed for maximizing throughput inside a cluster. In a wide-area network, Multi-Ring Paxos requires complex tuning and exhibits higher latency than necessary by virtue of its communication patterns and non-genuineness. In this work, we propose PrimCast, a genuine atomic multicast protocol that can deliver messages at every destination in three communication steps in the absence of concurrent messages, and five otherwise. This is an improvement of one communication step over the state-of-the-art. We present the complete algorithm for PrimCast and its proof of correctness. We then show how loosely synchronized clocks can be used to reduce the convoy effect that further delays messages under high system load. We also describe a couple of other extensions to PrimCast that may provide benefits to practical applications. Finally, we evaluate a prototype implementation of PrimCast against two other state-of-the-art protocols and show that it achieves lower delivery latencies while still providing higher throughput.

Linearizable Atomic Multicast. State-machine replication (SMR) is a fundamental approach to rendering applications fault-tolerant while providing the strongest level of consistency, linearizability. In SMR, each command is executed by every replica, in the same order. This total ordering of requests is sufficient to ensure linearizability, and no other coordination is required of the replicas. In classic SMR, throughput is limited by the slowest replica, and state partitioning has been proposed as a solution to the scalability problem. Perhaps unintuitively, partially ordering requests with atomic multicast is not enough to ensure linearizability in partitioned systems. We thus propose a stronger version of atomic multicast that allows for partitioned applications to provide linearizability without additional coordination. We first show why the ordering guarantees of atomic multicast are not enough to ensure linearizable applications and then describe a stronger ordering property, atomic global order, that fixes the issue. Finally, we show how a classic atomic multicast protocol can be modified to provide the stronger property. This contribution has been published in the 11th Latin-American Symposium on Dependable Computing (LADC '22). [78]

1.4 Thesis outline

This thesis is organized as follows. Chapter 2 describes the system model and provides definitions used in the rest of the thesis. Chapter 3 presents the design, implementation and evaluation of GlobalFS. Chapter 4 presents PrimCast, a latency efficient genuine atomic multicast protocol. Chapter 5 shows how to strengthen atomic multicast to allow for linearizable applications without extra coordination. Chapter 6 concludes this thesis and discusses possible directions for future work.

Chapter 2

System Model and Definitions

2.1 System model

2.1.1 Processes and communication

We assume a distributed system composed of a finite set of interconnected processes. There is an unbounded set of *client processes* and a bounded set of *server processes* Π. Processes may fail by crashing, but do not experience arbitrary or malicious behavior (i.e., no Byzantine failures). A process that crashes is said to be *faulty*, otherwise it is *correct*. Processes communicate by message passing through pairwise communication channels. Communication channels are *quasi-reliable*. Channels do not create, corrupt or duplicate messages, and given two correct processes p and q, if p sends m to q, q eventually receives m.

We define $\Gamma = \{g_1, g_2, ..., g_m\}$ as the set of process groups in the system. Process groups are disjoint [46], and $\bigcup \Gamma = \Pi$. Associated with each group g, there is a set of quorums Q_g. Each quorum q in Q_g is a set of processes, such that $q \subset g$. The intersection between any two quorums in Q_g cannot be empty, and at least one of the quorums in Q_g must contain no faulty processes.

2.1.2 Synchrony assumptions

The FLP impossibility [39] says that a consensus protocol cannot ensure both liveness and safety in an asynchronous system. We thus consider a system that is *partially synchronous* [32]: it is initially asynchronous and eventually becomes synchronous. The time when the system becomes synchronous is called the *global stabilization time* (GST) and is unknown to the processes. Before the GST, there are no bounds on the time it takes for messages to be transmitted and actions to be executed. After the GST, such bounds exist but are unknown, and remain in effect forever. In practice, for the purposes of this work, "forever" means long enough for atomic multicast to make progress, that is, deliver messages.

We also assume that processes in group g have access to a *weak leader election oracle*, Ω_g. At each process in the group, Ω_g outputs a single process contained in g and has the following property: there is a time after which, at every correct $p_i \in g$, Ω_g outputs the same correct process $p_l \in g$.

2.2 Definitions

2.2.1 Multicast abstractions

A multicast abstraction allows a process to send a message m to multiple destinations. In this thesis, we only consider groups in Γ as destinations, and use $m.dest$ to refer to the set of destinations for m. When $|m.dest| = 1$ we say that m is a *local message*, otherwise, when m has multiple destinations, we say it's a *global message*.

Relying on the properties of a fault-tolerant multicast abstraction [48] greatly simplifies reasoning about the correctness of algorithms. In the following, we describe the properties of the different types of multicast used in this work.

Non-uniform FIFO reliable multicast

FIFO non-uniform reliable multicast provides two primitives to processes in the system: R-MULTICAST(m) and R-DELIVER(m). The former is used by processes to send a message to $m.dest$, and the latter signals its delivery at destinations.

The properties of non-uniform FIFO reliable multicast are:

- *Validity:* If a correct process executes R-MULTICAST(m) then, eventually, all correct processes in $\bigcup m.dest$ execute R-DELIVER(m).
- *Integrity:* For any message m and process p, p may only do R-DELIVER(m) once, and only if R-MULTICAST(m) was previously issued by some process.
- *Non-uniform agreement:* If a correct process executes R-DELIVER(m) then, eventually, every correct process in $\bigcup m.dest$ executes R-DELIVER(m).
- *FIFO order:* If a process executes R-MULTICAST(m) before executing R-MULTICAST(m') then, every process that executes R-DELIVER(m') must first execute R-DELIVER(m).

FIFO non-uniform reliable multicast implementations allow for a message to be delivered in *one communication step*, from origin to destinations [48].

Consensus

Consensus allows for processes in a group to propose values and then agree on a single proposed value. It provides two primitives to processes in group g: $\text{PROPOSE}_g(i, v)$ is used by a process to propose a value v for instance i, and $\text{DECIDE}_g(i)$ returns the value decided for instance i. It ensures the following properties:

- *Uniform integrity:* If $\text{DECIDE}_g(i) = v$ at some process then $\text{PROPOSE}_g(i, v)$ was previously executed by some process.
- *Termination:* If a correct process executes $\text{PROPOSE}_g(i, v)$ then, eventually, $\text{DECIDE}_g(i) = v'$ at every correct process.
- *Uniform agreement:* If $\text{DECIDE}_g(i) = v$ at some process and $\text{DECIDE}_g(i) = v'$ at another process then $v = v'$.

Atomic multicast and broadcast

Atomic multicast allows processes in the system to send a message m to the set of destination groups $m.dest$. It provides two primitives to processes in the system: $\text{A-MULTICAST}(m)$ to send messages and $\text{A-DELIVER}(m)$ to signal deliveries. It ensures the following properties:

- *Validity:* If a correct process executes $\text{A-MULTICAST}(m)$ then, eventually, all correct processes in $\bigcup m.dest$ execute $\text{A-DELIVER}(m)$.
- *Integrity:* For any message m and process p, p may only execute $\text{A-DELIVER}(m)$ once, and only if $\text{A-MULTICAST}(m)$ was previously issued by some process.
- *Uniform agreement:* If any process executes $\text{A-DELIVER}(m)$ then, eventually, every correct process in $\bigcup m.dest$ executes $\text{A-DELIVER}(m)$.
- *Global total order:* Let $\prec$ be a relation on the set of messages that processes A-DELIVER, such that $m \prec m'$ iff some process executes $\text{A-DELIVER}(m)$ before it executes $\text{A-DELIVER}(m')$. The $\prec$ relation is acyclic.

Global total order avoids cycles in the delivery sequence of messages. For example, suppose there are three processes, p, q, and r, each one in a different group, and messages m_1, m_2 and m_3. Global total order prevents a situation where p a-delivers m_1 and then m_2 ($m_1 \prec m_2$), q a-delivers m_2 and then m_3 ($m_2 \prec m_3$), r a-delivers m_3 and then m_1 ($m_3 \prec m_1$).

But global total order by itself allows faulty processes to a-deliver undesired sequences of messages. Indeed, it allows “holes" to appear in the message delivery sequence of faulty processes. For example, consider an execution where messages m_1 and m_2 are a-multicast to group g. A process $p \in g$ a-delivers m_1 and then m_2, and a faulty process $q \in g$ a-delivers m_2, then fails, and never a-delivers m_1 (i.e., m_1 leaves a hole in the delivery sequence of q). This execution satisfies all atomic multicast properties above, but it is undesired because processes p and q may produce different results if those messages were application requests to be executed. To prevent such executions, we also require atomic multicast to satisfy uniform prefix order [89].

- *Uniform prefix order:* Let m and m' be messages and p and q processes such that $\{p, q\} \subseteq \bigcup(m.dest \cap m'.dest)$. If p executes A-DELIVER(m) and q executes A-DELIVER(m') then either p executes A-DELIVER(m') before A-DELIVER(m) or q executes A-DELIVER(m) before A-DELIVER(m').

Atomic broadcast is a special case of atomic multicast, where every message is addressed to a single group.

Genuineness

A multicast protocol is *genuine* [46] when only the sender and destinations of a message m need to take steps for m to be delivered. Intuitively, a genuine protocol scales with the number of groups in the system, as long as most messages are local or destined to only a subset of the groups. We formally define genuineness as follows:

- *Genuineness:* for any admissible run $\mathscr{R}$ of the algorithm and for any process p, if p sends or receives a message in $\mathscr{R}$ then m is a-multicast in $\mathscr{R}$ and either $p \in \bigcup m.dest$ or p does the a-multicast of m.

2.2.2 Consistency criteria

Servers in a distributed application typically coordinate to provide clients with an interface that behaves as similarly as possible to the same application executing in a single machine. An execution is a finite sequence describing an interleaving of requests and respective responses, from one or more clients accessing the application. Consistency criteria restrict the set of valid executions. Some applications can afford to relax its consistency guarantees to allow for lower latency or lower coordination overhead. In this work, we consider the following consistency criteria:

Linearizability

An application is *linearizable* [7, 49] if, for any execution σ, there is a total order π on application requests that:

(i) respects the semantics of the requests, as defined in their sequential specifications, and

(ii) respects the real-time precedence of requests, where a request precedes another request in real time if the first request finishes before the second request starts.

Linearizability is a composable property [49]: a system solely composed of linearizable applications is also linearizable.

Sequential consistency

An application is *sequentially consistent* [7] if, for any execution σ, there is a total order π on application requests that:

(i) respects the semantics of the requests, as defined in their sequential specification, and

(ii) respects the partial ordering of commands defined by each client.

Differently from linearizability, sequential consistency is not a composable property.

Causal consistency

Let e and f be requests in some execution of a given application. We define the *happens-before* relation [57], here denoted $\rightarrow$, as follows:

- $e \rightarrow f$ if both e and f are requests from the same client, and e ends before f begins.
- $e \rightarrow f$ if the execution of request f observes any state update caused by e.

- the → relation is transitive.

An application is *causally consistent* if, for any execution σ and application requests e and f, if $e \rightarrow f$ then the execution of f observes all state updates caused by e. Under causal consistency, two clients may see unrelated updates in different orders [66].

2.2.3 State-machine replication

State-machine replication [22, 93], or active replication, is an approach to rendering applications fault-tolerant that provides linearizability. The application state is fully replicated by every server, and every server executes every application request, in the same order. Since each server independently executes each request, request execution must be deterministic, only depending on application state. Thus, replicas go through the same sequence of state changes, and produce the same output for each request executed.

2.2.4 Primary-based replication

Primary-based replication [17, 52, 64, 108], also known as passive or primary-backup replication, is an approach to fault-tolerance in which one server, the primary, executes every application request. The resulting state changes are then propagated from the primary to the other replicas, commonly referred to as followers or backups. Hence, in primary-based replication, request execution does not need to be deterministic.

Chapter 3

A Strongly Consistent Multi-Site File System

This chapter presents GlobalFS, a strongly consistent, geographically distributed file system providing a POSIX-like interface. GlobalFS handles file data and metadata separately and builds on two fundamental building blocks: group communication in the form of atomic multicast and multiple instances of a single-site data store. Instead of executing every file system operation the same way, GlobalFS takes into account the individual requirements of each operation and places it into one of four execution modes. We define each execution mode and show how all file system operations can be implemented with these modes, while ensuring strong consistency and tolerating failures. We describe our prototype implementation of GlobalFS in detail and provide an in-depth study of its performance. In particular, GlobalFS has been deployed on Amazon's EC2 platform, across all nine available regions at the time, and we show that it scales geographically, with performance comparable to other distributed file systems for commands local to a region, while providing strongly consistent operations over the whole system.

3.1 Motivation

Cloud infrastructures, composed of multiple interconnected datacenters, have become an essential part of modern computing systems. They provide an efficient and cost-effective solution to hosting web-accessible services, storing and processing data, or performing compute-intensive tasks. Large companies like Amazon or Google do not only use such architectures for their own needs, but they also rent them to external clients in a variety of options, e.g., infrastructure (IaaS), platform (PaaS), software (SaaS), or data (DaaS) as a service. Such global infrastructures rely on geographically distributed datacenters for fault-tolerance, scalability, and performance reasons.

Almost every distributed application has some need for data storage. Given the diverse requirements of these applications, distributed data storage comes in many flavors: file systems, SQL databases, key-value stores, persistent logs and many others. Out of these, distributed file systems provide a simple and familiar interface for applications to take advantage of data replication and availability. Providing access through a standard API such as POSIX may allow for existing applications to be migrated with little to no modifications.

In this chapter, we focus on the design of a geographically distributed file system, accessible via a POSIX-like interface. Most previous designs for geographically distributed file systems [54, 70] have provided weak consistency guarantees (e.g., eventual consistency [29]) to work around the limitations formalized by the CAP theorem [42], which states that distributed applications can fully support at most two of the following three properties simultaneously: consistency, availability, and tolerance to partitions. Our goal is to ensure strongly consistent file system operations despite node failures, at the price of possibly reduced availability in the event of a network partition. Weak consistency is suitable for domain-specific applications where programmers can anticipate and provide resolution methods for conflicts, or work with last-writer-wins resolution methods. Our rationale is that for general-purpose services such as a file system, strong consistency is more appropriate as it is both more intuitive for the users and does not require human intervention in case of conflicts.

Strong consistency requires ordering commands across replicas, which needs coordination among nodes at geographically distributed sites (i.e., regions). Designing strongly consistent distributed systems that provide good performance requires careful tradeoffs. One such tradeoff happens between the latency of operations touching data in a single region and operations touching data across regions. We capture this compromise with the notion of *geographical scalability*: having some region be part of a given deployment should not negatively impact the performance of operations not touching that region.

Geographical scalability is motivated by geo-distributed applications that wish to exploit the locality of data access without compromising consistency or reducing the scope of operations to a single region. This trend is becoming increasingly more important with the wide range of applications that are deployed over multiple datacenters spanning several regions on cloud platforms such as Amazon's EC2. [1] Yet, achieving geographical scalability is notoriously difficult. As of the

[1] https://aws.amazon.com/ec2/

writing of this thesis, Amazon's own solution for distributed file systems, Elastic File System (EFS), does not support cross-region deployments. As another example, among the few existing file systems with explicit support for geographical distribution, CalvinFS [105] totally orders requests. As a consequence, end users with applications spanning multiple geographic regions are left with one of two options: (1) ad hoc solutions for synchronizing data across regions or (2) increased latency for cross-region deployments, even for operations that access objects in a single region.

The rest of this chapter is organized as follows. Section 3.3 introduces GlobalFS's architecture. Section 3.4 presents the protocol design. Section 3.5 describes the implementation of our prototype. Section 3.6 discusses the results of our experimental evaluation. Section 3.7 reviews related work. Section 3.8 concludes the chapter with some discussion on the limitations of and possible improvements to GlobalFS.

3.2 General idea

We consider a global deployment environment, with server processes placed in *datacenters* that are geographically distributed around the world. Datacenters are grouped into *regions* by their proximity; datacenters in the same region can typically communicate with low latency, while still providing independent failure scenarios. Thus, data can be replicated to multiple datacenters in a single region to ensure some level of disaster tolerance. Still, some applications have clients distributed around the globe and need to be deployed to multiple regions, so that their clients can access them with lower latency. Since the latency of communication between regions is in general much higher than communication within a region, such applications need to use protocols that are geographically scalable.

GlobalFS is a distributed file system that achieves geographical scalability by exploiting two abstractions. First, it relies on *data stores* located in geographically distributed datacenters. File data is stored and replicated as immutable blocks in these data stores, which are organized as distributed hash tables (DHTs). Second, GlobalFS uses an *atomic multicast* abstraction to maintain mutable file metadata and orchestrate multi-site operations. Atomic multicast provides strong order guarantees by partially ordering operations (Section 2.2.1).

GlobalFS notably differs from other distributed file systems by defining a partition model in which files and folders can be placed according to access patterns (e.g., in the same region as their most frequent users), as well as four execution modes corresponding to the operations that can be performed in the file

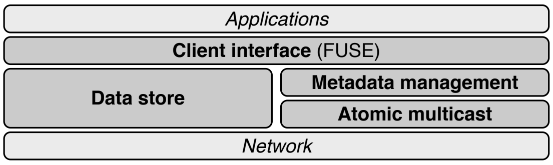


Figure 3.1. Overall architecture of GlobalFS.

system: (1) single-partition operations, (2) multi-partition uncoordinated operations, (3) multi-partition coordinated operations, and (4) read-only operations. While single-partition and read-only operations can be implemented efficiently by accessing each region independently, the other two operations require synchronization between multiple regions. By leveraging atomic multicast and distinguishing between these four modes of execution, GlobalFS can exploit geographical locality, providing low latency for single-region commands while allowing for consistent operations across the whole file system. GlobalFS ensures sequential consistency for update operations and causal consistency for reads.

3.3 System architecture

This section presents in detail the architecture and design of GlobalFS and how the file system can be partitioned and replicated.

3.3.1 Components

The architecture of GlobalFS consists of four components: the client interface, the data store, metadata management, and atomic multicast (see Figure 3.1).

The *client interface* provides a file system API supporting a subset of POSIX 1-2001 [1]. GlobalFS implements file system operations sufficient to manipulate files and directories. Some file system calls change the structure of the file system tree (i.e., the files and directories within each directory). Each file descriptor seen by a client when opening a file is mapped to a local file descriptor at each GlobalFS server. We support file-specific operations: `mknod`, `unlink`, `open`, `read`, `write`, `truncate`, `symlink`, `readlink`; directory-specific operations: `mkdir`, `rmdir`, `opendir`, `readdir`; and general-purpose operations: `stat`, `chmod`, `chown`, `rename`, and `utime`. We support symbolic links, but not hard links.

Like most contemporary distributed file systems (e.g., [19, 41, 94, 98]), GlobalFS decouples metadata from data storage. Metadata in GlobalFS is handled by the *metadata management* layer. Each file has an associated *inode* block (`iblock`) containing the metadata information about the file (e.g., its size, owner, and access rights) and pointers for its data blocks. The actual content of a file is stored in data blocks (`dblocks`). The two types of blocks are handled differently and stored separately: `dblocks` are immutable and stored directly by the clients in the storage servers; `iblocks` are mutable and maintained by the metadata servers.

GlobalFS distinguishes updates (i.e., operations that modify the state of a file or directory) from read-only operations. Updates are sequentially consistent while reads are causally consistent (Section 2.2.2). Every update operation is ordered by atomic multicast. Partially ordering messages, as defined by atomic multicast, is a fundamental requirement for achieving scalable distributed systems.

The *data store* provides a key-value store with primitives to read (`get`) and create (`put`) data items. It is implemented as a collection of distributed hash tables (DHTs), with one instance of the data store per datacenter. The `get` and `put` operations on each instance of the data store are linearizable (see Section 2.2.2). Maintenance of the data in the DHT is simple and efficient given that data blocks are immutable. DHT-based data stores scale remarkably well horizontally [27, 29, 56, 85]. The design of the data store itself is orthogonal to this work.

3.3.2 Partitioning and replication

Data partitioning and replication have an important impact on the performance and reliability of a data management system. Horizontal partitioning (sharding) is commonly used to scale distributed file systems. For example, hashing the pathname of each file is a straightforward way to distribute files across the system [105]. Hashing provides good load distribution of files but its lack of support for locality might place files far away from their most frequent or likely users. Even though the design of GlobalFS could support any partitioning scheme, including hashing, we explore a different approach to partitioning and replication, which takes locality into consideration, as we now explain.

Clients and metadata replicas have access to the *partitioning function* mapping a given path to the partition replicating it. The file system is partitioned and replicated according to the expected client access patterns and the degree of fault tolerance desired. Files that are mostly read and rarely modified (e.g., system and application programs) are placed in a single "global" partition, replicated across regions; files that experience locality of access (e.g., temporary files related to a client) are placed in "local" partitions, replicated in datacenters inside a single region, close to the clients most likely to access them. In this setup,

Partition	Replication	Performance	Fault tolerance
Global	across regions	best for reads	disaster
Local	within region	best for reads & writes	datacenter crash

Table 3.1. Partitions in GlobalFS.

a file in the global partition can be read locally from any region, resulting in high throughput and low latency for read operations. Updating a file in the global partition, however, is an expensive operation involving all regions. Local partitions, on the other hand, can provide high throughput and low latency for both reads and updates, as long as the client is close to the file's location. Both local and global partitions can tolerate the failure of an entire datacenter. Moreover, the global partition can tolerate the failure of all datacenters in a region (i.e., a disaster). Table 3.1 summarizes the two partition types in GlobalFS.

To allow for a flexible system deployment, GlobalFS decouples data from metadata. Although data and metadata for a given file are likely to be stored in the same region, the system can cope with the case in which the metadata of a file is stored in a region and the file data is stored in a different region.

3.3.3 Use of atomic multicast

In order to allow operations to be consistently propagated to the replicas, one multicast group is associated with each partition. Servers subscribe to two multicast groups: one, g_{all}, associated with all the servers in the system, and another associated with servers in the datacenters in the same region. This particular characteristic of having a global g_{all} group comes from our choice of atomic multicast protocol, Multi-Ring Paxos [68]. Multi-Ring Paxos provides a slightly different interface than the classic atomic multicast described in Section 2.2.1. In Multi-Ring Paxos, messages can only be sent to a single group, but each server can subscribe to multiple groups. Commands that update files in the global partition or update files in multiple local partitions are multicast to g_{all}; commands that update files in a local partition are multicast to the group associated with the partition. When a server receives a command it is not interested in, it simply ignores it. The use of atomic multicast allows for independent local partitions while still providing consistent operations across them. GlobalFS exploits this flexibility through its different execution modes, detailed in Section 3.4.1.

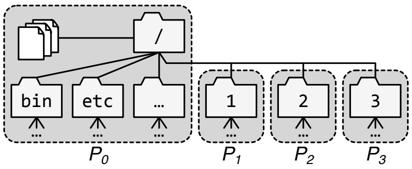


Figure 3.2. Illustrative deployment of GlobalFS with 4 partitions. Partition P_0 is replicated in all regions and each other partition is replicated in one different region.

3.3.4 Example deployment

Consider a deployment involving three regions, R_1, R_2, and R_3, each with three datacenters. The file system is partitioned in four partitions, $P_0, ..., P_3$ (see Figure 3.2), such that P_0 is replicated in datacenters in all regions and partition P_i, $1 \leq i \leq 3$, is replicated in datacenters in region R_i. In this scenario, we have clients and servers (metadata and data store) distributed across the regions, that is, in addition to the metadata associated with the region's partition, each datacenter also hosts an instance of the data store. More precisely, the metadata and data for the directory `/1` and all its contents (recursively) are stored in servers in P_1. In the same manner, `/2` and `/3` are respectively mapped to P_2 and P_3. Files not contained in any of these directories (e.g., `/`, `/bin`, `/etc`) are in partition P_0.

3.4 Protocol design

GlobalFS differentiates between four classes of operations and defines for each one a different execution mode. GlobalFS's execution modes provide the basis for the implementation of each file system operation. We start by going through the details of each execution mode. We then describe the execution of `open`, `read` and `write` operations, from start to finish. Finally, we discuss how failures are handled in GlobalFS.

3.4.1 Execution modes

Each operation in GlobalFS follows one of the following execution modes. Except for `read` and `write` operations, all file system operations access only the metadata servers.

Single-partition operations. A single-partition operation modifies metadata stored in a single partition. As a consequence, operations in this class are multicast to the group associated with the concerned partition and, when delivered, executed locally by the replicas. The execution of a single-partition operation follows state-machine replication 2.2.3: each replica delivers a command and executes it deterministically. One of the replicas replies to the client.

The following operations are single-partition in GlobalFS, where the terms *child* and *parent* are used to refer to an object and the directory that contains it, respectively.

- `chmod`, `chown`, `truncate`, `open`, and `write`;
- `mknod`, `unlink`, `symlink`, and `mkdir` when the parent and child are in the same partition; and
- `rename`, when the origin, origin's parent, destination, and destination's parent are in the same partition.

Note that while a single-partition operation in a local partition involves only servers in one region, a single-partition operation in the global partition (multicast to group g_{all}) involves servers in all regions of the system.

Uncoordinated multi-partition operations. An uncoordinated multi-partition operation accesses metadata in more than one partition, but the operation's execution at each partition can complete without any input from the other partitions involved. This is similar to the notions of *independent transactions* in Granola [26] or *one-shot transactions* in H-Store [53]. In GlobalFS, these commands are those involving only a parent directory and child, where each is mapped to a different partition. Since the parent's partition knows whether the child already exists or not, the partial ordering of atomic multicast is sufficient to guarantee consistency: both partitions will independently reach the same decision in regards to success or failure.

To execute an operation that concerns multiple partitions $P_1, P_2, ..., P_n$, the operation is atomically multicast to all replicas of all involved partitions. Upon delivery, each replica P_i executes the operation and one of the replicas replies to the client. To reach replicas in multiple partitions, the operation is multicast to group g_{all}; if a replica delivers an operation it is not concerned about, the replica just ignores the operation.

The following file system commands are implemented as uncoordinated multi-partition operations:

- mknod, unlink, symlink, mkdir, rmdir when the parent and child are in different partitions.

Coordinated multi-partition operations. The execution of some operations requires the involved partitions to exchange information. In GlobalFS, this may happen in the case of a rename (i.e., moving the location of a file or directory). In this case, file metadata has to be moved from the origin's partition to the destination's partition. As a result, a rename may involve up to four partitions, given by the placement of the origin, origin's parent, destination, and destination's parent. Consequently, a rename operation might fail in one of the partitions (e.g., origin does not exist) but not in another.

To execute a coordinated multi-partition operation, the client multicasts the operation to all concerned partitions (i.e., multicast group g_{all}). Upon delivery of the operation, the involved partitions exchange information about the command and whether it can or cannot be locally executed. In the case of a rename, the file's attributes and list of block identifiers need to be sent to the destination partition. Similarly to a two-phase commit protocol, the command only executes if all involved partitions agree that it can be executed successfully.

Read-only operations. Read-only operations are executed by a single metadata replica and data store server.[2] For read-only operations, GlobalFS provides causal consistency. This is not obvious to ensure since a client may submit a write operation against a server and later issue a read operation against a different server or even read from two separate servers. When the second server is contacted, it may not have applied the required updates yet. GlobalFS provides causal consistency for read operations by carefully synchronizing clients and replicas, as we explain in the following.

We use an approach inspired by vector clocks [37, 83] where clients and replicas keep a vector of counters, with one counter per system partition. In the example described in Section 3.3.4, clients and replicas keep a vector with four entries, associated with partitions $P_0, ..., P_4$. Every request sent by a client contains v_c, the client's current vector, and each reply from a replica includes the replica's vector, v_r. A read is executed by a replica only when $v[i]_r \geq v[i]_c$, i being the object's partition. The idea is that the replica knows whether it is running late, in which case it must wait to catch up before executing the request.

[2] GlobalFS does not implement *atime* (i.e., time of last access), as recording the time of the last access would essentially turn every read into a write operation to update the file's access time.

Operation	Partitions	Multicast	Performance
Read-only	one	not multicast	1^{st} (best)
Single-partition	one	g_{all} or g_i	2^{nd}
Uncoord. multi-partition	two or more	g_{all}	3^{rd}
Coord. multi-partition	two or more	g_{all}	4^{th} (worst)

Table 3.2. Operations in GlobalFS.

When a replica receives an update operation from a client, the client's vector v_c is atomically multicast together with the operation. Upon delivery of the command by a replica of P_i, entry $v[i]_r$ is incremented. Every other entry j in the replica's vector is updated according to the delivered v_c, whenever $v[j]_c > v[j]_r$. Clients update their vector on every reply, updating $v[i]_c$ if $v[i]_r > v[i]_c$, for each entry i.

The following file system commands are implemented as read-only operations: `read`, `getdir`, `readlink`, `open` (read-only), and `stat`.

Table 3.2 summarizes GlobalFS operations. Single-partition and read-only operations access a single partition. While a single-partition operation is multicast to the group associated with the partition, a read-only operation is not multicast but is executed by a single metadata replica (and a data store server). For example, according to the illustrative deployment described in Section 3.3.4, a write operation for partition P_0 is multicast to g_{all} and a write operation for any of the other partitions P_i is multicast to g_i. Uncoordinated multi-partition and coordinated multi-partition operations access multiple partitions. Such operations are multicast to group g_{all}. Since read-only operations only involve a single metadata server and are not multicast, we expect such operations to outperform any other operations in GlobalFS. Single-partition operations involve all replicas within a single partition, and therefore should perform better than the multi-partition operations. Finally, because uncoordinated multi-partition operations do not require servers in different partitions to interact during the execution of a command, they are expected to perform better than coordinated multi-partition operations.

3.4.2 The life of some file system operations

To `open` a file, the client uses the partitioning function to discover the set of partitions replicating the provided path. The client then issues an `open` RPC to the closest replica from any of the involved partitions. The response for this RPC is a file handle that the client uses to issue subsequent `read` and `write` operations.

Upon receiving an open RPC from the client, a replica checks whether the file is being opened for reading or writing. If the file is open for reading, the replica creates a local file handle, valid only at this replica, and returns it to the client. If the file is open for writing, the file handle needs to be opened in all replicas as writes are replicated. The open command is multicast to the associated group (given by the partitioning function) and executed by all responsible replicas. Once a replica has finally delivered and executed the command, it directly replies to the client.

For a read operation, the client needs to execute two steps. First, it issues a read RPC to the replica holding the file handle. The replica, upon receiving the read, finds the requested file's metadata and looks for the blocks that match the offset and number of bytes requested. The reply from the RPC is a list of block identifiers and pointers. With the block identifiers, the client contacts the closest data store replicating the file to get the actual data for the blocks. Multiple blocks can be requested in parallel from different data store nodes. After that, the client can build the sequence of bytes that need to be returned by the read operation.

For a write operation, the client first creates one or more data blocks from the bytes that need to be written to the file. The client then contacts all the data stores that need to replicate the file (given the partitioning function), and inserts the blocks there, with unique identifiers generated at random. Insertion of multiple blocks can be done in parallel. If all inserts are successful, the client uses the partitioning function to get the partition replicating the file, chooses the closest replica and issues a write RPC with the block identifiers as parameters. The replica, upon receiving the write RPC, multicasts the command to the responsible group. Upon delivery of the command, a replica finds the metadata for this file and inserts the new blocks. The replica that received the initial RPC from the client replies. On success, the write returns the number of bytes written.

3.4.3 Failure handling

Metadata replicas use state-machine replication to handle metadata within partitions. A replica only executes a command that has been successfully delivered by multicast. Thus, if a replica executes a command, other correct replicas in the same partition will eventually deliver and also execute the command. GlobalFS uses Multi-Ring Paxos as its atomic multicast implementation (see Section 3.5.2). As long as one metadata replica in each partition and a quorum of acceptors in each communication group are available, metadata in the whole file system is available for writing and reading.

The recovery of a metadata replica is handled by installing a replica check-point and replaying missing commands [11]. Coordinated multi-partition commands require one extra step. For coordinated multi-partition commands, replicas in the involved partitions need to exchange information before deciding whether the command can execute or not. A recovering replica, upon replaying a coordinated multi-partition command, requests this information from replicas in the other partitions. To allow for recovery, whenever a replica sends information out regarding a coordinated command, it also stores this information locally.

Each key-value pair in the data store of a given partition is replicated in $f + 1$ storage nodes. Hence, up to f storage nodes can fail concurrently without affecting data block availability.

To account for datacenter failures and disasters, metadata replicas and storage nodes can be placed in different datacenters inside a region.

Client failures during a `write` or a file delete operation can leave "dangling" `dblocks` inside the data store. `dblocks` without pointers in any `iblock` are unreachable and can be removed from the data store (the implementation of a garbage collector is left as future work).

3.5 Implementation

In this section, we discuss the implementation of GlobalFS main components, as depicted in Figure 3.3.

3.5.1 Clients

Files are accessed through a *file system in user space* (FUSE) implementation [38]. FUSE is a loadable kernel module that provides a file system API to user space programs, letting non-privileged users create and mount a file system without writing kernel code. According to [104], FUSE is a viable option in terms of performance for implementing distributed file systems. Clients know the partitioning function used by the system (currently hard-coded in the client) and use Zookeeper [50] to find the set of available replicas. When using FUSE, every system call directed at the file system is translated to one or more callbacks to the client implementation. In GlobalFS, most FUSE callbacks have an equivalent RPC (remote procedure call) available in the metadata servers. By using the partitioning function, a client can discover to which metadata replica or data store it needs to direct a given operation. Whenever a client has the option of directing a command to more than one destination, it chooses the closest one (the one with the lowest latency).

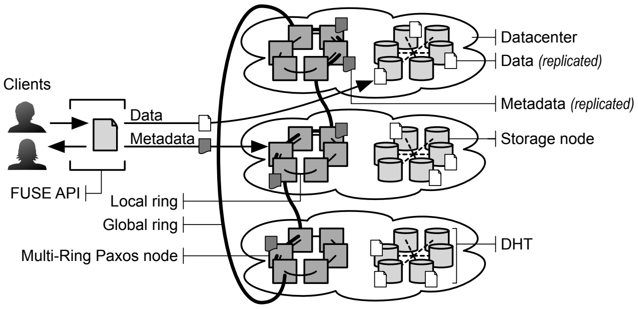


Figure 3.3. Components and interactions in GlobalFS.

3.5.2 Atomic multicast

We use URingPaxos,[3] a unicast implementation of Multi-Ring Paxos [68], which implements atomic multicast by composing multiple instances of Paxos to provide scalable performance. Each multicast group is mapped to one instance of Paxos. A message is multicast to one group only. Processes that subscribe to multiple groups use a deterministic merge procedure to define the delivery order of the messages such that processes deliver common messages in the same relative order.

For each Paxos instance, Multi-Ring Paxos disposes proposers, learners, and a majority-quorum of acceptors in a logical directed ring in order to achieve high throughput. Processes in the ring can assume multiple roles and there is no restriction on the relative position of these processes in the ring, regardless of their roles. Each ring has a Paxos coordinator, typically the first acceptor in the ring.

In our setup we keep a global ring that includes all metadata replicas in the system, as illustrated in Figure 3.3. This ring implements the g_{all} group discussed in Section 3.3.3. Each other group is implemented by a ring that includes replicas in the same region.

3.5.3 Metadata replicas

Metadata in GlobalFS is kept by replicated servers, using state machine replication [93]. Replicas can be part of multiple multicast groups. In our prototype, each replica is a Multi-Ring Paxos learner. When a replica delivers a command, the replica checks whether it should execute the command by using the partitioning function. The file system metadata is kept in memory by the replica and

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

the sequence of commands is stored by Multi-Ring Paxos acceptors. Replicas can be configured to keep their state in memory or on disk, with asynchronous or synchronous disk writes.

The file system is represented as a tree of nodes. There are three node types: directory, file, and symbolic link. A directory node stores the directory properties (e.g., owner, permissions, times) and a hash table of its children nodes, stored by name. A file node keeps the file properties and a list of blocks representing its contents. Symbolic link nodes only need to store the node properties and the target path of the link.

The metadata replicas are implemented in Java and expose a remote interface to the clients via Thrift [6].

3.5.4 Data store

GlobalFS is designed to support any back-end data store that exposes a typical key-value store API and provides linearizability. Our data store is implemented in Go and uses LevelDB [61] as its storage backend. Depending on the application requirements and fault model, data may be stored persistently on disk or maintained in memory.

The data store is organized as a ring-based DHT and uses consistent hashing for data placement. Each server maintains a full membership of other servers on the ring, allowing one-hop lookups. This design, similar to Cassandra [56] or Dynamo [29], provides good horizontal scalability and stable performance.

Each block is assigned to the first server whose logical identifier follows the block identifier on the ring. A block is replicated as r copies, by copying it onto the $r-1$ successors (i.e., servers that immediately follow this first server on the ring). This ensures data availability with up to $r-1$ simultaneous failures. Servers periodically check for the availability of copies of their blocks onto their successors and create additional copies when necessary. Similarly, servers periodically check for their predecessor availability and take over the responsibility for their ranges upon failure, also creating additional copies. Since blocks are written only once to the DHT, there is no need for coordination to ensure consistency.

Clients contact the DHT via any of its proxy servers. The client picks one of the servers and sends the block only once: the server will then create the r copies of the block.

3.6 Evaluation

We evaluate GlobalFS using Amazon's EC2 platform. We deploy VMs in all nine EC2 regions available at the time of our experiments. For each region, we distribute servers and clients in three separate availability zones to tolerate datacenter failures. More specifically, inside a single region, we place one server (metadata colocated with storage) and one client machine in each availability zone (six VMs per region). In regions where only two availability zones are present (e.g., `eu-central-1`) we compromise by placing two servers and clients in the same zone. We used `r3.large` (memory optimized) and `c3.large` (compute optimized) instance types, with 2 virtual CPUs, 32 GB SSD storage, and respectively 15.25 and 3.75 GiB memory [5]. We use `r3.large` instances for servers and `c3.large` instances for clients.

We configure the atomic multicast layer based on Multi-Ring Paxos to use in-memory storage. The data store nodes use LevelDB with asynchronous writes to persistent storage.

Our evaluation starts by assessing that the data store implementation in Go using LevelDB [61] can sustain enough throughput not to constitute a bottleneck in our GlobalFS microbenchmarks. We deploy five storage nodes inside a single region with a replication factor of 2 (i.e., each block has 2 copies). For block sizes of 1 KB, the data store achieves more than 8,000 `put` operations per second, i.e., around 0.06 Gb/s of aggregate traffic. With larger block sizes (32 KB), the same set-up could sustain around 6,500 `get` operations per second, or around 1.58 Gb/s. For the rest of our experiments, we use blocks of 1 KB.

3.6.1 Microbenchmarks

We use a custom microbenchmark to evaluate the performance and scalability of GlobalFS for the following types of operations:

▹ **read 1 KB:** each client reads sequentially from a small file (10 KB), in 1 KB chunks. Upon reaching the end of the file, a client wraps and continues reading from the beginning. We disable caching on the client side so that all reads go through the complete protocol.

▹ **write 1 KB:** each client writes sequentially to a file in 1 KB chunks.

▹ **create:** each client repeatedly creates empty files. This operation accesses only the metadata servers.

▹ **create 1 KB:** each client repeatedly creates a file and writes 1 KB to it. Each operation requires 3 sequential metadata operations: `mknod`, `open`, and `write`.

Each operation type is further divided into two categories: *local* operations target files located in the client's local partition and *global* operations target files located in the global partition.

Performance with 3 regions

For these experiments, we use 3 different geographically distributed regions: `us-west-2`, `us-east-1`, and `eu-west-1`. We deploy 1 local partition in each region. Each partition features 3 servers, each in a different datacenter (availability zone). Metadata and storage are co-located: each server holds a metadata replica and a storage node. Each datacenter also holds one client machine, thus there are 3 clients per region. Each client machine has one GlobalFS FUSE mount point. We then run multiple instances of our benchmark application on top of each client machine.

For comparison, we also show values reported by HDFS in [98] and CalvinFS in [105]. HDFS uses a centralized non-replicated metadata server. The values reported for HDFS consider only metadata performance, and thus represent an upper bound for the actual performance of HDFS. For CalvinFS, we report the approximate values with 9 servers. As the exact values for CalvinFS with 9 servers are not provided in [105], we approximate them by interpolating the values for 6 and 18 servers (we contacted the authors but could not obtain the source code). Due to the linear behavior exhibited by CalvinFS, our approximation should be fairly accurate.

Throughput. Figure 3.4 shows the maximum throughput achieved for each operation. For read operations, GlobalFS achieved around 60% higher throughput than CalvinFS, for both local and global operations. HDFS achieves higher performance for reads, but it takes only metadata performance into account. Reads in GlobalFS scale linearly with the number of replicas (a single replica needs to be contacted).

For writes, GlobalFS was able to surpass the throughput of HDFS for local operations, even though HDFS considers only metadata. GlobalFS was able to achieve 6 times the throughput of CalvinFS for local writes. For global writes, CalvinFS's throughput was 1.7 times higher. In our setup for GlobalFS, the global partition is replicated by *all* servers in the system (thus it cannot scale).

For creating a file with content, by not complying to the POSIX interface, CalvinFS is able to execute the operation using a single metadata access (by means of a custom transaction). Adhering to POSIX requires a sequence of *three* metadata operations: `create` the file, `open`, and `write`. The `close` is omitted as the write is

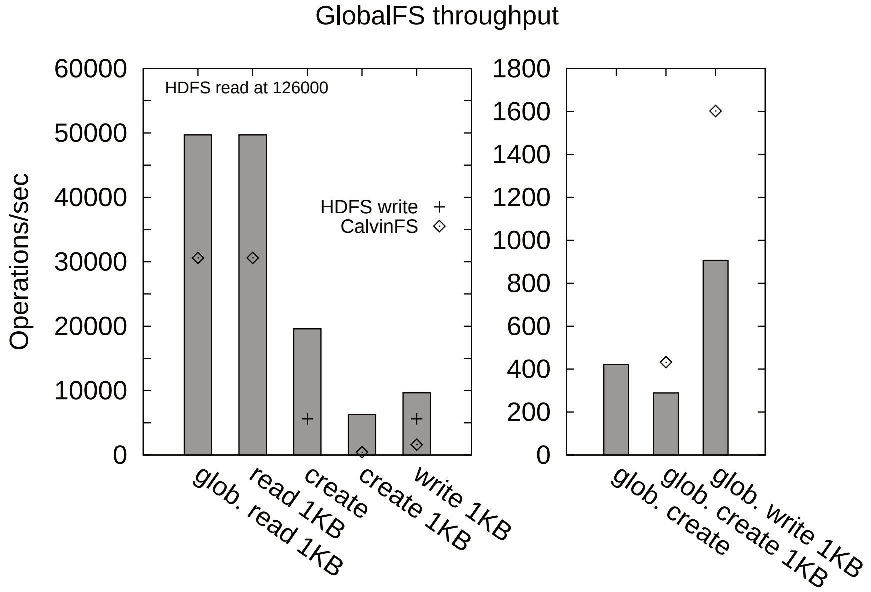


Figure 3.4. Maximum throughput for different GlobalFS operations with the baseline deployment of 3 partitions.

synchronous. Even though GlobalFS needs the three operations in the same scenario, it can achieve throughput 14.5 times higher than CalvinFS using the faster local partitions. Considering global creates, CalvinFS achieves 1.5 times higher throughput. On the other hand, creating an empty file requires a single metadata operation. In this case, GlobalFS was able to surpass even the performance of HDFS when using the local partitions (3.5 times the throughput). Values for this operation are not reported in the paper that presents CalvinFS [105].

These results show the benefit of exploiting data locality. CalvinFS, while scaling throughput with the number of replicas within a datacenter, does not benefit from local, fast operations. In CalvinFS, all write operations need to go through the global log, thus introducing an overhead on latency. This problem is exacerbated in WAN deployments: either the log is disaster tolerant and all operations pay the cost, or the log is local to a region and clients in other regions need to pay the round-trip latency. GlobalFS, on the other hand, allows for files to be either locally or globally replicated, thus providing the option for users to choose between availability (disaster tolerance) and performance (throughput and latency). Note that operations across the whole system are still strongly consistent in GlobalFS. The results also show that GlobalFS can deliver good performance while still providing a POSIX interface, thus allowing for existing applications to be used without modification.

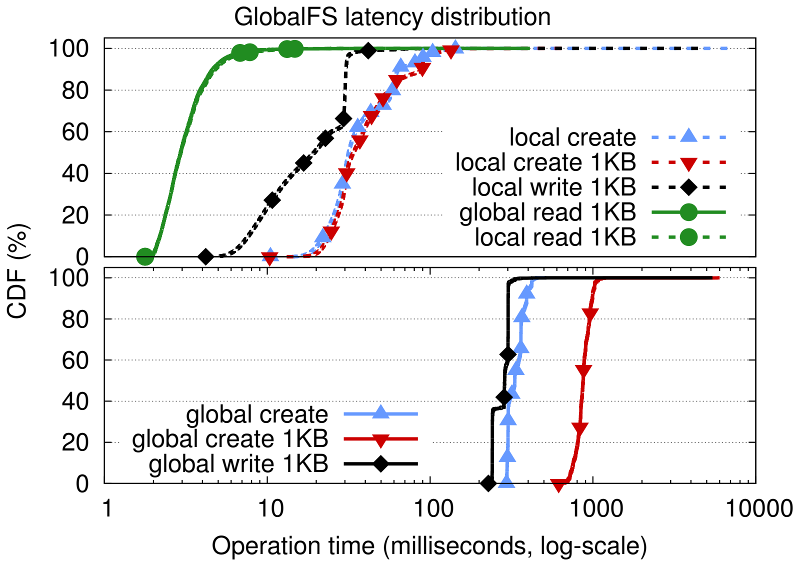


Figure 3.5. Latency distribution for different GlobalFS operations with the baseline deployment of 3 partitions. Latencies measured at 50% of maximum throughput.

Latency. Figure 3.5 shows the latency distribution for the different types of operations. We measure latency with the system supporting around 50% of its maximum throughput. The results show that operations can be divided roughly in 3 groups in regards to latency: reads, local writes, and global writes (we group creates with writes). Read operations, global and local, observe the lowest latency values, an average of 3.5 ms. This is due to reads being executed by a single metadata replica and not having to go through atomic multicast. Clients can also obtain `dblocks` from the local data store. Local writes, which need to be multicast to servers in a single region, can achieve the second lowest latency, with averages around 20–40 ms. Finally, global writes observe the highest latency values. In our setup, global writes need to be multicast to all servers in the system, across all regions. Clients also need to insert `dblocks` in all data stores. Even so, latency values for writes and creating empty files on the global partition had an average of around 300 ms.

Geographical scalability

We use the notion of *geographical scalability* to assess the impact of geographical deployments on performance. As a metric for geographical scalability, we use the ratio between the maximum throughput of local commands in a region in a system that spans multiple regions and the throughput of the region when deployed alone. A geographical scalability of 1 is ideal. Intuitively, it means that the throughput achieved in a single region is not affected by the system being deployed to other regions.

We compute geographical scalability as follows. We first measure the throughput achieved with GlobalFS in a single EC2 region, `eu-west-2`. Then, we consider multi-region deployments with 3, 6, and 9 regions:

▷ **3 regions:** `us-west-2`, `us-east-1`, `eu-west-1`.

▷ **6 regions:** + `us-west-1`, `eu-central-1`, `ap-northeast-1`.

▷ **9 regions:** + `ap-southeast-1`, `ap-southeast-2`, `sa-east-1`.

The reported value is the ratio between the multi-region and the single-region configurations.

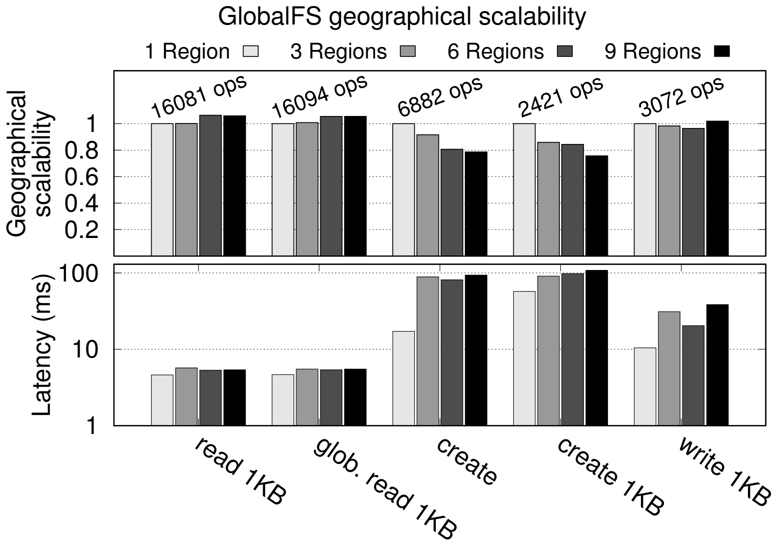


Figure 3.6. Geographical scalability and 95th percentile latencies for different GlobalFS operations, with increasing system size. Latencies measured at around 50% of maximum throughput.

Figure 3.6 (top) shows that GlobalFS scales almost perfectly for all local operations. For create operations, we see a drop in performance as regions are added, down to around 0.8 when all available regions are used. Maximum absolute throughput is shown above the single-region configuration. Figure 3.6 (bottom) shows the 95th-percentile of latency in each deployment, measured at around 50% of maximum load. Read commands suffer no impact in latency as they can be executed by a single replica (note that both lines are superimposed).

For local writes and creates, the largest increase in latency happens when the system grows from 1 to 3 regions. While commands are executed by replicas inside a single region, Multi-Ring Paxos still needs to synchronize groups. Therefore, latency variations in the global ring can affect the performance of local commands [68].

3.6.2 Compilation benchmarks

We now present results of an experimental evaluation conducted with a more practical workload. In particular, we chose as workload the compilation process of two well-known open-source projects: the `bc` numeric processing language (v1.06), and the Apache `httpd` web-server (v2.4.12). These two projects differ in size of the compressed archives (278 kB and 6 MB), number of shipped files (94 and 2,452) and lines of ansi-C code to compile (8,510 and 157,575). They expose different workloads to the underlying file system and are often used as benchmarks [103]. We evaluate the performance of these workloads when executed on global and local partitions of GlobalFS. We compare the results against three widely used distributed file systems: NFS (v4.1) [102], GlusterFS (v3.7) [28] and CephFS (v0.94) [111]. Our objective is to assess that, while providing stronger guarantees, GlobalFS compares favorably to *de-facto* industry implementations.

We configure NFS with one single shared directory mounted remotely by the same clients. The NFS server runs in the `us-west-2` region. We disable all caching features on GlobalFS, and the NFS clients mount the remote directory with `lookupcache=none,noac,sync` options. Note that NFS lacks native support for replication,[4] while GlobalFS is configured to always guarantee two copies per `dblock`. We use FUSE-based bindings for GlobalFS, GlusterFS, and CephFS.

Table 3.3 embeds the operations breakdown of the system calls issued by the different commands (decompress, configure, and compile) used for these experiments. We evaluate GlobalFS either within a global or a local partition, and compute the average over 3 distinct executions. All file systems are mounted by 9 clients spread equally across 3 regions, but the workload is executed on a single client. We use equivalent settings for GlusterFS,[5] and CephFS. For NFS, all clients mount a shared directory, and a client co-located with the service executes the commands. For GlusterFS we evaluate two different deployments,

[4] The `replicas` mount option of NFS is a client-side fail-over feature, but the replication of the shared data has to be handled independently from the NFS protocol.

[5] GlusterFS experiments over the global partition are executed only once due to the required AWS budget.

local (one region) and global (three regions). Each deployment consists of a *distributed/replicated* volume on top of regular *storage bricks*, one on each of the availability zones for the given EC2 regions. We deployed CephFS only at a single region (3 storage daemons, 1 metadata server, and 3 clients) as cross regions deployments were not well supported [33]. We set the replication factor of GlusterFS, and CephFS to 3.

access □ open ☒ read ■ write ▨ fstat ▨ lstat ▨ lseek ▨ close ▨

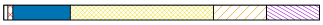
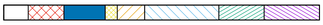
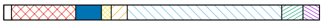
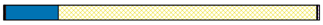
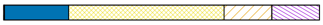
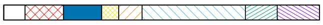
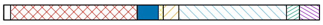
			GlobalFS		**GlusterFS**		**CephFS**
Command	Operations breakdown	**NFS**	*global*	*local*	*global**	*local*	*local*
`tar xzvf bc-1.06.tgz`		1.94s	47.09×	1.36×	149.05×	1.63×	0.17×
`configure`		5.32s	44.66×	2.02×	45.67×	0.96×	0.56×
`make -j 10`		5.9s	29.90×	2.38×	49.34×	1.17×	0.63×
`make`	*(same as above)*	13.14s	20.73×	1.16×	55.20×	0.92×	0.30×
`gzip -d httpd-2.4.12.tgz`		3.87s	117.12×	2.47×	284.75×	0.37×	0.11×
`tar xvf httpd-2.4.12.tar`		60.01s	41.46×	1.08×	99.17×	0.12×	0.14×
`configure -prefix=/tmp`		29.32s	49.35×	2.04×	56.53×	1.34×	0.33×
`make -j 10`		714.37s	2.74×	0.52×	139.68×	0.87×	0.48×
`make`	*(same as above)*	3432.72s	1.82×	0.36×	83.72×	0.50×	0.64×

Table 3.3. Execution times for several compilation workloads on GlobalFS with operations executed over global and local partitions. Execution times are given in seconds for NFS, and as relative times w.r.t. NFS for GlobalFS, GlusterFS and CephFS. *Note that GlusterFS does not support deployments with both global and local partitions; thus, we report results from two separate deployments.

Table 3.3 presents our results. We observe that GlobalFS performs consistently better than GlusterFS when operating across regions. GlobalFS performs competitively against the other file systems across the whole suite of benchmarks. Indeed, GlobalFS is up to 50.9× faster that GlusterFS in compiling Apache `httpd` over the global partition. Note that for the same benchmark on a local partition, GlobalFS is actually faster than NFS. When evaluating GlusterFS and CephFS we use their default, out-of-the-box configuration. Both are heavily optimized systems and some optimizations are on by default (e.g., clients in CephFS use write-back caching, which improves write performance by batching small writes). As expected, the performance penalty for accessing the global partition is higher for write-dominated workloads (extracting an archive, configuring the software package). For read-dominated or compute-intensive (`make`) operations, this overhead decreases because read operations can be completed locally. For comparison purposes, we also tested HDFS (v2.6) with FUSE bindings on a local partition with some of the benchmarks and observed performance in the order as GlobalFS and GlusterFS (e.g., 2.12× slower for the first command as compared to 1.36× and 1.63×, respectively).

Our results demonstrate that GlobalFS performs on par with widely adopted distributed file systems, it ensures a stronger consistency model, it supports replication, and allows users to benefit from locality thanks to its partitioning model.

3.7 Related work

3.7.1 File systems with strong consistency

CalvinFS [105] is a multi-site distributed file system built on top of Calvin [106], a transactional database. Metadata is stored in main memory across a shared-nothing cluster of machines. File operations that modify multiple metadata elements execute as distributed transactions. CalvinFS supports linearizable writes and reads using a single log service to totally order transactions, a mechanism known to scale throughput with the number of nodes within three regions [106]. Using more regions penalize all operations, implying lack of data locality support for CalvinFS. We note that CalvinFS relies on "custom transactions" that group multiple commands into a single operation to boost performance. For example, creating and writing a file, which in POSIX would require three sequential calls (i.e., create, open and write), can be executed as a single transaction in CalvinFS. As a consequence, the POSIX file system API cannot benefit from these optimizations.

CephFS [111] is a file system implementation atop the distributed Ceph block storage [20]. It uses independent servers to manage metadata and link files and directories to blocks stored in the block storage. CephFS is able to scale metadata servers and change the file system partitioning at runtime for load balancing, through its CRUSH [112] extension. Although CephFS supports geographical distribution, WAN deployment over Amazon's EC2 was discouraged by the CephFS developers [33].

The Google File System (GoogleFS) [41] stores data on a swarm of *slave* servers. It maintains metadata on a logically centralized master, replicated on several servers using state machine replication and total ordering of commands using Paxos. GoogleFS is a flat storage system. It does not consider the case of a file system spread over multiple datacenters and the associated partitioning. MooseFS [69] is designed around a similar architecture and has the same limitations. Colossus [25], GoogleFS successor, provides the same strong consistency guarantees, but many of its internal details remain undisclosed.

FhGFS/BeeGFS [9] is distributed file system for high-performance computing clusters that targets read-dominated workloads.

GeoFS [65] is a POSIX-compliant file system for WAN deployments. It exploits user-defined timeouts to invalidate cache entries. Clients pick the desired consistency for files and metadata, as in WheelFS's semantic cues [101].

Farsite [2] is a distributed file system designed to run over a set of desktop workstations. In Farsite, a file system tree is maintained by a group of replicas through byzantine fault-tolerant state-machine replication. Each group can further delegate parts of its namespace (i.e., sub-trees) to other replica groups. File data is replicated separately from metadata: replica groups only keep a list of file replicas and a checksum of the file contents.

Red Hat's GFS/GFS2 [82] and GlusterFS [28] support strong consistency by enforcing quorums for writes, which are fully synchronous. GlusterFS can be deployed across WAN links, but it scales poorly with the number of geographical locations, as it suffers from high-latency links for all write operations.

HDFS [98] is the distributed file system of the Hadoop framework. It is optimized for read-dominated workloads. Data is replicated and sharded across multiple data nodes. A name node is in charge of storing and handling metadata. As for GoogleFS, this node is replicated for availability. The HDFS interface is not POSIX-compliant and it only implements a subset of the specification via a FUSE interface. QuantcastFS [76] is a replacement for HDFS that adopts the same internal architecture. Instead of three-way replication, it exploits Reed-Solomon erasure coding to reduce space requirements while improving fault tolerance. HopsFS [72] is an extension for HDFS that stores metadata in a sharded NewSQL database. Operations become distributed transactions.

SeaweedFS [95] is a distributed file system that follows the design of Haystack [8]. It supports multiple master nodes and multiple metadata managers to locate files. It is optimized for (small) multimedia files.

XtreemFS [51] is a POSIX-compliant system that offers per-object strong-consistency guarantees on top of a set of independent volumes managed by a metadata server (MRC). To best of our understanding, it does not provide a global integrated file system. Furthermore, it does not offer consistency guarantees for inter-volume operations.

PVFS [19] and HDFS can be adapted to support linearizability guarantees for metadata [99] by delegating the storage of the file system's metadata to Berkeley DB [74], which uses Paxos to totally order updates to its replicas.

3.7.2 File systems with weak consistency

There are several distributed file systems for high-performance computing clusters, such as PVFS, PVFS2/OrangeFS [75], Lustre [94], and FhGFS/BeeGFS [9].

These systems have specific (e.g., MPI-based) interfaces and target read-dominated workloads. GIGA+ [80] implements eventual consistency and focus on the maintenance of very large directories. It complements the OrangeFS cluster-based file system.

ObjectiveFS [73] relies on a backing object store (typically Amazon S3) to provide a POSIX-compliant file system with read-after-write consistency guarantees. If deployed on a WAN, ObjectiveFS suffers from long round-trip times for operations such as `fsync` that need to wait until data has been safely committed to S3.

Close-to-open consistency (CTO) was introduced along with client-side caching mechanisms for the Andrew file system and implemented in its open-source implementation OpenAFS [87]. This was a response to previous distributed file systems designs such as LOCUS [110], which offered strict POSIX semantics but with poor performance. Close-to-open semantics are also used by NFS [102], HDFS [98], and WheelFS [101].

Oracle OCFS [97] is a distributed file system optimized for the Oracle ecosystem (e.g., database, application-server). It provides a cache consistency guarantee by exploiting Linux's `O_DIRECT`. Its successor OCFS2 [97] supports the POSIX standard while guaranteeing the same level of cache consistency.

3.7.3 Peer-to-peer file systems

Peer-to-peer file systems target deployments over a large number of independent servers rather than a collection of datacenters. We do not list these systems in Table 3.4 as they are less directly linked to our work, but discuss them below.

A common aspect of peer-to-peer file systems is that they store both metadata and data in the same storage substrate, unlike previously listed approaches and GlobalFS. This storage is typically a DHT. CFS [27] and PAST [85] are early examples of single-writer peer-to-peer file systems, using the Chord [100] and Pastry [86] DHTs. Ivy [70] is an evolution of CFS for multiple writers. The set of writers is static and each writer maintains its own log of modifications to the file system. A reader must causally parse through all writers' logs. Ivy supports eventual but *read-your-write* consistency. CFS and Ivy use immutable blocks, similarly to GlobalFS. Pastis [18] similarly extends PAST for multiple writers. It supports *read-your-write* semantics and *close-to-open* consistency. OceanStore [54] is a DHT-based file system that offers both eventual and strong consistency. It leverages the eventually serializable data storage [36]: weak operations may execute at any replica, while strong operations are totally ordered between writers.

3.7.4 Overview

The characteristics of all surveyed systems are provided in Table 3.4. We categorize file systems by their geographical scaling potential and identify three possible scenarios: file systems that work on LAN (*WoL*) mainly intended for cluster deployments; file systems that support but perform poorly in wide-area network deployments (*WoW*); and file systems that scale in WAN (*SoW*). GlobalFS is the only system to support data locality while at the same time providing strong consistency and geographical scalability.

3.8 Discussion

This chapter introduces GlobalFS, a geographically distributed file system that accommodates locality of access, scalable performance, and resiliency to failures without sacrificing strong consistency. GlobalFS builds on two abstractions: single-site linearizable data stores and an atomic multicast based on Multi-Ring Paxos. This modular design was crucial to handle the complexity of the development, testing, and assessment of GlobalFS. Our evaluation reveals that GlobalFS outperforms other geographically distributed file systems that offer comparable guarantees and delivers performance comparable to single-site networked file systems. We credit GlobalFS performance to its flexible partition model and four execution modes, which allow us to exploit common access patterns and optimize for the most frequent file system operations. These original features distinguish GlobalFS from other distributed file systems and are key to providing geographical scalability without compromising consistency.

In the following, we discuss some limitations and directions for improving the design of GlobalFS:

- *Hard link support*: GlobalFS does not support hard links (i.e., the same file referenced by multiple paths). It could support them in one of two ways: (1) replicate the file in each partition holding a hard link, allowing for expensive writes but fast local reads, or (2) keep a single copy of the file with a reference count and have hard links pointing to it, allowing for faster writes but possibly remote reads.

- *Dynamic partitioning*: The partitioning scheme of GlobalFS could be made dynamic using a mechanism close to the one described in [59]. In a nutshell, the change of partitioning function and related metadata movements would be ordered through atomic multicast, and executed as coordinated multi-partition commands.

- *Asynchronous data movement*: While metadata can be quickly moved between partitions, the data for large files might be slow or expensive to move. Since data blocks in GlobalFS are immutable, it should be possible to move or copy data blocks from remote partitions either on demand by clients or in the background by servers.

- *Caching and pre-fetching*: clients could pre-fetch and locally cache data blocks to speed up sequential reads and frequently accessed files.

- *Small blocks optimization*: Small data blocks could be stored together with metadata. This would allow for very fast access to small files. These inline blocks could later be merged into larger data blocks in the background.

- *Garbage collection*: currently, GlobalFS does not have any mechanism for garbage collection of orphaned data blocks. Since data blocks are not currently shared by multiple files, block references from deleted files could be kept by metadata servers and later erased in the background. The same mechanism could be used for data blocks that are completely overwritten.

Name	**Consistency level**	**POSIX interface**	**Code available**	**Client type**	**Scaling potential**
GlobalFS	_S_	√	√	_User_	_SoW_
AFS [87]	_W,CTO_	×	√	_User_	_WoW_
CalvinFS [105]	_S_	×	×	_User_	_SoW_
CephFS [111]	_S_	√	√	_Kernel,User_	_WoL_
CodaFS [88]	_E_	√	√	_Kernel_	_WoL_
Colossus [25]	_S_	–	×	–	_SoW_
BeeGFS [9]	_S*_	√	√	_User_	_WoL_
Farsite [2]	_S,CTO_	√	×	_Kernel_	_WoL_
GeoFS [65]	_S*,CTO_	√	×	_User_	_WoW_
GFS/GFS2 [82]	–	√	√	_Kernel_	_WoL_
GIGA+ [80]	_E_	√	×	_User_	_WoL_
GlusterFS [28]	_S_	√	√	_User_	_WoW_
GoogleFS [41]	_S_	√	×	–	_SoW_
HDFS [98]	_S, CTO_	×	√	_User_	_SoW_
HopsFS [72]	_S_	×	√	_User_	_SoW_
LOCUS [110]	_S_	√	×	_Kernel_	_WoL_
Lustre [94]	_CH_	√	√	_Kernel_	_WoL_
MooseFS [69]	_S*_	√	√	_User_	_WoL_
NFS/pNFS [102]	_CTO_	√	√	_Kernel_	_WoW_
ObjectiveFS [73]	_RaW_	√	√	_User_	_WoW_
OCFS [97]	_CH_	×	√	_Kernel_	_WoL_
OCFS2 [97]	_CH_	√	√	_Kernel_	_WoL_
PVFS [19]	_RaW_	×	√	_User_	_WoL_
OrangeFS [75]	_RaW_	×	√	_User_	_WoL_
QuantcastFS [76]	_E_	×	√	_User_	_WoL_
SeaweedFS [95]	_S_	×	√	_Kernel_	_WoL_
XtreemFS [51]	_S*_	√	√	_User_	_WoW_
WheelFS [101]	_S,CTO_	√	√	_User_	_WoW_

Table 3.4. Survey of distributed file systems along several criteria: consistency level (Strong=S, Weak=W, Eventual=E, Cache=CH, Close-To-Open=CTO, Read-after-Write=RaW), support of the POSIX standard, code availability, client type (user-space=User, kernel-space=Kernel), scaling potential (Works-on-LAN=WoL, Works-on-WAN=WoW, Scale-on-WAN=SoW). Some properties are unknown (–) or not by default (*).

Chapter 4

A Latency Efficient Atomic Multicast

Distributed systems use replication to tolerate the failure of system components. A fully replicated system, in which every server has a complete copy of the application data, has inherent scalability limits since every server needs to eventually apply every data update. Thus, data partitioning, also known as sharding, is the typical approach used to scale writes in replicated systems. Commonly, in partitioned systems, servers are divided into groups with each group being responsible for storing only a subset of the application data.

Building distributed applications is already a complex endeavor, and data partitioning introduces another issue: coordinating operations across server groups. Atomic multicast is a primitive which greatly simplifies building and reasoning about partitioned systems. It allows for messages to be partially ordered and reliably delivered to only a subset of the system groups. Given a favorable workload, where commands are evenly distributed across partitions and most commands access one or a few partitions, system throughput should scale with the number of partitions. This places the same restriction on the underlying atomic multicast protocol. For atomic multicast to scale, only the sender and destinations of a message can be involved in its ordering. A protocol satisfying this property is said to be genuine. Another crucial aspect of a communication protocol is latency, particularly in geographically distributed systems. Ideally, messages should be delivered at their destinations in as few communication steps as possible from the message being sent.

In this chapter we present PrimCast, the first genuine atomic multicast protocol able to deliver messages *to every destination* in three communication steps. PrimCast uses a primary-based consensus protocol for deciding on local timestamps at each group, but, differently from previous work, does not rely on consensus for advancing and maintaining logical clocks. PrimCast introduces a novel approach, relying on simple quorum intersection, to decide when a given timestamp is safe for delivery. We present the complete algorithm for PrimCast and its proof of correctness. We then show how loosely synchronized clocks can be used to reduce the convoy effect that further delays messages under high system load. We also describe a couple of other extensions to PrimCast that may provide benefits to practical applications. We implemented a prototype of PrimCast and evaluated its performance under various scenarios. Our results show that PrimCast achieves lower latency than state-of-the-art approaches while providing higher or comparable throughput.

The rest of this chapter is organized as follows. Section 4.1 provides the necessary background. Section 4.2 describes PrimCast and presents the complete algorithm. Section 4.3 presents proof for the properties of PrimCast. Section 4.4 describes how to exploit loosely synchronized clocks and presents a couple of other extensions to PrimCast that may benefit practical applications. Section 4.5 presents the results of our experimental evaluation. Section 4.6 reviews related work and Section 4.7 concludes the chapter.

4.1 Background

Users of modern distributed applications typically expect reliability, availability and performance. These properties are often at odds, and system designers must choose the right tradeoff for each application. Some form of replication is generally used to provide fault-tolerance. In a fully replicated system, where every server stores a complete view of the application, write performance is inherently limited by what a single server can do. To circumvent this problem, applications that need to scale must employ *partial replication*. In partially replicated systems, the whole state of the system is partitioned, and each partition is replicated by only a subset of the servers. We refer to such a subset as a *process group*, and the system as a whole is fault-tolerant only if every group is independently fault-tolerant. In a partitioned system, operations may touch one or multiple partitions at once, and ensuring that replicas remain consistent can be a complex effort. Operations or state changes should be reliably propagated to the relevant servers, and concurrent operations must be handled properly so replicas do not diverge.

One particular way of preventing inconsistencies in replicated systems involves totally ordering messages. *Atomic broadcast* is a communication protocol that allows messages to be reliably sent to every replica in the system, and a strict total order is imposed on the order of their delivery. It is a convenient primitive for fully replicated systems, greatly simplifying reasoning about their correctness. For partially replicated systems, though, in which application data is partitioned and each process stores only a subset of the application data, atomic broadcast is a scalability bottleneck. *Atomic multicast* is a primitive that allows messages to be delivered by only a subset of the groups of the system. A trivial implementation of atomic multicast consists of using atomic broadcast to send messages and then simply discarding the messages a process is not interested in. That obviously is not a big improvement over atomic broadcast, as totally ordering messages is inherently not scalable. For any given message, if a process is neither multicasting nor delivering it, the process should not have to take steps for the message to be delivered. An atomic multicast protocol that satisfies this property is said to be *genuine* [46]. Such protocols are of particular interest to geographically distributed applications. A genuine protocol allows for throughput to scale with the number of groups, if the workload allows for it. Furthermore, if the workload exhibits locality of access, servers in a group can be placed close to clients and each other, allowing for low latency of local operations, as discussed in Section 3.1.

4.1.1 Timestamp-based message ordering

Most protocols for atomic multicast achieve a partial ordering of messages by assigning them timestamps, and then having processes deliver messages in timestamp order. The basic idea was first proposed in what is known as Skeen's protocol [15], as a solution for atomic multicast between individual non fault-tolerant processes. It works as follows:

1. Each process in the system has its own logical clock.
2. A message m, destined to processes in $m.dest$, is sent to each one of these processes.
3. A process that receives m increments its logical clock and then assigns m a *local timestamp* from its clock. The local timestamp is then sent to other processes in $m.dest$
4. Once a process gets a local timestamp from each process in $m.dest$, the maximum of the local timestamps is chosen as m's *final timestamp*. The process then updates its clock to m's final timestamp, if not already past it.
5. Message m can then be delivered by process p once no pending message (i.e., those assigned a local timestamp by p but not yet delivered) has a

possibly smaller final timestamp than m.

Skeen's protocol is genuine, as only the sender and processes in $m.dest$ take steps to deliver m. The complete protocol is shown in Algorithm 4 (ignoring the parts in gray).

In [40], Fritzke et al. propose a solution for fault-tolerant atomic multicast. This solution is further refined in [90]. The core idea is to replace individual processes in Skeen's protocol with fault-tolerant process groups. Inside each group, atomic broadcast (i.e., consensus) is used to both maintain the group's logical clock and to timestamp messages. In essence, steps 2, 3 and 4 in Skeen's algorithm are modified. In step 2, message m must be independently atomically broadcast to each group in $m.dest$. In step 3, using the delivery order of atomic broadcast, each group decides on its next clock value and m's local timestamp. Finally, in step 4, once a group knows all local timestamps for m, the final timestamp for m is atomically broadcast locally to update the group's logical clock and allow for m to be delivered. We refer to protocols that rely on assigning message timestamps as timestamp-based.

4.1.2 Collision-free and failure-free latency

In [43], Gotsman et al. propose two metrics for describing the delivery latency in atomic multicast protocols: failure-free and collision-free latency. We consider the *delivery latency* of a message as the time between its a-multicast and its last a-delivery, that is, the time for it to be a-delivered at every correct destination. [1] Both collision-free and failure-free latencies set bounds on the delivery latency of messages in periods of system stability. More precisely, we consider the system stable after some unknown time t, past GST (Section 2.1.2), when there are no process failures, message delay is bounded, group leaders are stable and there is no ongoing or future reconfiguration due to previous leader changes. For simplicity, we assume that local computation takes no time and message delay between any two processes is fixed after t (a communication step). *Collision-free latency* is the maximum delivery latency for delivering a message when there are no conflicting concurrent messages. A message m is *concurrent* with another message m' if m is a-multicast before m' is first a-delivered, and m' is a-multicast before m is first a-delivered. Two messages m and m' are *conflicting* iff $m.dest \cap m'.dest \neq \emptyset$. In the presence of concurrent messages, message delivery may be subject to a *convoy effect* [4, 16], where the delivery of a message needs to wait for other messages to be delivered. *Failure-free latency* is the maximum delivery latency

[1] This differs from the definition in [43], which considers the first a-delivery of a message.

for delivering a message in the presence of concurrent messages. In practice, in periods of system stability, the failure-free and collision-free latencies can be seen as the worst and best case delivery latencies of an atomic multicast protocol, respectively.

Still in [43], a method is proposed to calculate the collision-free and failure-free latency values in timestamp-based atomic multicast protocols. First, two values must be obtained from the algorithm: the *clock update latency C* and the *commit latency D*. Let m be a message that is multicast at time t (as defined in the previous paragraph). The clock update latency C is the maximum delay after which no other message can be assigned a local timestamp higher than m's final timestamp. Essentially, the clock update latency limits the interval in which conflicting concurrent messages can be a-multicast. The commit latency D of m is the maximum delay after which a destination knows the final timestamp of m and has its group's logical clock value equal to or higher than m's final timestamp. In the absence of conflicting concurrent messages, m can be delivered after $t+D$, and the collision-free latency is thus equal to D. The earliest another conflicting message can be multicast before m is $t + C$ (otherwise it would have a larger final timestamp). Thus, after $t + C + D$, every final timestamp smaller than m's must be known, and the respective messages can be delivered together with m. Hence, the failure-free latency is equal to $C + D$.

We now analyze the latency values for the classic protocols of [40, 90], assuming that some Paxos [58] based protocol is used as the consensus protocol inside each group. For every destination to know the final timestamp ts of a message m, four communication steps are needed: one for the timestamp proposal to reach the consensus leader, two for the consensus decision, and one more for groups to exchange local timestamps. Then, once the final timestamp is known at the group's consensus leader, two more communication steps are needed for consensus to update the logical clock to ts at the group members. Thus, the commit latency D of these protocols is six. Since local timestamps smaller than ts can be proposed by destinations with a logical clock value smaller than ts, the clock update latency C is also six. The collision-free and failure-free latencies values are thus six and twelve communication steps, respectively.

4.2 PrimCast

PrimCast is a genuine atomic multicast protocol that achieves collision-free and failure-free latency of three and five communication steps, respectively, *at every destination*. Previous work achieved these latency values only at group leaders [43]. In this section, we discuss PrimCast's basic ideas and then present the algorithm in detail.

4.2.1 Basic ideas

PrimCast is based on the following ideas:

- *Primary-based consensus at each group:* Each group in PrimCast employs a primary-based consensus protocol. Similarly to other primary-based protocols [52, 64], PrimCast is epoch based. Each epoch is owned by a single process in the group. Inside a group, each process tracks its current epoch, and only accepts proposals from the primary of that epoch. In the absence of failures, when processes in a group follow the same epoch, advancing the logical clock of the primary to a given value is enough to ensure new messages are assigned a larger local timestamp. Hence, for any given message m, after two communication steps (i.e., the time for group primaries to exchange their timestamp proposals), no other message can be assigned a local timestamp smaller than the final timestamp of m. The clock update latency C is thus two communication steps.

- *Quorum-based logical clocks:* One of the requirements for a message m to be safely delivered at a given destination is that no new message targeting that same destination should be assigned a smaller final timestamp than m's. Updating the logical clock of primaries is enough to prevent this situation in the failure-free case, but when primaries change, this is not enough. To ensure safety in the presence of failures, previous approaches rely on consensus to agree on the group's logical clock, and delivery of a message m can only happen at a given process after its group's logical clock is larger than or equal to m's final timestamp. In PrimCast, instead of relying on consensus for logical clock agreement, a quorum-based approach is used. Inside each group, processes track each other's clock values. On an epoch change, the new primary must pick a clock value larger than all values seen in some quorum of clocks from previous epochs. When a message m is multicast, by carefully exchanging and tracking clock values, every destination can have its group's logical clock advanced past m's final timestamp in three communication steps.

- *Cross-group quorum tracking:* Instead of exchanging local timestamps after consensus is reached inside each group, PrimCast replicas directly send acknowledgment messages to other destination groups. Each destination process individually tracks when the quorum a for local timestamp from another group is reached. Every local timestamp for a given message is thus learned in three communication steps at every destination. This, together with quorum-based logical clocks, ensures the commit latency D is three communication steps at every destination.

4.2.2 Algorithm

PrimCast is presented in Algorithm 1 (initialization and predicates), Algorithm 2 (main logic), and Algorithm 3 (primary change logic). Processes communicate through the *r-multicast* and *r-deliver* primitives of FIFO non-uniform reliable broadcast (see Section 2.2.1), which can deliver messages in one communication step. In the following, we give an overview of the algorithm and provide some insight into how it achieves safety.

A note on epochs

PrimCast employs a primary-based protocol inside each group to assign local timestamps to messages. The protocol proceeds in epochs, a given epoch $\mathcal{E}$ being owned by a single process p, the epoch leader. If a quorum of processes accept $\mathcal{E}$ as their *current epoch* ($\mathcal{E}_{cur} = \mathcal{E}$), p may become the effective primary. Epochs from different groups are not related: each group has its own set of epochs, and advances epochs independently of other groups.

Assigning timestamps

To a-multicast a message m, the sender r-multicasts $\langle \textsc{start}, m \rangle$ to each destination in $\bigcup m.dest$ (line 31). When r-delivered, the tuple is added to the $\mathcal{M}$ set. The primary for each group in $m.dest$ will eventually update its $clock$, pick a timestamp for m, append the proposal to $\mathcal{T}$ and send the respective ACK to every destination in $m.dest$ (line 35).

When a process $p \in g$ r-delivers an ACK for m coming from a process in its own group g (line 40), p first stores the tuple in $\mathcal{M}$. Then, if the ACK is coming from the primary of its current epoch, p accepts the timestamp proposal by appending it to $\mathcal{T}$, updates its *clock* if needed, and then also r-multicasts its own ACK to every destination in $m.dest$. When p instead r-delivers an ACK for m coming from a process in a remote group h (i.e., $p \notin h$), p simply stores the tuple in $\mathcal{M}$ (line 46).

Algorithm 1 PrimCast initialization and definitions at process $p \in g$.

```
initialization:
    $\mathscr{M} \leftarrow \emptyset$    ▷ set of r-delivered START, ACK and BUMP tuples
    $\mathscr{D} \leftarrow \emptyset$    ▷ set of a-delivered messages
    $\mathscr{T} \leftarrow \emptyset$    ▷ sequence of tuples for timestamps proposed in $g$ (in the format $\langle \mathscr{E}, m, ts \rangle$)
    $clock \leftarrow 0$    ▷ $p$'s clock value
    $\mathscr{E}_{cur} \leftarrow$ initial epoch    ▷ current epoch
    $\mathscr{E}_{prom} \leftarrow$ initial epoch    ▷ promised epoch (always $\geq \mathscr{E}_{cur}$)
    $state \leftarrow$ PRIMARY if leader($\mathscr{E}_{cur}$) $= p$ else FOLLOWER

local-ts($m, h$) $\equiv$    ▷ local timestamp for $m$ in $h$ if known, otherwise $\bot$
    if $\exists ts, \mathscr{E}', quorum \in Q_h : \forall q \in quorum : \langle \text{ACK}, m, h, \mathscr{E}', ts, q \rangle \in \mathscr{M}$ then $ts$
    else $\bot$

final-ts($m$) $\equiv$    ▷ max of all local-ts in $m.dest$ if all are decided, otherwise $\bot$
    if $\forall h \in m.dest$ : local-ts($m, h$) $\neq \bot$ then $\max_{h \in m.dest}$(local-ts($m, h$))
    else $\bot$

min-clock($q$) $\equiv$    ▷ highest $ts$ seen in messages from $q$ in epoch $\mathscr{E}_{cur}$ or earlier
    max($\{0\} \cup \{ts \mid \exists \mathscr{E}' \leq \mathscr{E}_{cur} : \langle \text{ACK}, \_, g, \mathscr{E}', ts, q \rangle \in \mathscr{M}$ or $\langle \text{BUMP}, \mathscr{E}', ts, q \rangle \in \mathscr{M}\}$)

quorum-clock() $\equiv$    ▷ lower bound for clock of the primary of epochs higher than $\mathscr{E}_{cur}$
    max($\{ts \mid \exists quorum \in Q_g : \forall q \in quorum$ : min-clock($q$) $\geq ts\}$)

min-ts($m$) $\equiv$    ▷ minimum possible value for final-ts($m$)
    max(if $\exists h$ : local-ts($m, h$) $\neq \bot$ then $\max_{h \in m.dest}$(local-ts($m, h$)) else 0,
        min(if $\exists \mathscr{E}, ts : \langle \mathscr{E}, m, ts \rangle \in \mathscr{T}$ then $ts$ else $\infty$,    ▷ any local-ts is a lower bound, so
            1 + min-clock(leader($\mathscr{E}_{cur}$)),    ▷ is the minimum possible proposal for $m$ in $g$
            1 + quorum-clock()))

proposable($m$) $\equiv$    ▷ $m$ is not decided or proposed in $g$
    $\langle \text{START}, m \rangle \in \mathscr{M}$ and local-ts($m, g$) $= \bot$ and $\langle \_, m, \_ \rangle \notin \mathscr{T}$

deliverable($m$) $\equiv$
    $m \notin \mathscr{D}$ and final-ts($m$) $\neq \bot$ and    ▷ $m$ has not been delivered and has a final timestamp
    final-ts($m$) $\leq$ min-clock($leader(\mathscr{E}_{cur})$) and    ▷ smaller than new proposals in $\mathscr{E}_{cur}$
    final-ts($m$) $\leq$ quorum-clock() and    ▷ and smaller than proposals in newer epochs
    $\forall m' : \langle \_, m', \_ \rangle \in \mathscr{T}, m' \notin \mathscr{D}, m' \neq m : \langle$final-ts($m$)$, m.id \rangle < \langle$min-ts($m'$)$, m'.id \rangle$
        ▷ and smaller than the possible timestamp of any other pending $m'$
```

Algorithm 2 PrimCast algorithm at process $p \in g$.

```
a-multicast($m$):                          ▷ process $p$ wants to atomically multicast $m$ to $m.dest$
    r-multicast($\langle \text{START}, m \rangle$) to $m.dest$

when r-deliver($\langle \text{START}, m \rangle$):
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle \text{START}, m \rangle\}$

when $\exists m$ : proposable($m$) and $state = \text{PRIMARY}$:   ▷ primary proposes local timestamp in $g$
    for each $m$ : proposable($m$)
        $clock \leftarrow clock + 1$
        $\mathcal{T} \leftarrow \mathcal{T} \bullet \langle \mathcal{E}_{cur}, m, clock \rangle$
        r-multicast($\langle \text{ACK}, m, g, \mathcal{E}_{cur}, clock, p \rangle$) to $m.dest$

when r-deliver($\langle \text{ACK}, m, h, \mathcal{E}, ts, q \rangle$) and $g = h$:        ▷ on ACK from our group
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle \text{ACK}, m, h, \mathcal{E}, ts, q \rangle\}$
    if $q =$ leader($\mathcal{E}$) and $\mathcal{E} = \mathcal{E}_{cur}$ and $state = \text{FOLLOWER}$ then   ▷ if ACK from primary
        $\mathcal{T} \leftarrow \mathcal{T} \bullet \langle \mathcal{E}_{cur}, m, ts \rangle$           ▷ send our own ack
        $clock \leftarrow \max(clock, ts)$
        r-multicast($\langle \text{ACK}, m, g, \mathcal{E}_{cur}, ts, p \rangle$) to $m.dest$

when r-deliver($\langle \text{ACK}, m, h, \mathcal{E}, ts, q \rangle$) and $g \neq h$:     ▷ on ACK from remote group
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle \text{ACK}, m, h, \mathcal{E}, ts, q \rangle, \langle \text{START}, m \rangle\}$
    if $ts > clock$ then                   ▷ on a remote ACK with $ts$ higher than our $clock$
        $clock \leftarrow ts$                     ▷ update $clock$ and inform our group
        r-multicast($\langle \text{BUMP}, \mathcal{E}_{prom}, clock, p \rangle$) to $g$

when r-deliver($\langle \text{BUMP}, \mathcal{E}, ts, q \rangle$):
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{\langle \text{BUMP}, \mathcal{E}, ts, q \rangle\}$

when $\exists m$ : deliverable($m$) and $state \in \{\text{PRIMARY}, \text{FOLLOWER}\}$:
    for each $m$ : deliverable($m$)
        $\mathcal{D} \leftarrow \mathcal{D} \cup \{m\}$
        a-deliver($m$)                           ▷ deliver $m$ to the application
```

Algorithm 3 PrimCast primary change algorithm at process $p \in g$.

```
when $\Omega_g = p$ and $state \notin \{\text{PRIMARY}, \text{CANDIDATE}\}$:
    $state \leftarrow \text{CANDIDATE}$
    $\mathscr{E}_{prom} \leftarrow$ next epoch higher than $\mathscr{E}_{prom}$ for which $p$ is the leader
    r-multicast($\langle \text{NEW-EPOCH}, \mathscr{E}_{prom} \rangle$) to $g$

when r-deliver($\langle \text{NEW-EPOCH}, \mathscr{E} \rangle$) and $\mathscr{E} \geq \mathscr{E}_{prom}$:
    if $p \neq$ leader($\mathscr{E}$) then $state \leftarrow \text{PROMISED}$
    $\mathscr{E}_{prom} \leftarrow \mathscr{E}$
    r-multicast($\langle \text{PROMISE}, \mathscr{E}, p, clock, \mathscr{E}_{cur}, \mathscr{T} \rangle$) to $p$

when $state = \text{CANDIDATE}$ and PROMISEs for $\mathscr{E}_{prom}$ from a $quorum \in Q_g$ were r-delivered:
    $\mathscr{E}_{max} \leftarrow$ highest epoch in promises
    $\mathscr{T}_{max} \leftarrow$ longest state from promises with $\mathscr{E}_{max}$    ▷ get $\mathscr{T}$ from most up-to-date replica
    $ts \leftarrow$ maximum clock from promises                                         ▷ see predicate quorum-clock
    r-multicast($\langle \text{NEW-STATE}, \mathscr{E}_{prom}, \mathscr{T}_{max}, ts \rangle$)

when r-deliver($\langle \text{NEW-STATE}, \mathscr{E}, \mathscr{T}', ts \rangle$) and $\mathscr{E} = \mathscr{E}_{prom}$:
    $\mathscr{T} \leftarrow \mathscr{T}'$
    $\mathscr{E}_{cur} \leftarrow \mathscr{E}$                                          ▷ move $p$ to $\mathscr{E}_{cur}$
    $clock \leftarrow \max(clock, ts)$
    r-multicast($\langle \text{ACCEPT}, \mathscr{E}_{cur}, p \rangle$) to $g$           ▷ inform other replicas we're at $\mathscr{E}_{cur}$

when $state \in \{\text{PROMISED}, \text{CANDIDATE}\}$ and $\mathscr{E}_{cur} = \mathscr{E}_{prom}$ and          ▷ when $p$ is at $\mathscr{E}_{cur}$ and
        ACCEPTs for $\mathscr{E}_{cur}$ from some $quorum \in Q_g$ were r-delivered:    ▷ so is a $quorum \in Q_g$
    if $state = \text{PROMISED}$ then $state \leftarrow \text{FOLLOWER}$
    if $state = \text{CANDIDATE}$ then $state \leftarrow \text{PRIMARY}$
    for each $\langle \mathscr{E}, m, ts \rangle$ in $\mathscr{T}$
        if $\langle \text{ACK}, m, g, \mathscr{E}, ts, p \rangle \notin \mathscr{M}$ then    ▷ send ACKs we have not yet sent (in $\mathscr{T}$'s order)
            r-multicast($\langle \text{ACK}, m, g, \mathscr{E}, ts, p \rangle \notin \mathscr{M}$) to $m.dest$
```

The local timestamp of m for h is tracked by local-ts(m, h) (line 9). The value is decided when, in $\mathscr{M}$, there are ACKs for m from a quorum of processes in h, all coming from the same epoch. The final-ts(m) is decided once local-ts(m, g) is decided for every $g \in m.dest$. When primaries are stable, after three communication steps, every correct destination in $\bigcup m.dest$ will have received an ACK for m from every other correct destination, ensuring final-ts(m) is decided.

Delivering a message

A message m can only be safely delivered when (1) the process knows final-ts(m), (2) every message with a smaller final timestamp has been delivered, and (3) no message may yet be assigned a smaller final timestamp. At a given process $p \in g$, these conditions are tracked by the deliverable(m) predicate (line 26), with message ids being used to break ties. This predicate depends on the following definitions:

- final-ts(m) (line 12): the final timestamp of m is known once the local-ts(m, h) (i.e., the local timestamp) for all $h \in m.dest$ are known.
- min-clock(q) (line 15): the maximum clock value seen in messages from process q, from epochs smaller or equal to $\mathscr{E}_{cur}$.
- quorum-clock() (line 17): lower bound for the starting clock of primaries for epochs higher than $\mathscr{E}_{cur}$ in the process's group. For a process p to become the primary for its group g, it must first obtain a quorum of promises for the new epoch from processes in g (line 65). The largest clock value seen in the set of received promises is chosen as the starting clock value of the new epoch (line 68). As an example, consider a group g of 5 processes with simple majority quorums (i.e., any 3 processes is a quorum). Suppose a new leader p starts an epoch $\mathscr{E}$, gets a promise from each process in g (including itself), and the set of clock values gathered from the promises is $\{1, 2, 3, 4, 5\}$. The minimum clock value that can be picked by p for the new epoch is 3, which comes from the quorum of promises with values $\{1, 2, 3\}$. From quorum intersection, there is a quorum of promises for which all clock values ($\{3, 4, 5\}$) are higher than or equal to 3. Thus, processes rely on quorum-clock() to know when a given timestamp is safe for delivery in epochs higher than $\mathscr{E}_{cur}$. For this reason, min-clock(q) ignores tuples coming from epochs higher than $\mathscr{E}_{cur}$.
- min-ts(m) (line 19): lower bound for the final timestamp of message m. Any known local timestamp for m is a lower bound. At process $p \in g$,

when local-ts(m, g) is not yet known, a lower bound can be inferred for its future value. The value of local-ts(m, g) will come either from the current primary (i.e., equal to the proposal in $\mathscr{T}$ or higher than 1+min-clock(leader($\mathscr{E}_{cur}$))) or from the primary of some future epoch (i.e., higher than quorum-clock()).

Propagating clock values inside a group

Instead of relying on consensus to maintain a group's logical clock, PrimCast instead carefully tracks the clock values from processes in the group, and uses quorum intersection to ensure safety during epoch changes (see the explanation for quorum-clock() in the previous section). Clock values are propagated in two ways: implicitly through ACK messages or through BUMP messages. Whenever p receives an ACK from q, it will update its own clock if needed. The ACK also updates what p knows about q's clock value. Processes in g will exchange ACKs for the local timestamp of m in g among themselves. This exchange is enough to both (1) move the clocks of processes in g past the local timestamp for m in g and (2) inform processes about the updated clock values. For a message m to be delivered, a process must know that a quorum of clocks in its group is past the final timestamp of m. When the local timestamp for some remote group is the largest for m, the ACKs alone are not enough to ensure delivery. Thus, when an ACK from a remote group is received, a process updates its clock if needed and sends a BUMP message to its group (line 50). Note that BUMP messages carry the sender's promised epoch, $\mathscr{E}_{prom}$: once a process is promised to epoch $\mathscr{E}$ it cannot influence the quorum-clock() calculation for epochs lower than $\mathscr{E}$.

Example execution

Figure 4.1 shows some of the messages sent during an example execution of the protocol. In the example we have two groups, $g = \{p_1, p_2, p_3\}$ and $h = \{p_4, p_5, p_6\}$, and we consider simple majority quorums for each. Processes p_1 and p_4 are the primaries of $\mathscr{E}_g$ and $\mathscr{E}_h$ respectively. Process p_5 does a-multicast(m), where $m.dest = \{g, h\}$. The diagram only shows the messages needed for process p_2 to a-deliver m. As it can be seen, in the absence of concurrent messages, m can be a-delivered by p_2 in three communication steps. The example also shows why BUMP messages are needed. Without the BUMP messages in $\mathscr{M}$, the value of quorum-clock() at p_2 would be equal to 1, preventing m from being delivered since final-ts$(m) = 2$.

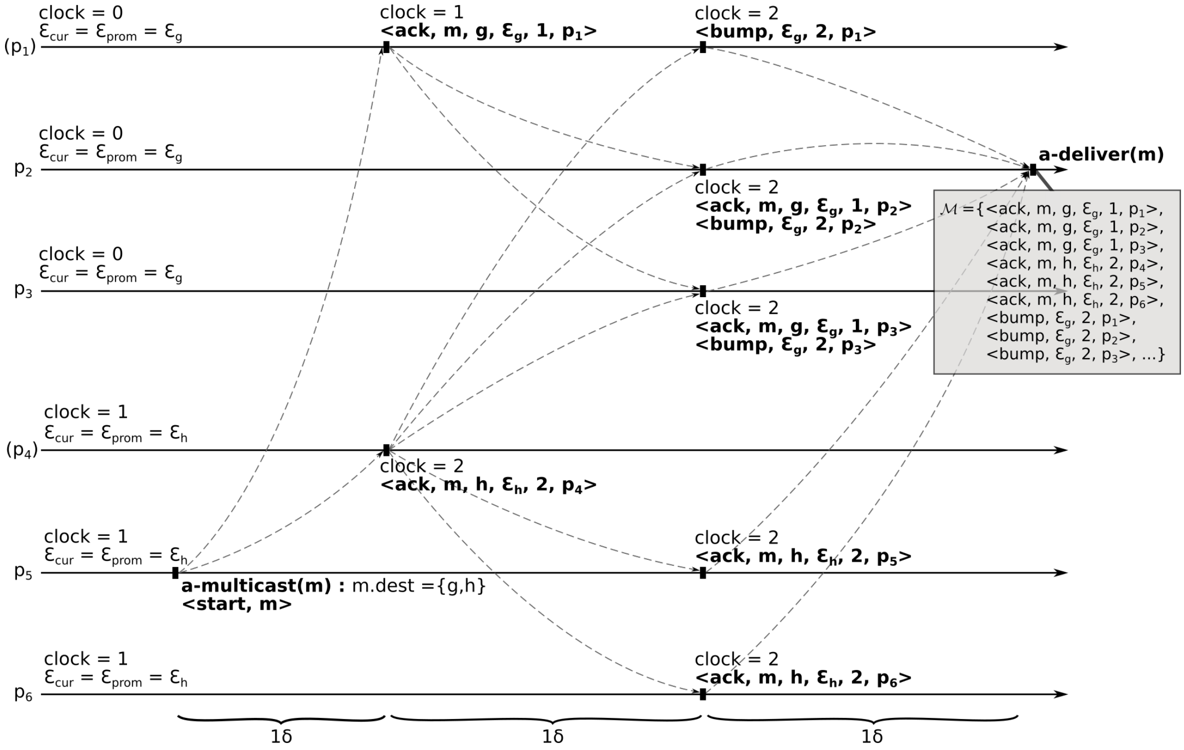

clock = 0
$\mathcal{E}_{cur} = \mathcal{E}_{prom} = \mathcal{E}_g$
(p_1)
clock = 1
<ack, m, g, $\mathcal{E}_g$, 1, p_1>
clock = 2
<bump, $\mathcal{E}_g$, 2, p_1>
clock = 0
$\mathcal{E}_{cur} = \mathcal{E}_{prom} = \mathcal{E}_g$
p_2
clock = 2
<ack, m, g, $\mathcal{E}_g$, 1, p_2>
<bump, $\mathcal{E}_g$, 2, p_2>
a-deliver(m)
$\mathcal{M}$ ={<ack, m, g, $\mathcal{E}_g$, 1, p_1>,
<ack, m, g, $\mathcal{E}_g$, 1, p_2>,
<ack, m, g, $\mathcal{E}_g$, 1, p_3>,
<ack, m, h, $\mathcal{E}_h$, 2, p_4>,
<ack, m, h, $\mathcal{E}_h$, 2, p_5>,
<ack, m, h, $\mathcal{E}_h$, 2, p_6>,
<bump, $\mathcal{E}_g$, 2, p_1>,
<bump, $\mathcal{E}_g$, 2, p_2>,
<bump, $\mathcal{E}_g$, 2, p_3>, ...}
clock = 0
$\mathcal{E}_{cur} = \mathcal{E}_{prom} = \mathcal{E}_g$
p_3
clock = 2
<ack, m, g, $\mathcal{E}_g$, 1, p_3>
<bump, $\mathcal{E}_g$, 2, p_3>
clock = 1
$\mathcal{E}_{cur} = \mathcal{E}_{prom} = \mathcal{E}_h$
(p_4)
clock = 2
<ack, m, h, $\mathcal{E}_h$, 2, p_4>
clock = 1
$\mathcal{E}_{cur} = \mathcal{E}_{prom} = \mathcal{E}_h$
p_5
a-multicast(m) : m.dest ={g,h}
<start, m>
clock = 2
<ack, m, h, $\mathcal{E}_h$, 2, p_5>
clock = 1
$\mathcal{E}_{cur} = \mathcal{E}_{prom} = \mathcal{E}_h$
p_6
clock = 2
<ack, m, h, $\mathcal{E}_h$, 2, p_6>
1δ
1δ
1δ

Figure 4.1. Example execution of PrimCast, only showing the messages needed for process p_2 to a-deliver a message a-multicast by process p_5.

Changing a group's primary

When a process $p \in g$ has $\Omega_g = p$, if p is not already the leader (line 57), p starts an epoch change. The new leader p starts by picking an epoch $\mathcal{E}$ higher than the one it is promised to ($\mathcal{E}_{prom}$). It then becomes a CANDIDATE, sending a NEW-EPOCH message to processes in g. Any process that receives the NEW-EPOCH, if $\mathcal{E}$ is larger than its promised epoch, becomes PROMISED to $\mathcal{E}$ (line 61) and then sends its current state ($\mathcal{E}_{cur}$, $\mathcal{T}$ and $clock$) to p. Once p gets a quorum of promises for $\mathcal{E}$ (line 65), it must pick the most up-to-date state from the promises received: out of all the promises with the highest current epoch, it picks the longest $\mathcal{T}$. Then, p picks the highest clock value out every promise as the starting clock value of the new epoch $\mathcal{E}$. Before p becomes the primary, it must ensure that the new epoch state is safe in a quorum of processes in g. Thus, p sends a NEW-STATE message to all processes in g (line 69). When a process receives the NEW-STATE for the epoch it is promised to (line 70), it installs the state by setting $\mathcal{T}$ and $\mathcal{E}_{cur}$, updates its clock, and then sends an ACCEPT message to every process in g. Once a process has $\mathcal{E}_{cur} = \mathcal{E}$ and receives a quorum of ACCEPTs for $\mathcal{E}$ (line 76), it either becomes the PRIMARY (in case of p) or a FOLLOWER. Finally, for each tuple present in $\mathscr{T}$, in order, if the process has not yet sent the respective ACK (i.e., the ack is not present in $\mathscr{M}$), it sends it to the relevant destinations (line 80).

On liveness

Eventually, from the properties of the leader election oracle Ω_g and our model assumptions (Section 2.1), at each group g, the same correct process $p \in g$ is forever output by Ω_g at every process in g. If p is not the primary of g then, from the algorithm (line 57), it will start a new epoch and eventually become the primary. Any message m destined to g that is a-multicast by a correct processes, if not yet proposed or delivered in g (lines 24 and 35), will eventually be present in p's $\mathscr{M}$ set and be proposed by p. Since no other process in g starts a new epoch, and p is correct, m is eventually assigned a local timestamp in g. The same is true for each other group in $m.dest$ and for messages with a smaller timestamp than m in g, thus m is eventually assigned a final timestamp and delivered at g. In practice, it is enough that primaries at each group are stable for periods long enough for local timestamps to be decided and propagated to other groups.

4.3 PrimCast correctness

Definition 1. *Global and local messages: we say a message m is global if $|m.dest| \geq 2$, otherwise, m is a local message (destined to a single group).*

Definition 2. *Final timestamp: Let m be a message that is a-delivered by some process p. We refer to the value of final-ts(m) at the time m is delivered as m's final timestamp.*

Lemma 1. *Stable epoch and primary: Every group g has an epoch $\mathscr{E}_l$ with leader l such that, eventually:*

- *perpetually, at every correct process in g, $\mathscr{E}_{cur} = \mathscr{E}_{prom} = \mathscr{E}_l$*
- *process $l \in g$, leader of $\mathscr{E}_l$, is correct*
- *perpetually, at l, $state =$ PRIMARY. At every other correct process in g, $state =$ FOLLOWER*

We call $\mathscr{E}_l$ the stable epoch and l the stable primary.

Proof. From the properties of the Ω_g failure detector (Section 2.1.2), there is a time t' after which Ω_g outputs the same correct process l at every correct process in g. Let t be some time after t' where only correct processes may send messages ($t >$ GST). After t, no other NEW-EPOCH messages will be r-multicast by processes other than l, since $\Omega_g = l$ (line 60). Let $\mathscr{E}$ be the highest epoch for which a NEW-EPOCH message is r-multicast before t and is eventually r-delivered by some correct process (in case no such message exists, let $\mathscr{E}$ be the initial epoch in g). There are two cases to consider:

- *l is the leader of $\mathscr{E}$:* If $\mathscr{E}$ is the initial epoch, correct processes already satisfy the conditions in the lemma. Otherwise, since no NEW-EPOCH for a higher epoch can be r-multicast, every correct process in g (other than l) will set $\mathscr{E}_{prom} = \mathscr{E}$ and $state =$ PROMISED (line 61) upon r-delivering the message. Eventually, l will get a quorum of promises for $\mathscr{E}$ (line 65) and r-multicast the NEW-STATE for $\mathscr{E}$. Processes that r-deliver the NEW-STATE will set $\mathscr{E}_{cur} = \mathscr{E}$ and then r-multicast an ACCEPT for $\mathscr{E}$ to processes in g (line 74). Once a process gets a quorum of ACCEPT messages for its $\mathscr{E}_{cur}$, it will move its $state$ to FOLLOWER (or PRIMARY in the case of l) (line 76).

- *l is not the leader of $\mathscr{E}$:* When l r-delivers the NEW-EPOCH for $\mathscr{E}$, it sets $\mathscr{E}_{prom} = \mathscr{E}$ and becomes PROMISED (line 61). Thus, after time t, since $\Omega_g = l$, process l will eventually propose an epoch $\mathscr{E}'$ that is higher than $\mathscr{E}$ (line 57). Every correct process in g will eventually r-deliver the NEW-EPOCH for $\mathscr{E}'$ and, by the same argument of the previous case, every correct process in g eventually has $\mathscr{E}_{cur} = \mathscr{E}_{prim} = \mathscr{E}'$ and is either a FOLLOWER or the PRIMARY (in the case of l), satisfying the conditions in the lemma.

□

Proposition 1. *Genuineness: for any admissible run $\mathscr{R}$ of the algorithm and for any process p, if p sends or receives a message in $\mathscr{R}$, then some message m is a-multicast and either p does a-multicast(m) or $p \in \bigcup m.dest$.*

Proof. For the purposes of this proof, we ignore messages sent due to a group's epoch changing (in Algorithm 3). These are only exchanged internally in each group g, and only when Ω_g changes output at some process in g. As per Lemma 1, epoch changes eventually stop happening. We note that the algorithm could be modified so that an epoch change in g is only triggered when the new leader has messages in $\mathscr{M}$ not yet delivered, but we chose not to so as to simplify the algorithm and proofs. There are only 3 kinds of messages that are sent or received in Algorithm 2:

- $\langle \textsc{start}, m \rangle$: Only sent by the process that a-multicasts a message m (line 32), to processes in $\bigcup m.dest$.

- $\langle \textsc{ack}, m, g, \mathscr{E}, clock, p \rangle$: An ACK message for m is either sent by the primary of a group in $g \in m.dest$ (line 39), or by a follower in g that receives the ACK from its primary (line 45). In both cases, an ACK for m is sent only to processes in $\bigcup m.dest$.

- $\langle \textsc{bump}, \mathscr{E}, clock, p \rangle$: A BUMP message is only sent by a processes in g that receives an ACK for some message m from a group other than g (line 50), hence, $g \in \bigcup m.dest$. The BUMP is only sent to processes in g.

□

Proposition 2. *The collision-free and failure-free delivery latencies of PrimCast, at any destination, are 3 and 5 communication steps, respectively.*

Proof. A message m that is a-multicast after t can be a-delivered by a process $p \in \bigcup m.dest$ as soon as deliverable(m) is true at p. Consider a time t after GST where, at each correct process in each group g, $\mathscr{E}_{cur}$ is equal to g's stable epoch, and $state \in \{\textsc{primary}, \textsc{follower}\}$ (Lemma 1). The deliverable(m) predicate is a conjunction of the following clauses:

1. $m \notin \mathscr{D}$ **and** final-ts(m) $\neq \perp$ (line 27): The final-ts(m) is known once the local-ts(m, g) for each $g \in m.dest$ is known. local-ts(m, g) is known once p receives ACKs from the same epoch from a quorum of processes in g (line 9). The primary of g sends its ACK for m to every destination once it receives $\langle \textsc{start}, m \rangle$ (line 39). The followers in g send their own ACKs to every destination after receiving the ACK from the primary (line 45). Thus, after 3 communication steps, p will have received a quorum of ACKs for m from g, and local-ts(g, m) is known.

2. final-ts(m) $\leq$ min-clock($leader(\mathscr{E}_{cur})$) (line 28): Let g be a group such that local-ts(m, g) $=$ final-ts(m) at p. If $p \in g$, then the clause is true once p receives the ACK from its primary, in 2 communication steps. Otherwise, the primary of p's group will receive the ACK from g's primary in 2 communication steps, and then send a BUMP message to p (line 50), ensuring the clause holds in at most 3 communication steps.

3. final-ts(m) $\leq$ quorum-clock() (line 29): After 3 communication steps, p will have received, from a quorum of processes in its group, an ACK or BUMP message with a clock value larger or equal to final-ts(m).

4. $\forall m' : \langle \_, m', \_ \rangle \in \mathscr{T}, m' \notin \mathscr{D}, m' \neq m : \langle \text{final-ts}(m), m.id \rangle < \langle \text{min-ts}(m'), m'.id \rangle$ (line 30): If there are no concurrent conflicting messages, the clause is true. Otherwise, let t be the time of the a-multicast(m), let ts be the final timestamp of m, and let p be a process such that $p \in g$ and $g \in m.dest$. After 2 communication steps from t, the primary of g has received all ACKs from other primaries, and its clock must be at least as high as ts. Thus, any message m' with a final timestamp ts' smaller or equal to ts must have been a-multicast before that. Since it takes 3 communication steps from the a-multicast for the final timestamp to be known at a destination, p knows final-ts(m') $= ts'$ after at most 5 communication steps from t. Now, consider each message m'' such that $\langle \_, m'', ts'' \rangle \in \mathscr{T}$ at p after 5 communication steps from t:

   - if $ts'' > ts$, then min-ts(m'') is larger than ts and it can't falsify the clause
   - if $ts'' <= ts$, then final-ts(m'') must be known at p

   Thus, after 5 communication steps from the a-multicast of m, every message with a final timestamp smaller than or equal to m's, starting with the lowest, can be delivered and added to $\mathscr{D}$ at p.

Since clauses 1, 2 and 3 always are true after 3 communication steps, and clause 4 is true when there are no pending messages at p, the collision-free latency of PrimCast is 3 communication steps. Finally, since clause 4 is true after 5 communication steps, the failure-free latency of PrimCast is 5 communication steps. □

Proposition 3. *Uniform integrity: For any process p and any message m, p a-delivers m at most once, and only if $p \in m.dest$ and m was previously a-multicast.*

Proof. For a message to be delivered it must first be assigned a timestamp, and that only happens if a $\langle \text{START}, m \rangle$ is present in $\mathscr{M}$ (line 24), which means it must have been a-multicast. Furthermore, when a message is a-delivered, it is added to $\mathscr{D}$ (line 55), ensuring it is never deliverable again (line 26). □

Lemma 2. *Let p and q be two processes in group g, and let $\mathscr{T}_p$ and $\mathscr{T}_q$ be their respective $\mathscr{T}$ values. If $\mathscr{E}_{cur}$ is the same value at p and q, then one of $\mathscr{T}_p$ or $\mathscr{T}_q$ must be a prefix of the other.*

Proof. From the algorithm, processes can only update $\mathscr{E}_{cur}$ when r-delivering a NEW-STATE message (line 70). When they do so, both $\mathscr{E}_{cur}$ and $\mathscr{T}$ are set to the values in the NEW-STATE message. For any given epoch, a NEW-STATE message is r-multicast only once, by that epoch's leader. While $\mathscr{E}_{cur}$ doesn't change, the value of $\mathscr{T}$ can only be modified by appending tuples proposed by the leader of $\mathscr{E}_{cur}$ (lines 38 and 43). From the FIFO properties of r-multicast, it follows that processes with a given $\mathscr{E}_{cur}$ value append tuples to $\mathscr{T}$ in the same order. □

Lemma 3. *Let g be a group, $\mathscr{E}$ an epoch in g, and m a message. If $\exists quorum \in Q_g$ such that $\forall q \in quorum$ a tuple $\langle \text{ACK}, m, g, \mathscr{E}, ts, q\rangle$ is r-multicast by some process, then, at any other process in g where $state \in \langle \text{FOLLOWER}, \text{PRIMARY}\rangle$ and $\mathscr{E}_{cur} > \mathscr{E}$, $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$.*

Proof. There are only two possible cases where an ACK is r-multicast in the algorithm:

1. *quorum* ACK*s are r-multicast by processes with $\mathscr{E}_{cur} = \mathscr{E}$ (line 45):* We'll prove this case by induction.
   - *base case:* Let $\mathscr{E}'$ be the next epoch higher than $\mathscr{E}$ that gets a quorum of promises in g. The leader of $\mathscr{E}'$ must pick the longest value of $\mathscr{T}$ from the promises (line 65). From the algorithm, a process only sends an ACK when $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$. From Lemma 2 and quorum intersection, the chosen $\mathscr{T}$ must contain $\langle \mathscr{E}, m, ts\rangle$.
   - *induction step:* Let $\mathscr{E}'$ be an epoch higher than $\mathscr{E}$, such that, for every process in g with a value of $\mathscr{E}_{cur}$ in between $\mathscr{E}$ and $\mathscr{E}'$, $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$. If $\mathscr{E}'$ gets a quorum of promises in g, from quorum intersection, the most up-to-date promise will come from a process with $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$ (line 65).
2. *some* ACK *is r-multicast by processes with $\mathscr{E}_{cur} > \mathscr{E}$ (line 80):* Let $\mathscr{E}'$ be the smallest epoch such an ACK was r-multicast by a process with $\mathscr{E}_{cur} = \mathscr{E}'$. From the algorithm, the ACK can only be sent if $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$ (line 80). From line 76, a quorum of ACCEPTs must have been seen for $\mathscr{E}'$. Thus, from the same argument in case 1, if a process in g has $\mathscr{E}_{cur} \geq \mathscr{E}'$ then $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$. Now, assume for a contradiction that the same doesn't hold for some $\mathscr{E}''$ in between $\mathscr{E}$ and $\mathscr{E}'$. If a process has $state \in \langle \text{FOLLOWER}, \text{PRIMARY}\rangle$ and $\mathscr{E}_{cur} = \mathscr{E}''$ then a quorum of processes also moved to $\mathscr{E}_{cur} = \mathscr{E}''$ (line 76). From quorum intersection, the next epoch higher than $\mathscr{E}''$ that gets a quorum will chose a $\mathscr{T}$ that does not contain $\langle \mathscr{E}, m, ts\rangle$. A tuple of $\langle \mathscr{E}, m, ts\rangle$ can only be appended to $\mathscr{T}$ if $\mathscr{E}_{cur} = \mathscr{E}$. Thus, from the same argument in case 1, at a process with $\mathscr{E}_{cur} > \mathscr{E}''$, $\mathscr{T}$ will not contain $\langle \mathscr{E}, m, ts\rangle$. Since $\mathscr{E}' > \mathscr{E}''$ and a process with $\mathscr{E}_{cur} = \mathscr{E}'$ has $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$ (an ACK was sent), we get a contradiction.

□

Lemma 4. *Let p be a process, g a group and m a message. If local-ts$(m, g) = ts$ at p and $ts \neq \bot$:*

1. *Perpetually, local-ts$(m, g) = ts$.*

2. *At every process, local-ts$(m, g) \in \{\bot, ts\}$.*

3. *Eventually, at every correct process in $\bigcup m.dest$, local-ts$(m, g) = ts$.*

Proof. From its definition, if local-ts$(m, g) = ts$ at p, there is a quorum of acks for m from the same epoch $\mathscr{E}$ in $\mathscr{M}$. From the algorithm, the leader of $\mathscr{E}$ will propose m only once. From Lemma 3, the leaders of epochs higher than $\mathscr{E}$ must have $\langle \mathscr{E}, m, ts\rangle$ in $\mathscr{T}$ and can't propose m (line 24). By the same argument, there can't be a quorum for m in some epoch smaller than $\mathscr{E}$, otherwise m would not be proposed by the leader of $\mathscr{E}$. Thus, the value of local-ts(m, g) at any process can only be either $\bot$ or ts. Since no tuple is ever removed from $\mathscr{M}$, once local-ts$(m, g) = ts$ holds at some process, it holds forever. From Lemma 3, $\langle \mathscr{E}, m, ts\rangle \in \mathscr{T}$ at every primary or follower of the stable epoch. From line 80, if a process did not r-multicast an ack for m in previous epochs, it will do so at $\mathscr{E}_l$. Thus, eventually, a quorum of correct processes must r-multicast an ack for m, and from the validity property of r-multicast, every correct process eventually r-delivers a quorum of acks and has local-ts$(m, g) = ts$ □

Lemma 5. *Let p be a process, g a group and m a message. If final-ts$(m, g) = ts$ at p and $ts \neq \bot$:*

1. *Perpetually, final-ts$(m, g) = ts$.*

2. *At every process, final-ts$(m, g) \in \{\bot, ts\}$.*

3. *Eventually, at every correct process in $\bigcup m.dest$, final-ts$(m, g) = ts$.*

Proof. Follows directly from the definition of final-ts (line 12) and Lemma 4. □

Lemma 6. *Let p be a process that eventually a-delivers a message m, and let ts be m's final timestamp. At any time, at p, min-ts$(m) \leq ts$.*

Proof. Let's consider each line from the definition of min-ts (line 19). For line 20, from the definition of final-ts(m), for any g, local-ts$(m, g) \leq$ final-ts(m). Thus, we must show that at least one of the values from lines 21, 22 and 23 is smaller than or equal to ts. Since p a-delivers m, $g \in m.dest$. Let local-ts$(m, g) = ts'$ and let $\mathscr{E}$ be the epoch in g of the quorum of ACKs for m. There are then three cases to consider:

- $\mathscr{E}_{cur} < \mathscr{E}$: From the definition of quorum-clock() and line 68, since the proposal for ts' comes at a later epoch, and $ts' \leq ts$, quorum-clock() must be smaller or equal to ts (line 23).
- $\mathscr{E}_{cur} > \mathscr{E}$: From Lemma 3, $\langle \mathscr{E}, m, ts' \rangle \in \mathscr{T}$, and $ts' \leq ts$ (line 21).
- $\mathscr{E}_{cur} = \mathscr{E}$: If $\langle \mathscr{E}, m, ts' \rangle \in \mathscr{T}$, then the value from line 21 is smaller than ts. Otherwise, from the FIFO properties of r-multicast, since we haven't yet received the proposal for ts' from leader($\mathscr{E}$), min-clock($leader(\mathscr{E})$) $< ts'$ (line 22).

□

Lemma 7. *Timestamp order delivery: let m and m' be two messages and ts and ts' their respective final timestamps. If a process p a-delivers m before a-delivering m', then $\langle ts, m.id \rangle < \langle ts', m'.id \rangle$.*

Proof. Assume for a contradiction that m is a-delivered before m' by p, but instead $\langle ts', m'.id \rangle < \langle ts, m.id \rangle$. From Lemma 6 and our assumption, min-ts(m') $\leq ts' \leq ts$. Thus, at the time m is delivered, the comparison at line 30 would be false and prevent m from being delivered, a contradiction. □

Lemma 8. *Let m be a message and p a correct process. If $\langle \text{START}, m \rangle \in \mathscr{M}$ at p then, eventually, final-ts(m) $\neq \perp$ at every correct process in $\bigcup m.dest$.*

Proof. Let g be any of the groups in $m.dest$, and let l be the leader of the stable epoch $\mathscr{E}_l$ in g. When l eventually becomes the primary, there are two cases to consider:

1. $\exists \mathscr{E}, ts : \langle \mathscr{E}, m, ts \rangle \in \mathscr{T}$: From the algorithm and the definition of the stable epoch, a quorum of correct processes will eventually get to $\mathscr{E}_l$ and send an ack for $\mathscr{E}, m, ts$ (line 80). Thus, every correct destination will eventually r-deliver a quorum of ACK's and have local-ts(m, g) $= ts$
2. $\langle \_, m, \_ \rangle \notin \mathscr{T}$: Since both p and l are correct, from the agreement property of r-multicast, l will eventually r-deliver $\langle \text{START}, m \rangle$ and add it to $\mathscr{M}$. Thus, l will eventually propose m by r-multicasting an ACK to $m.dest$ (line 39). Since l is correct, every correct destination eventually r-delivers the ACK. From the FIFO properties of r-multicast, any process in g that r-delivers the ACK will have $\mathscr{E}_{cur} = \mathscr{E}_l$, since it must r-deliver the NEW-STATE coming from l first, and will then r-multicast their own ACK for m (line 45). Thus, a quorum of correct processes will eventually r-multicast an ACK for m in $\mathscr{E}_l$, and every correct destination will eventually r-deliver a quorum of these ACKs and have local-ts(m, g) $= ts$.

Thus, eventually, at every correct process in $\bigcup m.dest$, local-ts is defined ($\neq \perp$) for every group in $m.dest$, ensuring final-ts(m) $\neq \perp$. □

Lemma 9. *Let p be a correct process in group g and m a message. If final-ts(m) $\neq \perp$ at p, then p eventually a-delivers m.*

Proof. Assume p reaches the stable epoch ($\mathscr{E}_{cur} = \mathscr{E}_l$) without having a-delivered m yet. Let q be any correct process in g. We'll first show that eventually min-clock(q) at p is greater than or equal to final-ts(m). From Lemma 8, q eventually also has final-ts(m) $\neq \perp$. If the largest local-ts for m comes from a group other than g, q will r-multicast the respective BUMP if needed when r-delivering an ACK for m from that group. If the largest local-ts for m comes from g, since q is correct, it will either r-multicast an ACK for m when reaching the stable epoch $\mathscr{E}_l$ (line 80) or will r-multicast its ACK when receiving the proposal from the leader of $\mathscr{E}_l$. In any case, eventually, min-clock(q) $\geq$ final-ts(m) at p, for every correct process $q \in g$. Now, consider each of the conditions necessary for m to be delivered (from the definition of deliverable(m) 26):

1. final-ts(m) $\leq$ min-clock($leader(\mathscr{E}_{cur})$): From the above, it eventually holds.

2. final-ts(m) $\leq$ quorum-clock(): Since a quorum of processes in g are correct, from the above and the definition of quorum-clock(), it will eventually hold.

3. $\forall m' \notin \mathscr{D}, m' \neq m : \langle$final-ts($m$)$, m.id\rangle < \langle$min-ts($m'$)$, m'.id\rangle$: From the algorithm (line 45), since a quorum of processes are correct and reach the stable epoch, every tuple in $\mathscr{T}$ from epochs smaller than $\mathscr{E}_l$ will eventually be acknowledged by a quorum of correct processes. Thus, from line 47 and Lemma 8, all these messages in $\mathscr{T}$ from previous epochs eventually get a final-ts. Since the leader of $\mathscr{E}_l$ is correct, any messages it proposes will also eventually get a final-ts (Lemma 8). Once final-ts(m) $\leq$ min-clock($leader(\mathscr{E}_{cur})$), no new messages can be proposed in g with a smaller timestamp than final-ts(m). Thus, eventually, every message present in $\mathscr{T}$ with a smaller final-ts than final-ts(m) will be known to p.

From the above, eventually, every message with timestamp up to and including final-ts(m) can be a-delivered by p, in final-ts order (with message id breaking ties). □

Proposition 4. *Uniform agreement: If a process p a-delivers m, then eventually all correct processes $q \in \bigcup m.dest$ a-deliver m.*

Proof. Since p a-delivers m, it must have final-ts(m) $\neq \perp$. From Lemma 5, every correct process eventually also does, and from Lemma 9 eventually a-delivers m. □

Proposition 5. *Uniform prefix order: Let m and m' be messages and p and q processes such that $\{p,q\} \subseteq \bigcup(m.dest \cap m'.dest)$. If p a-delivers m and q a-delivers m', then either p a-delivers m' before m or q a-delivers m before m'.*

Proof. From Lemma 7, messages are delivered in timestamp and message id order. Assume for a contradiction that $\langle$final-ts(m)$, m.id\rangle < \langle$final-ts(m')$, m'.id\rangle$, but q delivers m' without first delivery m. Let g be q's group. Since p a-delivers m, from Lemma 4, q must eventually have local-ts(m, g) $\neq \perp$. Now, consider the time at which m' is delivered at process q:

1. If m is present in $\mathscr{T}$: from Lemma 6, min-ts(m) $\leq$ final-ts(m), and from line 30 m' would not be deliverable (a contradiction).

2. If m is not present in $\mathscr{T}$: The proposal for local-ts(m, g) can't come from a previous epoch than $\mathscr{E}_{cur}$, otherwise from Lemma 3 it would be present in $\mathscr{T}$. Since min-clock($leader(\mathscr{E}_{cur})$) $\geq$ final-ts(m') $\geq$ local-ts(m, g), the proposal for local-ts(m, g) also can't come from the leader of $\mathscr{E}_{cur}$. Finally, since quorum-clock() $\geq$ final-ts(m'), the proposal for local-ts(m, g) also can't come from a higher epoch than $\mathscr{E}_{cur}$. From Lemma 4, q must see the proposal for m, causing a contradiction.

□

Proposition 6. *Global total order: Let $\prec$ be a relation on the set of messages that processes a-deliver such that $m \prec m'$ iff some process a-delivers m before m'. The $\prec$ relation is acyclic.*

Proof. Messages are delivered in timestamp and message id order (Lemma 7). □

Proposition 7. *Validity (liveness): If a correct process p a-multicasts m, then eventually all correct processes $q \in \bigcup m.dest$ a-deliver m.*

Proof. Since p is correct, every correct process eventually has $\langle start, m\rangle \in \mathscr{M}$, and from Lemmas 8 and 9, m eventually gets a-delivered. □

4.4 PrimCast extensions

In the following, we describe a few modifications to PrimCast that may benefit practical applications of the protocol.

4.4.1 Exploiting loosely synchronized clocks

Many datacenters today provide loosely synchronized clocks through the use of satellite and atomic clocks [25, 96]. When synchronized clocks are available, we propose the following modification to PrimCast, inspired by hybrid logical clocks [55]. Assuming that real-clock() returns the server's hardware clock value, we modify line 37 in the following way:

$clock \leftarrow \max(clock + 1, \text{real-clock()})$

Assume a period of system stability (as defined in Section 4.1.2) where Δ is the communication step latency, and that clocks are synchronized with a maximum skew of ϵ from real time (i.e., 2ϵ from each other). By having the primary update its clock before proposing a message's local timestamp, the failure-free delivery latency changes from 5Δ to $\min(5\Delta, 4\Delta + 2\epsilon)$. The argument is as follows. Let m be a message delivered at process $p \in g$ with final timestamp ts, and let t be the time at which m is a-multicast. The time it takes for m to arrive at any primary in $m.dest$ from t is Δ, thus the maximum timestamp possibly assigned to m is $t + \Delta + \epsilon$. Let m' be any message such that $m.dest \cap m'.dest \neq \emptyset$. The latest time at which m' can be a-multicast and still be ordered before m by some primary in $m.dest$ is the minimum of:

- $t + 2\Delta$: the time for primaries in $m.dest$ to exchange timestamp proposals for m and update their clocks to ts.

- $t + \Delta + 2\epsilon$: the time for m to reach primaries is $t + \Delta$, and 2ϵ comes from the worst case of ts being assigned a value ϵ in the future and ts' a value ϵ in the past.

Let p be some process in $m.dest \cap m'.dest$. Since the collision-free latency of PrimCast is 3Δ, the final timestamp of m' is known at p, at the latest, by time $t + min(2\Delta, \Delta + 2\epsilon) + 3\Delta$, allowing for m to be delivered. Assuming 2ϵ is smaller than Δ, this effectively reduces the worst case convoy effect by the difference between the two. We expect this technique to be particularly effective in a geographically distributed deployment where, considering the values reported in [25], 2ϵ can be roughly an order of magnitude smaller than Δ. We note that this modification does not impact the correctness of the algorithm, and also cannot increase the worst case convoy effect, even if clocks are not synchronized.

4.4.2 Timestamped atomic multicast

In the classic atomic multicast interface (Section 2.2.1), message timestamps are hidden from the application. We propose to extend atomic multicast by (1) providing a message's final timestamp upon message delivery, (2) allowing for senders to propose a lower bound timestamp when a-multicasting a message, and (3) exposing a *safe timestamp* that is ensured to be lower than any new deliveries. More precisely, the timestamped atomic multicast abstraction provides the following three primitives:

- A-MULTICAST(m, ts): message m is a-multicast to $m.dest$ and must be assigned a final timestamp larger than ts.
- A-DELIVER(m, ts): signals the delivery of m with final timestamp ts.
- SAFE-TS(): timestamp value for which all messages with a lower timestamp have been a-delivered.

The updated A-MULTICAST and A-DELIVER primitives can be trivially implemented in PrimCast by having primaries update their clock if needed in line 37, and by exposing final-ts(m) upon the delivery of m. The SAFE-TS() primitive can be implemented in the following way:

```
safe-ts() ≡
    min(min-clock($leader(\mathscr{E}_{cur})$),          ▷ min new proposal in current epoch
        quorum-clock(),                                   ▷ min new proposal in next epoch
        min($ts \mid \langle \_, m, ts \rangle \in \mathscr{T}$ and $m \notin \mathscr{D}$))   ▷ min proposed and not delivered
```

These modifications allow for clients to capture causal dependencies by including their highest seen timestamp in each application request. Causal reads could be fulfilled by a single replica by simply waiting for SAFE-TS() to be equal to or larger than the timestamp provided by the client, once all earlier updates have been applied. Another benefit of exposing timestamps is that of identifying global snapshots [71]. Each timestamp ts identifies a single consistent global snapshot, composed of the state at each group after applying every request with a timestamp smaller than or equal to ts. A server can check SAFE-TS() to know up to which timestamp it has delivered all requests. This technique could be particularly useful in practice when combined with real-time based timestamps (Section 4.4.1), allowing for meaningful snapshot identifiers.

4.4.3 Exploiting commutativity

In some applications, there are pairs of requests that do not conflict with each other, that is, their relative execution order does not affect their result. This is the case of two read-only requests, for example. When the application can provide this pairwise CONFLICTS predicate, the ordering property of atomic multicast can be relaxed in the following way [4]:

- *Generalized prefix order:* Let m and m' be messages and p and q processes such that $\{p, q\} \subseteq \bigcup(m.dest \cap m'.dest)$. If p executes A-DELIVER(m) and q executes A-DELIVER(m'), and CONFLICTS(m, m') is true, then either p executes A-DELIVER(m') before A-DELIVER(m) or q executes A-DELIVER(m) before A-DELIVER(m').

To account for the commutativity of requests, we can modify PrimCast's deliverable predicate (line 26) in the following way (addition in gray):

```
deliverable($m$) $\equiv$
    $m \notin \mathcal{D}$ and final-ts($m$) $\neq \perp$ and
    final-ts($m$) $\leq$ min-clock($leader(\mathcal{E}_{cur})$) and
    final-ts($m$) $\leq$ quorum-clock() and
    $\forall m' : \langle \_, m', \_ \rangle \in \mathcal{T}, m' \notin \mathcal{D}, m' \neq m$, conflicts($m, m'$) :
        $\langle$final-ts($m$)$, m.id\rangle < \langle$min-ts($m'$)$, m'.id\rangle$
```

In practice, this modification prevents two requests that commute from blocking each other's delivery, eliminating the convoy effect in such cases.

4.5 Performance evaluation

In this section, we begin by presenting the different protocol implementations evaluated in our experiments. We then describe the different experiment scenarios, and for each scenario we present our results and discuss our findings.

4.5.1 Implementation

We implemented a prototype of PrimCast in Rust using Tokio [107], an asynchronous runtime for building network applications.[2] Our implementation is not specifically designed for multi-thread execution, but Tokio's executor can exploit the parallelism and we run PrimCast with 2 threads. In our experiments,

[2] In the final version of this thesis we will make the code available as open source. We have not done it yet due to a concurrent double-blind paper submission.

Scenario	Cross-group RTT (between leaders)	Intra-group RTT	Description
LAN	0.09ms	0.09ms	8 groups deployed inside a cluster.
WAN - colocated leaders	0.09ms	60ms, 76ms, 130ms	3 regions, 8 groups. Each group deployed across all regions.
WAN - distributed leaders	90ms	30ms	8 regions with 3 datacenters each. Each of 8 groups deployed in its own region.

Table 4.1. Deployment scenarios

we compare PrimCast against two state-of-the-art atomic multicast protocols, FastCast and White-Box (see Section 4.6 for details). For both FastCast and White-Box, we use the open-source implementations provided by the authors in [35] and [113] respectively, both implemented in C using libevent [63]. We use in-memory storage for all implementations. We also run PrimCast using the hybrid clock approach described in Section 4.4.1 (PrimCast HC in the figures).

4.5.2 Setup and scenarios

We run all experiments in a cluster, each machine consisting of an eight-core Intel Xeon L5420 2.5GHz processor, 8GB of memory, and 1Gbps ethernet card. The RTT (round-trip time) inside the cluster is around 0.09ms. Besides the deployment inside a LAN, we consider two different emulated wide-area network (WAN) scenarios. To emulate WAN latencies, we used Linux traffic control tools. In all scenarios we deploy 8 groups, each group consisting of 3 replicas. Table 4.1 summarizes the deployment scenarios.

We collocate one client with each replica in the system. For each message, a client chooses the destination groups at random, except for the group of the replica it is connected to, which is always included. To increase the load in the system we uniformly increase the number of outstanding messages from each client. Latency is measured at the client as the time from the message being sent to it being delivered and returned to the client by its replica. We report latency values gathered from all clients in the system.

4.5.3 LAN performance

Figure 4.2 compares the performance of the four protocols in a cluster deployment, as load increases, with every message multicast to 2 destinations. Our results show that both versions of PrimCast have better performance than both FastCast and White-Box, at every load level measured. FastCast reaches saturation earlier, as it needs to run a slow and a fast path in parallel for message

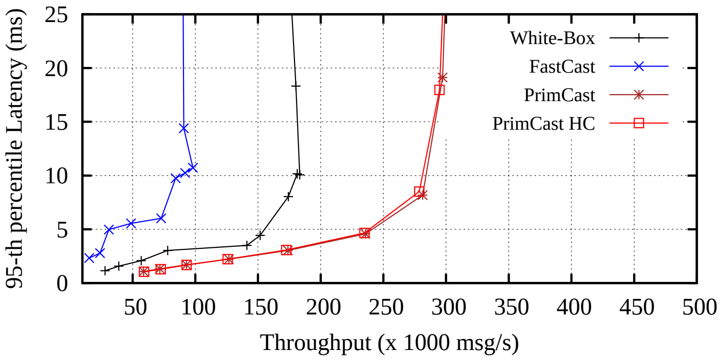


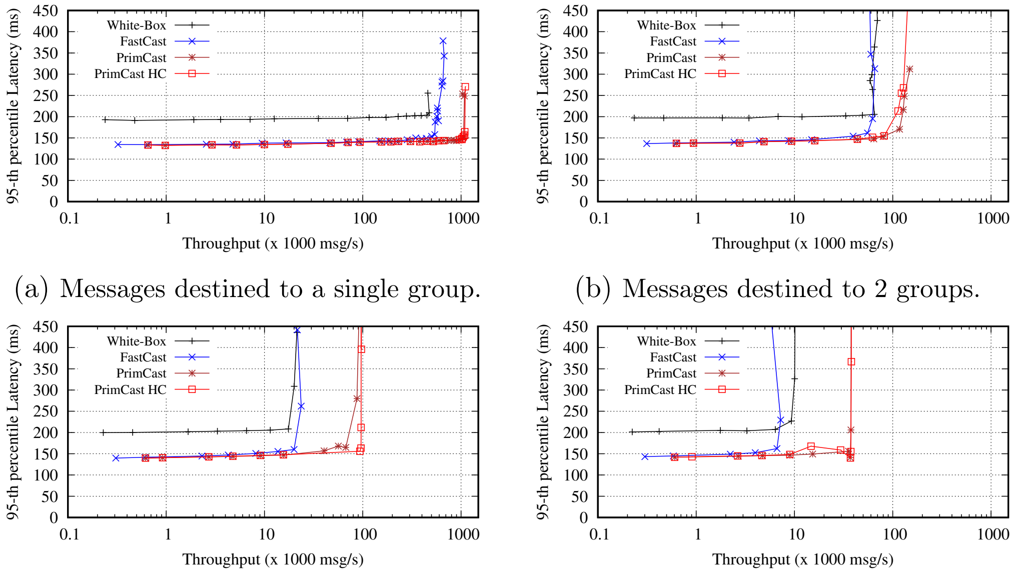
Figure 4.2. Throughput and 95th-percentile latency in a LAN, with all messages multicast to two groups.

delivery (Section 4.6.1). When compared to White-Box, even though PrimCast needs extra information to be exchanged between groups and replicas, most of it consists of small acknowledgment information that can, with careful design, be piggybacked on or inferred from other protocol messages, allowing for an efficient implementation, as our results show. We also note that the hybrid clock approach does not have any particular impact on performance when leaders are colocated, as the convoy effect is mostly a function of cross-group latency. Even tough none of the protocols were designed with a LAN deployment in mind, these results show that PrimCast can be a reasonable alternative in a LAN.

4.5.4 WAN performance with colocated leaders

This scenario evaluates the performance of the protocols in a WAN deployment when group leaders are colocated in the same datacenter. We emulate 3 geographic regions, each with one datacenter, and deploy one replica from each group in each datacenter. We use the latency values reported in [43], the RTT between region pairs being 60ms, 76ms and 130ms, with a standard deviation of 5%.

Figure 4.3 shows how the three protocols behave under increasing load, with messages multicast to 1, 2, 4 or 8 destination groups. Both PrimCast and FastCast exhibit the same latency behavior until close to saturation. In the common setup of 3 replicas per group, FastCast can also quickly deliver messages at non-leader replicas. Even then, PrimCast can deliver from 1.6x (1 destination) to 5x (2 destinations) the throughput of FastCast. While some of this difference can be be accounted for by the use of 2 threads in PrimCast's asynchronous execution library, FastCast performance degrades faster with increasing destinations due



(a) Messages destined to a single group.

(b) Messages destined to 2 groups.

(c) Messages destined to 4 groups.

(d) Messages destined to all 8 groups.

Figure 4.3. Throughput and 95th latency in a WAN with no cross-group latency (i.e., collocated leaders).

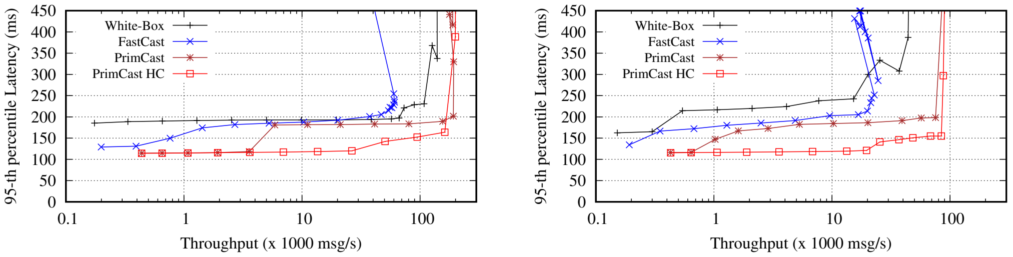
to the fast and slow paths that need to be executed by the protocol. White-Box on the other hand, needs one extra communication step from leaders, where a message is initially delivered, to the other replicas. This extra latency shows in the 95th percentile latency over the whole system. Similarly to the results in a LAN, the convoy effect in this deployment is almost non-existent, as it is a function of cross-group latency. Hence, using hybrid clocks has no effect on latency.

4.5.5 WAN performance with distributed leaders

Under high load, timestamp based atomic multicast protocol can exhibit a convoy effect [4], where a message needs to wait for other messages, with smaller final timestamps, to be delivered first. This extra latency induced by the convoy effect is mostly a function of the latency between replica groups, being almost imperceptible in a deployment with no latency between groups, as seen in Section 4.5.4. To evaluate the convoy effect in the different protocols, we emulate a WAN deployment distributed leaders. Each of the 8 groups is deployed to its own geographic region, the RTT being 90ms between regions and 30ms inside a region, with a standard deviation of 5%.

Figure 4.4 shows the behavior of the three protocols with messages multicast to 2 and 4 destination groups, as load increases. Differently from the previous deployment, the convoy effect is now clearly visible. From the latency curves, it can be seen that the convoy effect kicks in at different load levels in each protocol, but all are affected by it. As with the previous deployment, White-Box shows worse 95th latency due to extra delay needed to deliver messages at non-leader replicas. Furthermore, in this deployment PrimCast is able to deliver messages at every destination earlier than any of the two other protocols: one intra-group communication step earlier, around 15ms in this deployment. More interestingly though, in this deployment, using hybrid clocks greatly reduces and delays the onset of the convoy effect.

Figure 4.5 shows the latency distribution for all clients in the system, for each protocol, at two different system loads: one with low load and thus low convoy (left), and one with high load (right). Figure 4.5a demonstrates how PrimCast consistently delivers lower latencies at every replica in the system. Figure 4.5b, on the other hand, clearly shows how the convoy effect impacts most messages in the system once it takes effect, and also shows how using hybrid clocks can almost eliminate the effects of convoy in this particular workload.



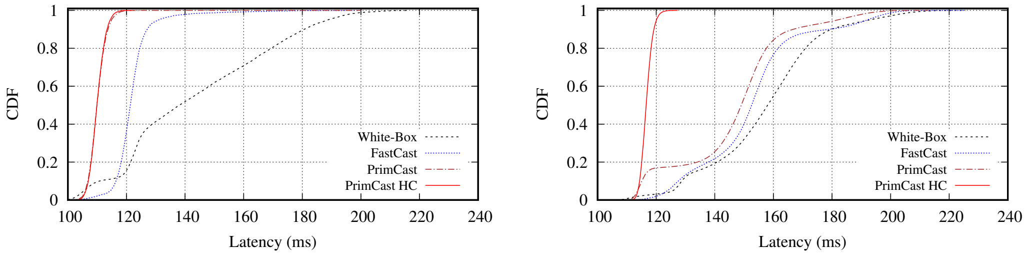
(a) Messages destined to 2 groups.

(b) Messages destined to 4 groups.

Figure 4.4. Throughput and 95th latency in a WAN with high cross-group latency.

4.6 Related work

Most proposals for genuine atomic multicast are derived from the timestamping scheme of Skeen's protocol [15]. In this section, we give a detailed account of FastCast and White-Box atomic multicast protocols as they are the most relevant to PrimCast. We conclude with an overview of other related proposals.



(a) 2 destination groups, 2 outstanding msgs per client (low convoy)

(b) 2 destination groups, 128 outstanding msgs per client (high convoy)

Figure 4.5. Latency CDFs at two different load levels, corresponding to the 2nd and 8th points from the curves in Figure 4.4a

4.6.1 FastCast

In [24], Coelho et al. propose FastCast, a genuine atomic multicast protocol that has collision-free and failure-free latency values of 4 and 8 communication steps, respectively. FastCast achieves faster delivery times than the classic protocols through an optimistic execution path that works as follows:

1. Each group has an elected leader, responsible for proposing local timestamps at its group.
2. Once a message m needs to be timestamped, a leader sends its proposal to the leaders of every other group in $m.dest$, before proposing the timestamp through consensus.
3. Once a leader gets the proposals from all leaders in $m.dest$, it sends the maximum of all proposals as an optimistic final timestamp through its group's consensus.

If the final timestamp matches the optimistic timestamp, the message can be delivered before the second sequential round of consensus. The optimistic path can be understood as a mechanism for updating a group's logical clock before the final timestamp is decided. In FastCast, both the commit latency and the clock update latency are equal to four communication steps. Hence, collision-free and failure-free latency values are four and eight communication steps respectively.

4.6.2 White-Box multicast

In [43], Gotsman et al. propose White-Box, an atomic multicast protocol that improves the collision-free and failure-free latency values to four and six communication steps, respectively. Furthermore, at group leaders, delivery happens one step earlier, in three and five communication steps. Differently from previous approaches, White-Box does not use a consensus protocol as a black-box, opting instead for an integrated protocol at group and global level. Another insight from White-Box is the use of a *primary-based* approach (i.e., passive replication) [52, 64]. Instead of having every replica execute the same sequence of operations after those are ordered (i.e., state-machine replication), the primary decides on the order of operations and then ensures the other replicas follow that same order.

The White-Box protocol works roughly as follows:

1. A process a-multicasts m by sending a message to the primaries of each group in $m.dest$.

2. Each primary then picks a local timestamp for m for its group, based on its own clock value. It then sends that proposal to every process in every group in $m.dest$, as an ACCEPT message.

3. Once a process receives the ACCEPT from every primary in $m.dest$, it will store the proposal for its group and update its clock to the highest local timestamp received, if needed. It then sends back an ACK to each of the primaries.

4. Once a primary receives all the ACCEPTs and a quorum of ACKs from each group in $m.dest$, it will pick the final timestamp for m as the largest local timestamp. The primary then carefully tracks pending messages to a-deliver messages in final timestamp order. At a primary, the a-delivery of a message m can happen as early as three communication steps after it is a-multicast. Once the primary a-delivers a message, it sends a DELIVER message to its group.

5. Followers a-deliver messages in the order of the DELIVER messages sent by the primary, in as early as four communication steps from the respective a-multicast.

From the above, we get to the collision-free latency of four communication steps, or three communication steps when only considering the delivery at group primaries. By having local timestamps be proposed only by the primary of a group, the clock update latency (Section 4.1.2) of the protocol is shortened, when compared to previous timestamp-based protocols. Once a message m is a-multicast, after two communication steps (enough for primaries in $m.dest$ to exchange local timestamp proposals and update their clocks), no new conflicting message can be assigned a local timestamp smaller than the final timestamp of m, as long as primaries remain stable. Hence, the failure-free latency of White-Box is six communication steps, or five when considering delivery at primaries only.

4.6.3 Other protocols

The use of multiple instances of a consensus protocol, one per group, to solve atomic multicast when processes may fail was proposed in [40] and [46]. These protocols can deliver a message in six communication steps in the collision-free case, that is, when there are no concurrent messages. In the presence of concurrency, atomic multicast protocols may suffer from a convoy effect [4]. These two protocols have a failure-free delivery latency (i.e., latency in the presence of concurrency) of twelve communication steps.

In [90], Schiper et al. propose improvements to [40] that reduce the collision-free delivery latency at some of the destination groups, those that assign a local timestamp equal to the final timestamp of the message, to four communication delays.

In MTO [45] and Scalatom [84], instead of running multiple consensus instances per message (one per group), a single instance of consensus is run among all destination groups. These protocols achieve a collision-free and failure-free latency values of five and nine, respectively.

Tempo [34] is a partitioned state-machine replication protocol that is built over a protocol that is essentially a genuine atomic multicast implementation. Instead of relying on a primary at each destination group, each message has a designated leader in each group (the closest replica) that communicates with other processes in the group to assign the message a local timestamp. To decide when messages are safe for execution, Tempo uses a notion of timestamp stability that works in parallel with the timestamping of messages, similar to how PrimCast exchanges bump messages to update quorum-clock() values. Even though there are similarities between the two approaches, PrimCast and Tempo have been developed in parallel.

Many non-genuine solutions to atomic multicast have also been proposed in the literature. In [91], Schiper et al. propose a round based protocol which can deliver messages in four communication steps. An unbounded sequence of rounds is executed, and each group chooses a set of messages to be delivered at each round. Proposals in each round are exchanged and then deterministically ordered and delivered by the destinations. ByzCast [23] is a byzantine fault-tolerant atomic multicast that organizes process groups in a tree. Each message is first ordered by the lowest common ancestor of its destinations, and then proceeds down the tree being ordered by each group until it is ordered at each destination. Partial order is ensured by having each group respect the ordering of its ancestors. In Multi-Ring Paxos [68], Ridge [14] and Elastic Paxos [10], processes subscribe to the groups they are interested in receiving messages from, and then a deterministic merge procedure is used to ensure a partial ordering of messages. These protocols have a slightly different interface: a message is addressed to a single group, but groups do not have to be disjoint: sending a message to multiple groups is possible by having a group that is a superset of those destination groups. We refer to the survey by Défago et al. [30] for an overview of total and partial order communication abstractions.

PrimCast relies on a primary-based approach (i.e., passive replication) for deciding on timestamps inside each group [52, 64]. Since the primary orders all operations in a group, it can optimistically update its state before having the rest of the group agree on it. This property can be exploited for faster logical clock updates inside groups, reducing the impact of the convoy effect. We refer to [109] for a discussion of the differences between state-machine replication (i.e., active replication) and primary-based replication.

4.7 Discussion

This chapter presents PrimCast, a genuine atomic multicast protocol that can deliver messages, from sender to any destination process, in three communication steps. In the presence of conflicting messages, delivery happens at every destination in at most five communication steps. This is an improvement over previous work, which needed at least four (or six in the presence of concurrency) communication steps for delivery at some of the destinations. PrimCast achieves a lower latency through the usage of a primary-based replication mechanism and a novel way of tracking logical clocks through simple quorum intersection. A formal proof of correctness is provided for the properties of the algorithm. We further show how to extend the protocol to (1) lower the convoy effect using loosely synchronized clocks, (2) expose timestamps to the application to track causal dependencies and (3) exploit the commutativity of requests. Our experimental evaluation of PrimCast shows that it consistently delivers lower latency than the alternatives while still providing higher throughput. The results also show that, in some cases, using loosely synchronized clocks can almost eliminate the effects of convoy on delivery latency.

Chapter 5

Linearizable Atomic Multicast

Partitioned state machine replication is a technique that extends classical state machine replication with state partitioning (or sharding) to provide both fault tolerance and performance scalability. The crux of the technique is ordering client requests within a partition, among the replicas that implement the partition, and across partitions, involving all the replicas accessed by the request. To cope with the complexity of ordering requests, partitioned state machine replication can use atomic multicast, a communication abstraction. Atomic multicast provides the means for requests to be propagated reliably and consistently to one or more sets of groups of replicas, where each replica group implements one partition. In this chapter we revisit atomic multicast from the perspective of partitioned state machine replication. More specifically, we relate the notions of strong consistency, in the form of linearizability (Section 2.2.2), and atomic multicast in the context of partitioned state machine replication. This chapter makes the following contributions: First, we show that if one implements partitioned state machine replication using an atomic multicast with global total order, a strong order property, then replicas would need to further coordinate as part of the execution of requests to ensure correctness. Second, we introduce a stronger version of atomic multicast that accounts for real-time dependencies between requests. Our proposed atomic multicast can be used to order requests within and across partitions so that replicas do not need to further coordinate to ensure linearizability. Third, we extend a well-known implementation of atomic multicast to ensure the stronger order property.

The rest of this chapter is organized as follows. Section 5.1 provides the necessary background information needed to follow the rest of the chapter. In Section 5.2, we prove that partitioned state machine replication with an atomic multicast primitive that ensures global total order requires additional replica coordination to implement linearizable applications. We then propose an atomic multicast, based on atomic global order, and prove that when equipped with such an atomic multicast, replicas do not need this additional coordination. Section 5.3 shows that the well-known atomic multicast protocol proposed by Skeen does not ensure atomic global order and present modifications to the original protocol to guarantee the stronger property. Section 5.4 reviews related work. Section 5.5 concludes the chapter.

5.1 Background

State machine replication (SMR) and primary-backup replication, the two most fundamental techniques for fault tolerance [44], do not scale performance as replicas are added to the system. In state machine replication (Section 2.2.3), client requests are executed by all the replicas in the same order. As long as execution is deterministic, replicas will transition through the same state changes and produce the same results. Since each replica executes every request, additional replicas will not result in any performance improvements. On the contrary, a larger number of replicas may lead to a degradation in performance. This happens because replicas need to coordinate to totally order requests, and the more replicas involved, the more messages need to be exchanged, reducing the number of requests that can be ordered in the system [47]. In primary-backup replication (Section 2.2.4), the primary replica receives and executes all requests, and then sends state changes to the backup replicas. The backups simply apply the state changes. As in state machine replication, increasing the number of replicas may degrade performance as it increases communication between the primary and the backups.

5.1.1 Partitioned state machine replication

To improve the scalability of classic SMR, some approaches have proposed to partition the application state, a technique also known as sharding, and implement each partition with an instance of SMR (e.g., [25]). The idea is to divide the application state into partitions and store each partition in a different set of servers. Therefore, requests that access different partitions can be executed in parallel. If the partitioning is such that requests fall within one partition only (i.e., the objects read and written by the request belong to a single partition) and requests are evenly distributed among partitions, then we can scale performance with the number of partitions in the system. Moreover, since linearizability is a

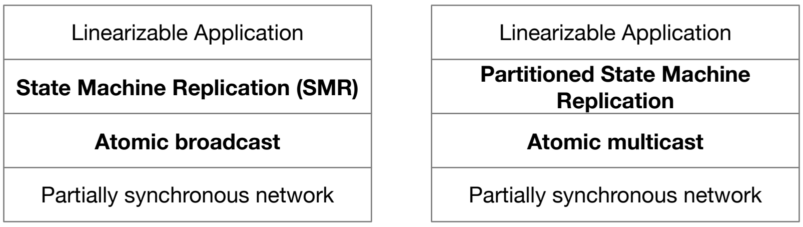

Linearizable Application
State Machine Replication (SMR)
Atomic broadcast
Partially synchronous network
Linearizable Application
Partitioned State Machine Replication
Atomic multicast
Partially synchronous network

Figure 5.1. State machine replication.

composable property [49], such a scheme will result in a linearizable application. If the application state cannot be perfectly partitioned, as described above, then one must account for requests that involve multiple partitions (i.e., a request that reads and writes objects that belong to more than one partition). There are two aspects concerning multi-partition requests: how to consistently order requests that span multiple partitions, and how to execute them.

Ordering requests within and across partitions is complex (e.g., it may involve solving multiple instances of consensus [21], one consensus instance per partition). One way to cope with this complexity is to encapsulate the ordering of requests in a group communication primitive: atomic multicast (Figure 5.1). Toueg and Hadzilacos [48] define three types of atomic multicast that differ by the strength of their guarantees. We consider the strongest type of atomic multicast, defined by the properties in Section 2.2.1. More precisely, we consider the uniform version of the atomic multicast properties defined in [48].

Executing requests that involve multiple partitions can also be challenging since partitions may lack the state needed to execute the request. For example, assume a request r that swaps the contents of state variables x and y, which reside in partitions P_x and P_y, respectively. Replicas in P_x (resp. P_y) need the value of y (resp. x) in order to update x (resp. y). In S-SMR [13], after delivering a request, replicas exchange the state needed to execute the request. In the example above, replicas in P_x (resp. P_y) send to replicas in P_y (resp. P_x) variable x (resp. y). After exchanging the needed variables, replicas in all involved partitions execute r. Updates on variables not part of a partition are kept by the partition momentarily, during the execution of the request, and then discarded. In DynaStar [59], all variables read and written by a multi-partition request are moved to one of the partitions involved in the request. Only replicas in this partition execute the request.

The details of how to execute multi-partition requests are orthogonal to our contributions in this work. Hence, for simplicity, we assume that multi-partition requests can be executed at each involved partition without exchange of data. A request finishes execution once the issuing client gets a reply from each involved partition. This simplified execution model does not allow a request to swap the contents of variables x and y if they reside in different partitions. But our model can still capture interesting applications. For illustrative purposes, hereafter, we consider a key-value store service that supports two types requests: inserts and range queries. Key-value pairs are partitioned in two partitions, P_0 being responsible for even-numbered keys and P_1 for odd-numbered keys. An insert $w(k, v)$ inserts the pair (k, v) and is a single-partition request, multicast only to the partition in charge of the key. A range query $r(k_{start}, k_{end})$ returns all previously inserted pairs for keys from k_{start} up to and including k_{end}. Range queries are multicast to both partitions, assuming ranges that span multiple keys.

5.2 Atomic Global Order

In this section, we argue that atomic multicast as the sole means of communication among replicas in partitioned state machine replication is not enough to ensure linearizability. We then extend the atomic multicast properties to achieve linearizable executions and prove their correctness.

5.2.1 Atomic multicast alone is not enough

State machine replication can be easily implemented with atomic broadcast [44]. It suffices for client requests to be atomically broadcast to all replicas, which execute the requests following the order in which the requests are delivered. One would expect that partitioned state machine replication could be implemented using a similar approach, namely, by atomically multicasting requests to all the groups involved in the request. Upon delivering the request, a replica executes the request and sends the results to the client. The request finishes at the client after the client receives a response from at least one server in each group involved in the request. As we show next, however, this simple execution model and the multicast properties as described in Section 2.2.1 are not enough to ensure linearizability.

Consider an execution of the key-value store service described in Section 5.1.1, involving partitions P_0 and P_1, as depicted in Figure 5.2 (top). The two commands $w(0, v_0)$ and $w(1, v_1)$ insert key-value pairs for keys 0 and 1, respectively. Moreover, $w(0, v_0)$ precedes $w(1, v_1)$ in real-time, that is, $w(0, v_0)$ finishes at client b before $w(1, v_1)$ starts at client c. Now consider a concurrent range query $r(0, 1)$ that tries to read previously inserted pairs for keys 0 and 1. The range query accesses both partitions and is delivered and ends at P_0 before $w(0, v_0)$ is delivered, and is delivered at P_1 after $w(1, v_1)$ ends. The prefix order property of atomic multicast is not violated since all processes that deliver the same messages, do it in the same order. The atomic multicast acyclic order property is not violated either: $w(0, v_0)$ and $w(1, v_1)$ are not directly related since they are delivered at different partitions, and thus, $r(0, 1) \prec w(0, v_0)$ at P_0 and $w(1, v_1) \prec r(0, 1)$ at P_1. Although the $\prec$ relation is acyclic, $w(1, v_1) \prec r(0, 1) \prec w(0, v_0)$, it does not account for the real time order in which $w(0, v_0)$ precedes $w(1, v_1)$. Even though $(0, v_0)$ is inserted before $(1, v_1)$ in the key-value store, the range query only returns the pair $(1, v_1)$, violating linearizability.

The problem above stems from the fact that atomic multicast properties do not capture real-time dependencies between requests. Consequently, a system that implements partitioned state machine replication with the atomic multicast properties described in Section 2.2.1 would need to introduce additional coordination across partitions to ensure linearizability. For example, in S-SMR[13], after multi-partition request $r(0, 1)$ is delivered and before it is executed by the replicas, partitions P_0 and P_1, involved in $r(0, 1)$, exchange “signal messages" to avoid the problem described above, as depicted in Figure 5.2 (bottom). Intuitively, if partitions P_0 and P_1 exchange messages during the execution of $r(0, 1)$, it is not possible for one partition to finish executing $r(0, 1)$ before the other starts. Thus, in between the execution of $r(0, 1)$ at the partitions involved, we cannot have other commands being executed at same partitions.

5.2.2 Linearizable atomic multicast

The discussion in the previous section shows that differently than atomic broadcast in state machine replication, atomic multicast is not sufficient as a communication abstraction to ensure linearizability in partitioned state machine replication without additional coordination among replicas. We now strengthen atomic multicast so that replicas can execute the request after delivering it without further coordination.

Our strategy is to enlarge the scope of the ordering guarantees of atomic multicast. We achieve this by replacing the global total order property of atomic multicast with the following property, which accounts for the real time relation of messages.

- *Atomic global order*: Define relation $\prec$ on the set of messages processes deliver as follows: $m \prec m'$ iff (i) there exists a process that delivers m

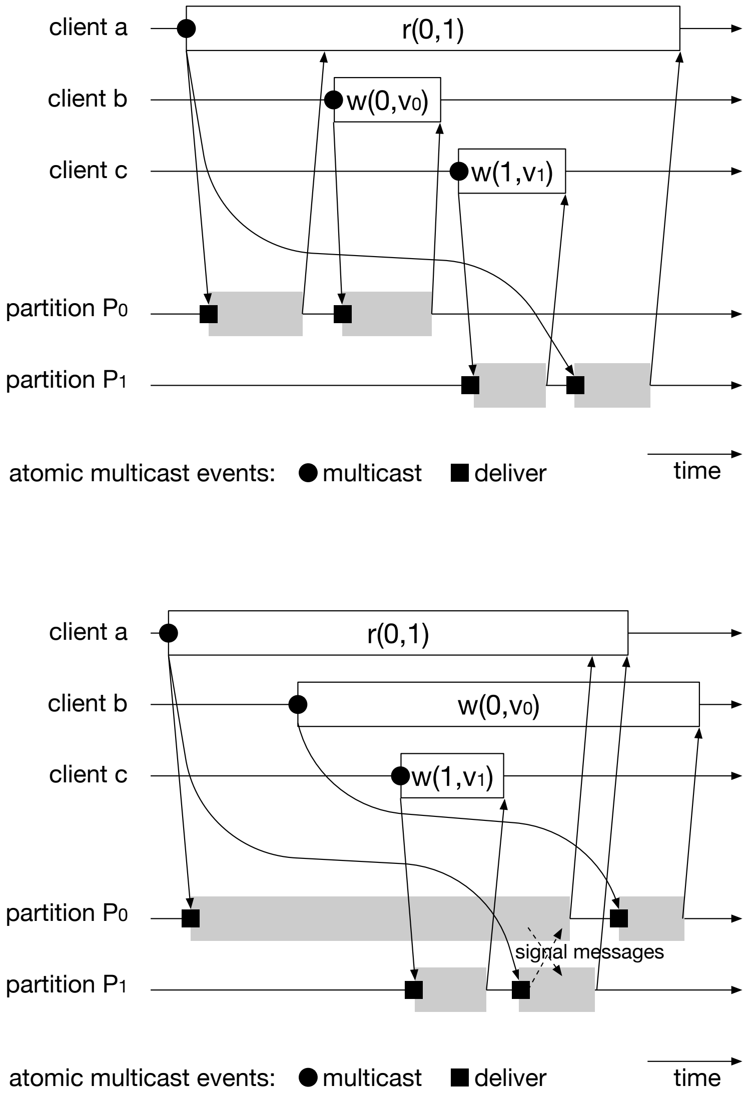


Figure 5.2. An execution that violates linearizability (top) and a linearizable execution from S-SMR [13] (bottom). For simplicity, we assume that each partition contains a single replica.

before m'; or (ii) m' is multicast after m is delivered at some destination, in real time. The relation $\prec$ is acyclic.

The atomic global order property introduces two aspects: it relates atomic multicast primitives in real time and it relates messages multicast to possibly disjoint destinations. Capturing these real-time dependencies is fundamental to linearizability. For example, the execution in Figure 5.2 (top) does not satisfy atomic multicast extended with atomic global order: (a) since $w(1, v_1)$ is multicast (by client c) after $r(0, 1)$ is delivered by the replica at partition P_0, it follows from atomic global order that $r(0, 1) \prec w(1, v_1)$; and (b) from the delivery order of requests at P_1, $w(1, v_1) \prec r(0, 1)$, which leads to a cycle and violates atomic multicast's global total order.

5.2.3 Proof of correctness

In the following, we show that atomic multicast extended with the atomic global order property ensures that partitioned state machine replication executions are linearizable.

Let σ be an execution of partitioned state machine replication where (a) a client starts a request by multicasting the request to all the partitions involved in the request (i.e., partitions containing data read and written as part of the request), (b) when the request is delivered by a replica, the replica immediately executes the request and responds to the client, and (c) the client considers the request as finished after it receives a response from at least one replica in each partition involved in the request.

Let π be a total order of requests in σ that respects $\prec$, the order induced on requests by atomic multicast extended with atomic global order.

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. This follows from (i) the fact that replicas execute requests sequentially, in the order in which they are delivered, and (ii) the assumption that when executing a multi-partition request, a replica does not read data stored at other partitions (see Section 5.1.1).

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 (a) $C_i \prec C_j$; or (b) neither $C_i \prec C_j$ nor $C_j \prec C_i$. In case (a), C_i precedes C_j

in π; in case (b), since C_i and C_j are not related, we can choose a total order π where C_i appears before C_j.

For a contradiction, assume that $C_j \prec C_i$. Since C_i ends in real time before C_j starts (from our initial assumption), the client issuing C_i has received a response from a replica p in each of the partitions involved in the execution of C_i. And from the algorithm, p has delivered C_i before executing it and responding to the client. We conclude that C_j is multicast after p delivers C_i. From the atomic order property of atomic multicast, we have that $C_i \prec C_j$, which leads to a cycle and violates the global total order property of atomic multicast, reaching a contradiction.

5.3 Implementing Atomic Global Order

In this section, we present an early atomic multicast algorithm attributed to Skeen [15] and show that it does not guarantee the atomic global order property. We then extend this algorithm to ensure the property and argue about the correctness of the extended algorithm.

5.3.1 Skeen's atomic multicast

In Skeen's algorithm, each process assigns unique timestamps to multicast messages based on a logical clock [57]. 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. (We recall that Skeen's atomic multicast algorithm does not tolerate failures.)

(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 local timestamp to m (a tuple of clock value and partition id for breaking ties), stores it, and sends its local timestamp for m to all destinations in $m.dest$. Upon receiving local timestamps from all destinations in $m.dest$, a process computes m's final timestamp ts as the maximum among all received local timestamps for m. The process then ensures its logical clock is higher than or equal to ts.

(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

Algorithm 4 Skeen's protocol at partition P_x. Additions to satisfy atomic global order in gray.

```
$clock \leftarrow 0$                                   ▷ $p$'s logical clock
$local[\,] \leftarrow \emptyset$                       ▷ map from message to local timestamp at $p$
$final[\,] \leftarrow \emptyset$                       ▷ map from message to decided final timestamp
$acked \leftarrow \emptyset$                           ▷ set of messages ACKed by all their destinations
$del \leftarrow \emptyset$                             ▷ set of delivered messages

**multicast**$(m)$:
    send $\langle \text{START}, m \rangle$ to $m.dest$

**when** receive $\langle \text{START}, m \rangle$:
    $clock \leftarrow clock + 1$
    $local[m] \leftarrow \langle clock, P_x \rangle$
    send $\langle \text{LOCAL-TS}, m, local[m] \rangle$ to $m.dest$

**when** receive $\langle \text{LOCAL-TS}, m, ts \rangle$ from all partitions $m.dest$:
    $final[m] \leftarrow$ maximum $ts$ received for $m$
    $clock \leftarrow \max(clock, final[m])$
    tryDeliver()
    send $\langle \text{ACK}, m \rangle$ to $m.dest$

**when** receive $\langle \text{ACK}, m \rangle$ from all partitions in $m.dest$:
    $acked \leftarrow acked \cup \{m\}$
    tryDeliver()

**tryDeliver**():
    **for each** $m \in final \setminus del : m \in acked$ **in** $final[m]$ **order**
        **if** $\forall m' \in local \setminus del$ :
                $(m' \in final \wedge final[m] < final[m']) \vee$
                $(final[m] < local[m'])$ **then**
            $del \leftarrow del \cup \{m\}$
            deliver$(m)$
```

than the final timestamp of any messages p will deliver after m (intuitively, this holds because logical clocks are monotonically increasing).

The complete protocol is shown in Algorithm 4 (ignoring the extensions in gray). Figure 5.3 (top) shows an example execution of the algorithm, where two clients a and b multicast messages m and m' respectively, with $m.dest = \{P_x, P_y\}$ and $m'.dest = \{P_x, P_z\}$. Initially, message m is sent to its destinations and is assigned the local timestamps 1 (from P_x) and 5 (from P_y). As soon as P_y receives the local timestamp from P_x, it knows the final timestamp of m is 5. P_y updates its logical clock to 5 and can immediately deliver m: any new messages will be assigned a higher local timestamp at P_y. P_x, on the other hand, cannot deliver m immediately after it receives the local timestamp from P_y: in the mean time it assigned a local timestamp of 2 to m', and must wait for the final timestamp of m' before it knows which of the two messages must be delivered first.

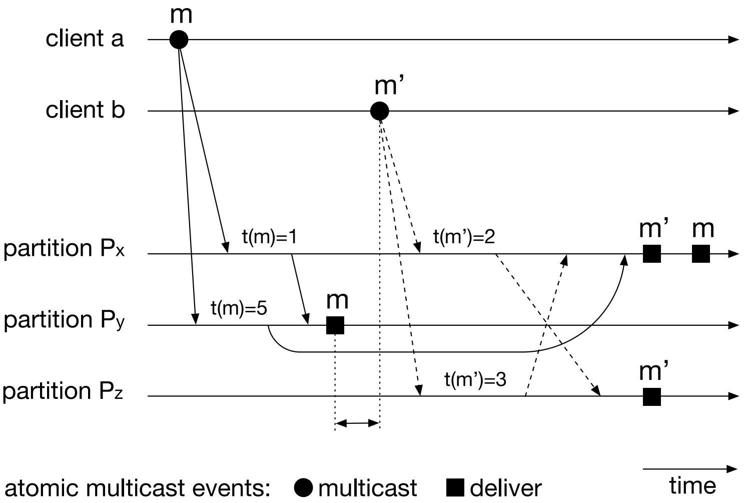
5.3.2 Extending Skeen's algorithm to ensure atomic global order

The execution in Figure 5.3 (top) demonstrates that Skeen's algorithm does not ensure atomic global order. Even though m' is multicast after m is delivered at partition P_y, in real-time, we have $m' \prec m$ at P_x.

We modify Skeen's algorithm to ensure atomic global order by including one extra property that needs to be satisfied: (iii) a process can only deliver a message with final timestamp ts once it knows that every destination in $m.dest$ will not assign a local timestamp smaller than or equal to ts. If (iii) is satisfied, once m is delivered with final timestamp ts by some process, no new message m' such that $m.dest \cap m'.dest \neq \emptyset$ can be assigned a final timestamp smaller than ts. The property is ensured by adding one extra message exchange between destinations. Once a process decides on the final timestamp of message m, after updating its clock if needed, it sends an acknowledgment to each other process in $m.dest$. A process can only deliver m once it receives an acknowledgment from every other process in $m.dest$. The additions to the protocol are shown in gray in Algorithm 4.

Figure 5.3 (bottom) shows a similar execution to the one in Figure 5.3 (top), but with the extended protocol. Partition P_y cannot deliver m at the moment it decides on the final timestamp: it must wait for P_x to acknowledge it. Since P_x timestamps m' before it acknowledges m, the delivery of m at P_y is delayed to a point after the multicast of m'. Thus, $m' \prec m$ does not create a cycle.

We now argue that the extended Skeen's protocol satisfies atomic global order. Let m and m' be two messages such that m' is multicast after m is delivered



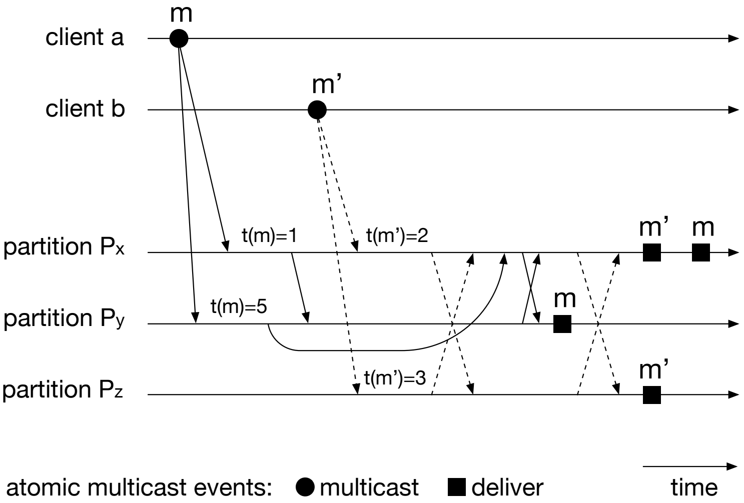


Figure 5.3. An execution showing that Skeen's atomic multicast violates atomic global order (top) and the extended algorithm that guarantees atomic global order (bottom).

at some destination P. Furthermore, assume for a contradiction that there is some destination in common, P', that delivers m' before it delivers m. When m is delivered at P, from the algorithm, it must have received an ACK for m from P'. Thus, P' must know the final timestamp ts of m, and must have advanced its logical clock past ts. Since the $\langle \text{START}, m' \rangle$ message arrives at P' after that point, it follows that the local timestamp assigned to m' at P', and consequently its final timestamp ts', must be larger than ts. But since P' delivers m before m', we have that $ts < ts'$, a contradiction.

5.4 Related work

We organize the related work from different perspectives. First we briefly comment on the existing atomic multicast properties and the here proposed atomic global order. We then examine existing atomic multicast protocols and discuss whether they satisfy atomic global order or not. Finally, we survey existing partitioned SMR protocols.

5.4.1 Atomic multicast properties

The functionality and semantics of an atomic broadcast or multicast protocol is defined by a set of properties (i.e., validity, agreement, integrity and order) that establish relations on the occurrence of its primitives in any given run of the protocol. Different applications may have different requirements, and these properties can be made stronger or weaker depending on the need of the application. Many protocols have been proposed in the literature, relying on different assumptions and satisfying different sets of properties [30]. Atomic global order differs from previously proposed ordering properties in that (i) it relates delivery and multicast events and (ii) it relates them in real-time. As previously shown, atomic global order allows partitioned systems to provide linearizability without further coordination among processes.

5.4.2 Atomic multicast algorithms

We focus on atomic multicast protocols based on destination agreement [30]. This is the class of protocols in which the order of messages results from agreement between the destination processes. We further divide these protocols in three classes: timestamp based, overlay based and deterministic merge based.

Timestamp based. In these algorithms, processes first agree on the assignment of message timestamps and then deliver messages in timestamp order. Every protocol discussed here is genuine, and employs a timestamping scheme similar to Skeen's, discussed in detail in Section 5.3.1.

The first protocol extending Skeen's protocol to be fault-tolerant by using consensus inside each destination group was presented in [40]. Each group acts as a process from Skeen's protocol, and relies on consensus to decide on timestamp proposals and advance the logical clock. Messages can be delivered in six communication steps. In [91], a similar protocol is proposed with optimizations to speed up delivery in specific circumstances.

In Scalatom [84], instead of using consensus inside each destination group to decide on timestamp proposals, a single consensus instance is executed among all destination processes. It can deliver messages in six communication steps. Scalatom also proposes and satisfies the additional property of *message size minimality*: protocol messages should have size proportional to the number of destination groups. We note that all algorithms discussed here also satisfy the minimality property.

FastCast [24] proposes the use of stable leaders and an optimistic execution path that can deliver messages in four communication steps, in the best case.

Instead of using consensus as a black-box, White-Box Atomic Multicast [43] weaves Skeen's timestamping scheme and Paxos into a unified protocol. The protocol is primary-based, and can deliver messages in three communication steps at group leaders, in the best case. Each group leader is responsible for assigning the local timestamp for its group, and decides when a message is safe for delivery.

RamCast [60] is a primary-based protocol that achieves high throughput and low latency through its use of Remote Direct Memory Access (RDMA). It combines ideas from Skeen and Protected Memory Paxos [3].

Overlay based. These algorithms rely on a predefined topology to propagate messages and to ensure atomic multicast properties.

In [31], a genuine atomic multicast protocol is proposed that uses a total order of groups as an overlay. A message m that needs to be multicast is initially sent to one group in $m.dest$. When the group receives m, consensus is used to order and deliver m inside the group, then m is forwarded to the next group in $m.dest$ (according to the total order of groups). A group that delivers m can only order the next message once it knows m is ordered in all of $m.dest$, after it receives an END message from the last group in $m.dest$.

Byzcast [23] is a byzantine fault-tolerant atomic multicast that arranges groups in a tree overlay. A message m enters the overlay in the lowest common ancestor of groups in $m.dest$. A group that receives m orders it and then propagates it down the tree, until all groups in $m.dest$ are reached. Byzcast may be considered partially genuine: delivering a message addressed to multiple groups may involve intermediary groups not part of the destination set.

Deterministic merge based. In these algorithms, each destination process applies a deterministic merge procedure to decide on the delivery order of received messages.

In [91], processes execute an unbounded sequence of rounds. Consensus is used inside each group to determine the set of messages proposed by the group in a given round. At the end of each round, processes gather messages from all groups and then deliver them in some deterministic order. The protocol is non-genuine.

Multi-Ring Paxos [68] builds on multiple instances of Ring Paxos [67], a ring-based consensus protocol. It provides guarantees akin to atomic multicast, but with a slightly different interface. Messages can only be sent to a single group, but receivers can subscribe to more than one group. Each group totally orders its messages and receivers use a deterministic merge to ensure a partial ordering of deliveries from the groups it subscribes to. Ridge [14] improves on Multi-Ring Paxos by reducing the latency inside each group and utilizing a timestamp-based merge procedure. Both protocols are non-genuine.

5.4.3 Existing algorithms and atomic global order

In the following, we discuss why, out of all surveyed protocols, only the round-based protocol described in [91] satisfies atomic global order.

Timestamp based. As shown in Section 5.3.1, Skeen's algorithm does not satisfy atomic global order. By the same argument, all of the algorithms that more closely match its execution (i.e., [24, 40, 91]) do not satisfy the property either.

In Scalatom [84], while a single instance of consensus is executed among groups in $m.dest$, a group's clock is only updated after it handles the decision for m's timestamp. It is possible for a group in $m.dest$ to propose a smaller timestamp even after some other group in $m.dest$ delivers m.

In the primary-based algorithms (i.e., [43, 60]), at the time a group leader delivers a message m with final timestamp ts, other group leaders in $m.dest$ may still propose local timestamps smaller than ts. This also holds for PrimCast, our proposed protocol from Chapter 4.

Overlay based. In the protocols relying on an overlay for partial order, messages may be ordered by groups in sequence, either following a total order [31] or down a tree [23]. Consider two messages m and m' and groups g and h such that $g \in m.dest$ and $h \in m.dest \cap m'.dest$, and g is earlier than h in the overlay. If m' is multicast after m is delivered by g, but before m its propagated to h, h may deliver m' before m, violating atomic global order.

Deterministic merge based. For Multi-Ring Paxos [68] and Ridge [14], consider the following case. Two messages m and m' are ordered by groups g and h respectively. A receiver subscribing only to g delivers m, and only then m' is multicast to h. The proposed merge procedures do not prevent another receiver, subscribing to both g and h, from delivering m' before m, violating atomic global order.

Out of all surveyed protocols, only the non-genuine, round-based protocol described in [91] satisfies atomic global order. If some process delivers a message m, it follows that the round to which m belongs is closed in all groups. Any message multicast after that must belong to some later round, ensuring it is delivered after m.

5.4.4 Partitioned SMR

The quest for scalable SMR very often involves partitioning techniques, where handling multi-partition operations (MPOs) efficiently is one of the main challenges.

Scalable State Machine Replication (S-SMR) [13] is an approach that achieves scalable throughput and linearizability without constraining service commands or adding additional complexity to their implementation. S-SMR partitions the service state and relies on an atomic multicast primitive to consistently order commands within and across partitions. It is shown that simply ordering commands consistently across partitions is not enough to ensure linearizability in partitioned state machine replication. To ensure linearizability, S-SMR implements execution atomicity, a property that prevents invalid command interleavings. Partitions involved in the same operation signal each other such that all of them finish the operation before signaling the client (see also Figure 5.2).

In [62], the authors propose a genuine protocol based on Skeen's total order multicast [15] to order MPOs. The inter-partition coordination for MPOs is removed from the critical execution path of operations. This is achieved by postponing the execution of MPOs to a future time when their ordering has already been agreed across the partitions involved. For this, a new consensus interface and properties are proposed. Operations are executed in rounds. The proposal primitive allows to propose operations with the rounds intended to execute them. While single partition operations can be ordered quickly and execute soon, MPOs have to go through an inter-partition procedure. Scheduling MPOs for future rounds allows other operations to execute while the MPOs are being ordered. To ensure linearizability, when a process starts executing an MPO, it notifies all involved partitions. A process only replies to the client after having received this notification from all involved partitions. This solution is similar to the one presented in [13] and ensures that the reply externalized to the client is consistent with linearizability.

DynaStar [59] is a partitioned SMR solution that provides dynamic state partitioning to handle workloads with varying access patterns. Data can be moved between partitions, and a location oracle is used to monitor the workload and to re-calculate an optimized partitioning on demand. In DynaStar there is no multi-partition execution of commands. If a command accesses multiple partitions, all data is temporarily moved to a single partition that is then responsible for executing the command.

Tempo [34] is a leaderless partitioned SMR protocol that relies on a timestamping scheme similar to Skeen's. Each command has a coordinator replica that communicates with the destination replicas to replicate the command and to decide on its timestamp. To ensure linearizability, replicas exchange information about each timestamp they assign and will only execute a command once every command with a lower timestamp is known.

5.5 Discussion

Partitioned state machine replication extends classic state machine replication (SMR) with the notion of state partitioning (or sharding). In both approaches, clients propagate requests to the replicas, which execute the requests sequentially in a consistent order. In the case of SMR, every request concerns all replicas, as each replica stores the full application state. In partitioned SMR, the application state is divided into partitions, and each request accesses data in one or more partitions. Clients must propagate requests to the partitions concerned by the data accessed in the request.

Many proposals that adopt the SMR model use an atomic broadcast primitive to order requests (e.g., [79, 81]). In the case of partitioned SMR, atomic multicast is more appropriate than atomic broadcast to propagate requests to replicas consistently (e.g., [92]) since propagating all requests to all replicas defeats the purpose of data partitioning. One can observe that as partitioned SMR generalizes classic SMR, atomic multicast generalizes atomic broadcast. Despite this analogy, there is an “asymmetry" in how the ordering abstractions are used in the replication approaches. In SMR, after delivering a request, replicas execute the request and reply to the client. No coordination between replicas is needed as part of the execution of a request. In partitioned SMR, as part of the execution of a multi-partition request, replicas in the involved partitions must coordinate to ensure linearizability. We show that this coordination is needed because atomic multicast with global total order does not capture real-time dependencies between requests. When equipped with atomic multicast that ensures atomic global order, replica coordination is not necessary.

Chapter 6

Conclusion

With the emergence of cloud infrastructure providers in the last few years, distributing servers around the globe has never been easier. Still, designing applications for such large scale deployments remains a challenge. One of the many tradeoffs involved is that between consistency and availability. Relaxing consistency guarantees can be a way of achieving better performance and availability, but it may place a burden on application developers and end users by increasing complexity and breaking expected application semantics. In contrast, a strongly consistent system typically provides a simpler more transparent interface.

Replication is the fundamental technique used to achieve reliability in distributed systems. However, full replication, an approach where every server executes every request, does not scale. A common solution to solving this problem is to instead rely on partial replication. In a partially replicated system, application data is split into partitions, each partition replicated by only a subset of the servers. Atomic multicast is a core abstraction for building strongly consistent partitioned systems. It provides a way for requests to be reliably sent to one or more groups of servers, and ensures a partial ordering of deliveries.

In this thesis, we explore the design of strongly consistent and geographically distributed applications relying on the atomic multicast abstraction.

6.1 Research assessment

This thesis presents 3 contributions: (1) the design and evaluation of GlobalFS, a strongly consistent geographically distributed file system; (2) PrimCast, a genuine atomic multicast protocol that can deliver messages in three communication steps at every destination; and (3) linearizable atomic multicast, a stronger version of atomic multicast that allows for linearizable applications without further coordination.

GlobalFS. With GlobalFS we propose a design for a geographically distributed file system providing a familiar POSIX-like interface. GlobalFS handles data and metadata separately, the latter being handled by a custom protocol built upon atomic multicast. Each file system operation is implemented using one of four execution modes, based on the requirements of the operation. GlobalFS provides low latency for single-region local commands while allowing for consistent global operations across regions. We validate our design with a global deployment of our prototype in Amazon EC2.

PrimCast. An atomic multicast protocol is genuine if only senders and destinations take steps to deliver a message. A genuine protocol can scale with the number of groups, given the right workload, and should also allow for fast delivery of messages when sender and destinations are close together. PrimCast is a genuine atomic multicast protocol that can deliver messages at every destination in three communication steps in the absence of conflicting messages, and five otherwise. We provide the complete algorithm for PrimCast and its proof of correctness. We also show how loosely synchronized clocks can be used to reduce the impact of the convoy effect under high load. Our experimental evaluation shows that that PrimCast achieves lower latency than state-of-the-art approaches while providing higher or comparable throughput.

Linearizable Atomic Multicast. We formalize a stronger version of atomic multicast that allows for building linearizable applications without the need for further coordination. We show why the guarantees of atomic multicast are insufficient and propose a stronger ordering property, atomic global order, that fixes the issue. We also describe how a classic atomic multicast protocol can be modified to provide the stronger property.

6.2 Future directions

In the following, we discuss some possible directions for continuing the work presented in this thesis.

6.2.1 GlobalFS

While Multi-Ring Paxos can be tuned to work in a global deployment [12], it was originally designed to maximize throughput in a local network. GlobalFS' usage of Multi-Ring Paxos only provides geographical scalability for single-partition operations: multi-partition operations need to go through the global ring. Using a genuine protocol designed for WAN environments, such as PrimCast, might allow for a more flexible and transparent partitioning scheme.

More execution modes could be devised to improve performance in specific situations. These execution modes could be enabled automatically given some predefined policy or by means of cues given by the application (e.g., as done in [101]). Some examples:

- Clients could opt to cache data writes until a file is closed, relaxing the consistency of a given file to close-to-open.

- A client could fetch a complete snapshot of a file's metadata upon opening the file for reading only, if the client is not interested in new modifications until reopening the file.

- A lease mechanism could be used to allow for modifications to remote files to be temporarily stored in the closest partition until the file is closed, shifting the cost of synchronization to readers.

Access to application specific workloads could help identify such optimization opportunities.

6.2.2 PrimCast

The protocol could be extended to support the crash-recovery model and group membership reconfiguration. Also, as is typical with these kinds of protocols, practical applications of PrimCast may raise interesting engineering issues. While loosely synchronized clocks proved to be effective in reducing the convoy effect in the evaluated scenario, it would be interesting to see how it performs with more complex scenarios and workloads. Finally, it is also worthwhile to investigate the possibility of modifying PrimCast (in particular its quorum clock mechanism) to provide linearizable atomic multicast while still allowing for message delivery in three communication steps.

6.2.3 Linearizable Atomic Multicast

The proposed atomic global order property is not minimal, in that it rules out executions that do ensure linearizability. One example is the execution depicted in Figure 5.2 (bottom), which is linearizable but is not allowed by atomic global order. To see why, notice that since request $r(0,1)$ is delivered at P_0 before $w(1,v_1)$ is multicast by client c, from atomic global order, $r(0,1) \prec w(1,v_1)$. Therefore, at P_1, $w(1,v_1)$ cannot be delivered before $r(0,1)$ (i.e., $w(1,v_1) \prec r(0,1)$) since this would create a cycle. Thus, one open question is whether one can come up with a property stronger than global total order but weaker than atomic global order that still allows for linearizable applications without further coordination.

Bibliography

[1] *IEEE Std 1003.1-2001 Standard for Information Technology – Portable Operating System Interface (POSIX) Base Definitions, Issue 6*. IEEE, 2001. ISBN 1-85912-247-7 (UK), 1-931624-07-0 (US), 0-7381-3047-8 (print), 0-7381-3010-9 (PDF), 0-7381-3129-6 (CD-ROM).

[2] Atul Adya, William J Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R Douceur, Jon Howell, Jacob R Lorch, Marvin Theimer, and Roger P Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *5th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2002.

[3] Marcos K. Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J. Marathe, and Igor Zablotchi. The impact of RDMA on agreement. *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing*, 2019.

[4] Tarek Ahmed-Nacer, Pierre Sutra, and Denis Conan. The convoy effect in atomic multicast. In *2016 IEEE 35th Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 67–72. IEEE, 2016.

[5] Amazon EC2 instances types. `https://aws.amazon.com/ec2/instance-types/`. [Internet Archive: `https://web.archive.org/web/20171230055211/https://aws.amazon.com/ec2/instance-types/`].

[6] Apache Thrift. `https://thrift.apache.org`. [Accessed: 2022-12-18].

[7] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2004. ISBN 0471453242.

[8] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2010.

[9] BeeGFS. `https://www.beegfs.io`. [Accessed: 2022-12-18].

[10] Samuel Benz and Fernando Pedone. Elastic paxos: A dynamic atomic multicast protocol. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 2157–2164. IEEE, 2017.

[11] Samuel Benz, Parisa Jalili Marandi, Fernando Pedone, and Benoît Garbinato. Building global and scalable systems with atomic multicast. In *15th ACM/IFIP/USENIX International Middleware Conference*, Middleware, 2014.

[12] Samuel Benz, Leandro Pacheco de Sousa, and Fernando Pedone. Stretching multi-ring paxos. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, pages 492–499, 2016.

[13] Carlos Eduardo Bezerra, Fernando Pedone, and Robbert Van Renesse. Scalable state-machine replication. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 331–342. IEEE, 2014.

[14] Carlos Eduardo Bezerra, Daniel Cason, and Fernando Pedone. Ridge: high-throughput, low-latency atomic multicast. In *2015 IEEE 34th Symposium on Reliable Distributed Systems (SRDS)*, pages 256–265. IEEE, 2015.

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

[16] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[17] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[18] Jean-Michel Busca, Fabio Picconi, and Pierre Sens. Pastis: a highly-scalable multi-user peer-to-peer file system. In *Euro-Par*, 2005.

[19] Philip H. Carns, W. B. Ligon III, Robert B. Ross, and Rajeev Thakur. PVFS: A parallel file system for linux clusters. In *4th Annual Linux Showcase and Conference*, ALS, 2000.

[20] Ceph block storage. `https://ceph.com/en/discover/technology/#block`. [Accessed: 2022-12-18].

[21] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[22] Bernadette Charron-Bost, Fernando Pedone, and Andre Schiper. *Replication: Theory and Practice*, volume 5959. Springer, 2010.

[23] Paulo Coelho, Tarcisio Ceolin Junior, Alysson Bessani, Fernando Dotti, and Fernando Pedone. Byzantine fault-tolerant atomic multicast. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 39–50. IEEE, 2018.

[24] Paulo R Coelho, Nicolas Schiper, and Fernando Pedone. Fast atomic multicast. In *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pages 37–48. IEEE, 2017.

[25] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31 (3):8, 2013.

[26] James Cowling and Barbara Liskov. Granola: Low-overhead distributed transaction coordination. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 223–235, Boston, MA, June 2012. USENIX Association.

[27] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *18th ACM Symposium on Operating Systems Principles*, SOSP, 2001.

[28] Alex Davies and Alessandro Orsaria. Scale out with GlusterFS. *Linux Journal*, 2013(235), November 2013. ISSN 1075-3583.

[29] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP, 2007.

[30] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys (CSUR)*, 36(4):372–421, 2004.

[31] Carole Delporte-Gallet and Hugues Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 107–122, 2000.

[32] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[33] Email exchange on CephFS mailing list. `https://www.mail-archive.com/ceph-users@lists.ceph.com/msg23788.html`. [Accessed: 2022-12-18].

[34] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 178–193, New York, NY, USA, 2021. ACM.

[35] FastCast implementation. `https://bitbucket.org/paulo_coelho/libmcast`. [Accessed: 2022-12-18].

[36] Alan Fekete, David Gupta, Victor Luchangco, Nancy Lynch, and Alex Shvartsman. Eventually-serializable data services. *Theoretical Computer Science*, 220, 1999.

[37] Colin J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. In *11th Australian Computer Science Conference*, pages 55–66, University of Queensland, Australia, 1988.

[38] File System in User Space (FUSE). `https://github.com/libfuse/libfuse`. [Accessed: 2022-12-18].

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

[40] Udo Fritzke, Philippe Ingels, Achour Mostéfaoui, and Michel Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of International Symposium on Reliable Distributed Systems (SRDS)*, pages 578–585, 1998.

[41] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *19th ACM Symposium on Operating Systems Principles*, SOSP, 2003.

[42] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33 (2):51–59, jun 2002.

[43] Alexey Gotsman, Anatole Lefort, and Gregory Chockler. White-Box atomic multicast. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 176–187. IEEE, 2019.

[44] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *Computer*, 30(4):68–74, 1997.

[45] Rachid Guerraoui and André Schiper. Total order multicast to multiple groups. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 578–585. IEEE, 1997.

[46] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2): 297–316, 2001.

[47] Rachid Guerraoui, Jad Hamza, Dragos-Adrian Seredinschi, and Marko Vukolic. Can 100 machines agree? *CoRR*, abs/1911.07966, 2019. URL `http://arxiv.org/abs/1911.07966`.

[48] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca, NY, USA, 1994.

[49] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[50] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference*, ATC, 2010.

[51] Felix Hupfeld, Toni Cortes, Björn Kolbeck, Jan Stender, Erich Focht, Matthias Hess, Jesus Malo, Jonathan Marti, and Eugenio Cesario. The XtreemFS architecture – a case for object-based file systems in grids. *Concurrency and Computation: Practice and Experience*, 20(17):2049–2060, 2008.

[52] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[53] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008.

[54] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35(11):190–201, 2000.

[55] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical physical clocks. In *International Conference on Principles of Distributed Systems*, pages 17–32. Springer, 2014.

[56] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2), April 2010.

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

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

[59] Long Hoang Le, Enrique Fynn, Mojtaba Eslahi-Kelorazi, Robert Soulé, and Fernando Pedone. Dynastar: Optimized dynamic partitioning for scalable state machine replication. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1453–1465. IEEE, 2019.

[60] Long Hoang Le, Mojtaba Eslahi-Kelorazi, Paulo Coelho, and Fernando Pedone. RamCast: RDMA-based atomic multicast. In *Proceedings of the 22nd International Middleware Conference*, pages 172–184, 2021.

[61] LevelDB. `https://github.com/google/leveldb`. [Accessed: 2022-12-18].

[62] Zhongmiao Li, Peter Van Roy, and Paolo Romano. Enhancing throughput of partially replicated state machines via multi-partition operation scheduling. In *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*, pages 1–10, 2017.

[63] LibEvent. `https://libevent.org`. [Accessed: 2022-12-18].

[64] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical report, Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[65] Guoliang Liu, Liuying Ma, Pengfei Yan, Shuai Zhang, and Liu Liu. Design and implementation of GeoFS: A wide-area file system. In *9th IEEE International Conference on Networking, Architecture, and Storage*, NAS, 2014.

[66] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. Consistency, availability, and convergence. Technical report, TR-11-22, Computer Science Department, UT Austin, May 2011.

[67] Parisa Jalili Marandi, Marco Primi, Nicolas Schiper, and Fernando Pedone. Ring Paxos: A high-throughput atomic broadcast protocol. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN, 2010.

[68] Parisa Jalili Marandi, Marco Primi, and Fernando Pedone. Multi-ring paxos. In *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pages 1–12. IEEE, 2012.

[69] MooseFS. `https://www.moosefs.org`. [Accessed: 2022-12-18].

[70] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: a read/write peer-to-peer file system. In *5th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2002.

[71] Robert HB Netzer and Jian Xu. Necessary and sufficient conditions for consistent global snapshots. *IEEE Transactions on Parallel and distributed Systems*, 6(2):165–169, 1995.

[72] Salman Niazi, Mahmoud Ismail, Seif Haridi, Jim Dowling, Steffen Grohsschmiedt, and Mikael Ronström. HopsFS: Scaling hierarchical file system metadata using NewSQL databases. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 89–104, 2017.

[73] ObjectiveFS. `https://objectivefs.com`. [Accessed: 2022-12-18].

[74] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *USENIX Annual Technical Conference*, ATC, 1999.

[75] OrangeFS. `https://www.orangefs.org`. [Accessed: 2022-12-18].

[76] Michael Ovsiannikov, Silvius Rus, Damian Reeves, Paul Sutter, Sriram Rao, and Jim Kelly. The Quantcast file system. *Proc. of the VLDB Endowment*, 6 (11):1092–1101, 2013.

[77] Leandro Pacheco, Raluca Halalai, Valerio Schiavoni, Fernando Pedone, Etienne Riviere, and Pascal Felber. Globalfs: A strongly consistent multi-site file system. In *Reliable Distributed Systems (SRDS), 2016 IEEE 35th Symposium on*, pages 147–156. IEEE, 2016.

[78] Leandro Pacheco, Fernando Dotti, and Fernando Pedone. Strengthening atomic multicast for partitioned state machine replication. In *Proceedings of the 11th Latin-American Symposium on Dependable Computing*, pages 51–60, 2022.

[79] Marta Patiño Martinez, Ricardo Jiménez-Peris, Bettina Kemme, and Gustavo Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, nov 2005.

[80] Swapnil Patil and Garth Gibson. Scale and concurrency of GIGA+: File system directories with millions of files. In *9th USENIX Conference on File and Storage Technologies*, FAST, 2011.

[81] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[82] Kenneth W Preslan, Andrew P Barry, Jonathan E Brassow, Grant M Erickson, Erling Nygaard, Christopher J Sabol, Steven R Soltis, David C Teigland, and Matthew T O'Keefe. A 64-bit, shared disk file system for linux. In *16th IEEE Symposium on Mass Storage Systems*, 1999.

[83] Michel Raynal, André Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Inf. Process. Lett.*, 39(6):343–350, October 1991.

[84] Luis Rodrigues, Rachid Guerraoui, and André Schiper. Scalable atomic multicast. In *International Conference on Computer Communications and Networks*, pages 840–847, 1998.

[85] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *18th ACM Symposium on Operating Systems Principles*, SOSP, 2001.

[86] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Middleware Conference*, Middleware, 2001.

[87] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 1990.

[88] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, April 1990.

[89] Nicolas Schiper. *On multicast primitives in large networks and partial replication protocols*. PhD thesis, Università della Svizzera italiana, 2009.

[90] Nicolas Schiper and Fernando Pedone. Optimal atomic broadcast and multicast algorithms for wide area networks. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 384–385. ACM, 2007.

[91] Nicolas Schiper and Fernando Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *International conference on Distributed computing and networking (ICDCN)*, pages 147–157, 2008.

[92] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. P-store: Genuine partial replication in wide area networks. In *Symposium on Reliable Distributed Systems (SRDS)*, 2010.

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

[94] Philip Schwan. Lustre: Building a file system for 1000-node clusters. In *Linux Symposium*, 2003.

[95] SeaweedFS. https://github.com/seaweedfs/seaweedfs. [Accessed: 2022-12-18].

[96] Amazon Time Sync Service. https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-the-amazon-time-sync-service/. [Accessed: 2022-12-18].

[97] John Shaw and Julian Dyke. Oracle Cluster File System (OCFS). In *Pro Oracle Database 10g RAC on Linux*, pages 171–200. Apress, 2006. ISBN 978-1-59059-524-4.

[98] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *26th IEEE Symposium on Mass Storage Systems and Technologies*, MSST, 2010.

[99] Dimokritos Stamatakis, Nikos Tsikoudis, Ourania Smyrnaki, and Kostas Magoutis. Scalability of replicated metadata services in distributed file systems. In *12th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems*, DAIS, 2012.

[100] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, February 2003.

[101] Jeremy Stribling, Yair Sovran, Irene Zhang, Xavid Pretzer, Jinyang Li, M Frans Kaashoek, and Robert Morris. Flexible, wide-area storage for distributed systems with WheelFS. In *6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2009.

[102] Sun Microsystems, Inc. NFS: Network file system protocol specification. RFC 1094, Network Information Center, SRI International, March 1989.

[103] Vasily Tarasov, Saumitra Bhanage, Erez Zadok, and Margo Seltzer. Benchmarking file system benchmarking: It *is* rocket science. In *13th USENIX Workshop on Hot Topics in Operating Systems*, HotOS, 2011.

[104] Vasily Tarasov, Abhishek Gupta, Kumar Sourav, Sagar Trehan, and Erez Zadok. Terra incognita: On the practicality of user-space file systems. In *7th USENIX Workshop on Hot Topics in Storage and File Systems*, HotStorage, 2015.

[105] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies*, FAST, 2015.

[106] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 1–12. ACM, 2012.

[107] Tokio asynchronous runtime. `https://tokio.rs`. [Accessed: 2022-12-18].

[108] Robbert Van Renesse and Fred B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[109] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4):472–484, 2014.

[110] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *9th ACM Symposium on Operating Systems Principles*, SOSP, 1983.

[111] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2006.

[112] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, and Carlos Maltzahn. CRUSH: Controlled, scalable, decentralized placement of replicated data. In *ACM/IEEE conference on Supercomputing*, SC, 2006.

[113] White-Box Atomic Multicast implementation. `https://github.com/imdea-software/atomic-multicast`. [Accessed: 2022-12-18].