View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by RERO DOC Digital Library

On the many faces of atomic multicast

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
Paulo Coelho

under the supervision of
Fernando Pedone

May 2019

Dissertation Committee

Antonio Carzaniga	Università della Svizzera italiana, Lugano, Switzerland
Robert Soulé	Università della Svizzera italiana, Lugano, Switzerland
Dan Alistarh	Institute of Science and Technology, Klosterneuburg, Austria
José Orlando Pereira	University of Minho, Braga, Portugal

Dissertation accepted on 6 May 2019

Research Advisor	PhD Program Director
Fernando Pedone	**Prof. Walter Binder, Prof. Olaf Schenk**

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.

Paulo Coelho
Lugano, 6 May 2019

To my wife Patrícia,
my son Gabriel and my daughter Giovanna

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport

Abstract

Many current online services need to serve clients distributed across geographic areas. Coordinating highly available and scalable geographically distributed replicas, however, is challenging. While State Machine Replication is the most direct way of achieving availability, no scalability comes from the traditional approach. Typically, scalability is obtained by partitioning the original application state among groups of servers, which leads to further challenges. Atomic multicast is a group communication abstraction that groups processes, providing reliability and ordering guarantees, and can be explored to provide partially replicated applications a scalable and consistent alternative. This work confronts the challenges of providing practical group communication abstractions for crash fault-tolerant and Byzantine fault-tolerant (BFT) models.

Although there are plenty of atomic multicast algorithms that tolerate crash failures, they suffer from two major issues: (a) high latency for messages addressed to multiple groups, and (b) low performance when proportion of messages to multiple groups is high. To solve the first problem and reduce the latency of multi-group messages, this work presents FastCast, an algorithm with unprecedented four communication delays. The second problem can be addressed by maximizing the proportion of single-group messages and eliminating additional communication among groups to execute operations. In this direction, this document introduces GeoPaxos, a protocol that partitions the ordering of operations like atomic multicast while still keeping the state fully replicated.

In the BFT model, the task is more challenging, since servers can behave arbitrarily. This thesis presents ByzCast, the first algorithm that tolerates Byzantine failures. ByzCast is hierarchical and introduces a new class of atomic multicast defined as partially genuine.

Lastly, since at the very core of most strong consistent replicated system resides a consensus protocol, the thesis concludes with Kernel Paxos, a Paxos implementation provided as a loadable kernel module, providing at the same time high performance, and abstracting ordering from the application execution.

Acknowledgements

I am very thankful to everyone that contributed somehow to this thesis. First of all, I wish to thank Professor Fernando Pedone for the support and patience. His dedication and enthusiasm with each student are an example I will try to carry on in my career.

I am grateful to the dissertation committee members, Antonio Carzaniga, Dan Alistarh, José Orlando Pereira and Robert Soulé for the time dedicated to my thesis and the helpful feedback.

I am very happy to be part of this research group and would like to truly express my gratitude to all the current and former colleagues I had the pleasure to work with: Daniele, Edson, Loan, Long, Mojtaba, Odorico, Pietro, Sam, Theo, Tu, Vahid, and especially Leandro and Enrique for the fruitful discussions and the friendship.

I wish to thank the Brazilian funding agency Conselho Nacional de Pesquisa (CNPq) and the Swiss Government for the financial support.

I wish to express all my gratitude to my parents Paulo and Jôze, who have always been there for me and for the example of family and partnership. Lastly, I would like to thank my wife Patrícia and my kids Gabriel and Giovanna for standing by my side these four years of PhD, providing me courage, love and unconditional support.

Preface

The result of this research appears in the following papers:

Coelho, Paulo; Schiper, Nicolas; Pedone, Fernando. *Fast Atomic Multicast*. 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2017, Denver.

Coelho, Paulo; Junior, Tarcisio Ceolin; Bessani, Alysson; Dotti, Fernando; Pedone, Fernando. *Byzantine Fault-Tolerant Atomic Multicast*. 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018, Luxembourg City.

Coelho, Paulo; Pedone, Fernando. *Geographic State Machine Replication*. IEEE 37th International Symposium on Reliable Distributed Systems (SRDS), 2018, Salvador.

Esposito, Emanuele Giuseppe; Coelho, Paulo; Pedone, Fernando. *Kernel Paxos*. IEEE 37th International Symposium on Reliable Distributed Systems (SRDS), 2018, Salvador.

Contents

Contents **xiii**

List of Figures **xvii**

List of Tables **xix**

1 Introduction **1**
- 1.1 Contributions of the thesis . . . 2
  - 1.1.1 Contributions in the crash-failure model . . . 3
  - 1.1.2 Contribution in the Byzantine-failure model . . . 4
- 1.2 System model and definitions . . . 4
  - 1.2.1 Processes, groups, and links . . . 4
  - 1.2.2 Consensus . . . 5
  - 1.2.3 Reliable and atomic multicast in crash-failure model . . . 6
  - 1.2.4 Atomic multicast in Byzantine-failure model . . . 7

2 Making atomic multicast faster **9**
- 2.1 Baseline Atomic Multicast . . . 10
  - 2.1.1 Overview . . . 11
  - 2.1.2 Detailed algorithm . . . 12
  - 2.1.3 Time complexity . . . 13
- 2.2 Fast Atomic Multicast . . . 13
  - 2.2.1 Overview . . . 13
  - 2.2.2 Detailed algorithm . . . 15
  - 2.2.3 Time complexity . . . 16
- 2.3 Proofs of Correctness . . . 18
  - 2.3.1 Proofs for Propositions 1 and 2 . . . 18
  - 2.3.2 Proof of correctness for Algorithm 1 . . . 19
  - 2.3.3 Proof of correctness for Algorithm 2 . . . 23
- 2.4 Performance evaluation . . . 27

2.4.1 Evaluation rationale . . . 27
2.4.2 Implementation and environments . . . 28
2.4.3 Social network benchmark . . . 29
2.4.4 Microbenchmark in LAN . . . 30
2.4.5 Microbenchmark in emulated WAN . . . 32
2.4.6 Microbenchmark in real WAN . . . 34
2.4.7 Social network in emulated WAN . . . 35
2.4.8 Summary . . . 36
2.5 Related work . . . 38
2.6 Conclusion . . . 39

3 Making atomic multicast safer 41
3.1 Byzantine Fault Tolerant Atomic Multicast . . . 43
3.1.1 Rationale . . . 43
3.1.2 Protocol . . . 44
3.1.3 Optimizations . . . 46
3.2 Proof of Correctness . . . 47
3.3 Implementation . . . 49
3.4 Performance evaluation . . . 50
3.4.1 Evaluation rationale . . . 51
3.4.2 Environments and configuration . . . 52
3.4.3 Overlay tree versus workload . . . 54
3.4.4 Scalability of ByzCast in LAN . . . 55
3.4.5 Throughput versus latency in LAN . . . 57
3.4.6 Latency without contention in LAN . . . 57
3.4.7 Performance with mixed workload in LAN . . . 58
3.4.8 Latency without contention in WAN . . . 58
3.4.9 Performance with mixed workload in WAN . . . 59
3.5 Related work . . . 59
3.5.1 Atomic Multicast . . . 60
3.5.2 Scalable BFT . . . 61
3.6 Conclusion . . . 61

4 Speeding up state machine replication in wide-area networks 63
4.1 Introduction . . . 63
4.2 System model specifics . . . 65
4.3 Overview . . . 65
4.3.1 Partial ordering of operations . . . 65
4.3.2 Optimizing performance . . . 67

4.3.3 Fault tolerance 67
4.3.4 Execution Model 68
4.4 Design 69
4.4.1 The ordering protocol 69
4.4.2 Extensions and optimizations 70
4.4.3 Practical considerations 72
4.5 Proof of correctness 72
4.6 Implementation 74
4.7 Evaluation 74
4.7.1 Performance in the LAN 75
4.7.2 Performance in the WAN 76
4.8 Related work 82
4.9 Conclusion 85

5 Speeding up Paxos 87
5.1 Background on Paxos 88
5.1.1 Paxos and state-machine replication 89
5.1.2 Optimizations 89
5.2 Paxos in the kernel 90
5.2.1 Linux kernel and TCP/IP stack 90
5.2.2 Kernel Paxos architecture 92
5.2.3 Message flow in Kernel Paxos 93
5.3 Implementation 94
5.4 Performance 94
5.4.1 Evaluation rationale 95
5.4.2 Environment 95
5.4.3 Throughput in a LAN 96
5.4.4 Latency in a LAN 98
5.4.5 Performance with similar number of clients 99
5.4.6 Context-switch overhead 100
5.4.7 Kernel Paxos in a 10Gbps network 102
5.4.8 Summary 103
5.5 Related work 103
5.5.1 Protocols that exploit special topologies 103
5.5.2 Protocols that exploit special hardware 104
5.5.3 Protocols that exploit message semantics 104
5.6 Conclusion 104

6 Conclusions **107**
6.1 Research assessment 107
6.2 Future directions 109

Bibliography **111**

Figures

2.1 Diagrammatic representation of BaseCast and FastCast. 12
2.2 Configuration in WAN. 29
2.3 Throughput for single-group messages in a LAN. 30
2.4 Atomic multicast in a LAN. 31
2.5 Atomic multicast in an emulated WAN. 33
2.6 Atomic multicast in a real WAN. 35
2.7 Social network application in an emulated WAN. 36

3.1 ByzCast overlay tree and sample execution. 43
3.2 Executions of ByzCast with local and global messages. 51
3.3 ByzCast performance with global messages. 53
3.4 Throughput in a LAN. 55
3.5 Throughput vs. latency in a LAN. 56
3.6 Single-client latency in a LAN. 57
3.7 Latency CDF with 10% of global messages. 58
3.8 Single-client latency in WAN. 59
3.9 Normalized throughput with mixed workload in a WAN. 60
3.10 Latency CDF with mixed workload. 60

4.1 Simple GeoPaxos executions. 66
4.2 GeoPaxos deployment in three regions. 68
4.3 Performance in LAN. 75
4.4 Impact of dynamic preferred site change on throughput and latency. 76
4.5 Latency in WAN. 79
4.6 Impact of the convoy effect on latency. 80
4.7 Impact of the convoy effect on throughput. 81
4.8 Latency and throughput in the presence of failures. 82

5.1 Multi-Paxos optimization. 90
5.2 Kernel Paxos architecture. 92

5.3 Kernel Paxos event-driven approach for Ethernet types. 93
5.4 Main components of Kernel Paxos module. 95
5.5 Throughput and median latency for increasing number of clients. 96
5.6 Throughput and latency CDF for selected number of clients. 97
5.7 Performance in 1Gbps and 10Gbps setups. 101

Tables

2.1 Comparison of atomic multicast protocols. 37

3.1 Latencies within Amazon EC2 infrastructure. 52
3.2 Uniform and skewed workloads. 54
3.3 Optimization model outcomes for uniform and skewed workloads. 54

4.1 Average RTT between Amazon EC2 regions, ... 77
4.2 Minimum inter-region delays and operation type. 84

5.1 Performance of user- and kernel-space echo application. 91
5.2 Context-switches versus number of clients. 101

Chapter 1

Introduction

Many modern online applications require scalable performance and high availability. Scalable performance guarantees that by increasing the system resources (e.g., servers) the application can handle additional client requests. High availability ensures that the application remains operational despite server crashes and datacenter disasters. Designing systems that combine scalability and fault tolerance, however, is challenging.

State Machine Replication (SMR) is largely used as a strategy to provide availability and reliability by having multiple replicas of a service — sometimes in different geographical regions — executing the same operations in the same order [47, 77]. With enough replicas performing correctly, SMR behaves just like a single server, with the necessary redundancy to exhibit failure transparency to clients. To behave accordingly and guarantee that every non-faulty server executes the same operations in the same order, servers must solve a distributed problem known as consensus [33]. Consensus is a fundamental distributed problem in which a set of processes must eventually decide on a common value among values proposed by each process. Atomic broadcast [31] is an abstraction that guarantees the total order of delivered operations, usually relying on consensus to decide on the next operation to be executed by the servers. If operations are deterministic, then replicas will reach the same state and produce the same output upon executing the same sequence of operations.

Although SMR solves the availability problem, it does not bring any benefits in terms of scalability, no matter how fast it can order operations. Because all the replicas need to execute every operation, the performance of SMR is similar to a single server's. Moreover, if replicas are geographically distributed, clients experience can be worse than a single server model since faraway replicas have to coordinate to order each operation before executing and replying.

One strategy to increase scalability would be to weaken consistency guarantees. While weak consistency has proved successful in some contexts (e.g., [10, 25, 30, 83]), it is not appropriate to every application and often places the burden on the clients, who must cope with non-intuitive application behavior. Strong consistency (e.g., linearizability [40]), on the other hand, leads to more intuitive application behavior, although it requires client requests to be ordered across the system before they are executed by the servers [47, 77].

To confront this trade-off between consistency and availability, the system model needs to be changed. Partitioning comes as a solution to scale up in most scenarios while also keeping linearizability. The basic idea is quite straightforward: the data is distributed among groups of servers (partitions) in such a way that the correlated data is located in the same partition to maximize single-partition operations, thus increasing the level of parallelism and consequently the throughput. Besides, the data distribution strategy can also take into account clients' access patterns and place partitions close to those clients which use that specific data subset more often.

In such partially replicated systems, differently from SMR, the coordination is typically held by atomic multicast protocols [72]. Atomic multicast is chosen over atomic broadcast because it makes possible to deliver messages only to partition subsets. As a consequence, atomic multicast keeps a reduced latency for operations within a single partition and allows parallelism across partitions for global operations directed to disjoint subsets of servers.

With partial replication, however, replicas still need additional coordination to remain consistent [13]. The execution of a multi-partition operation demands synchronization among addressed partitions so that they can exchange data and update their state before executing the next ordered operation.

This thesis addresses the introduced challenges related to SMR and atomic multicast from different perspectives: it reduces latency for multi-group messages in atomic multicast, explores partial order in SMR, improves performance of consensus implementation, and proposes the first atomic multicast that tolerates Byzantine failures.

1.1 Contributions of the thesis

This work brings contributions in both crash (CFT) and Byzantine fault-tolerant (BFT) failure models. In the former, there are three different approaches to stretch the limits of both SMR and partitioned system (§1.1.1), while in the latter the first atomic multicast to support malicious behavior is introduced, with algorithm, proof of correctness and performance evaluation in both local-area and wide-area networks (§1.1.2).

1.1.1 Contributions in the crash-failure model

In sharded systems that rely on atomic multicast, the main issue is characterized by the imposition of high latency for messages addressed to multiple groups. To reduce the latency of those operations, a novel atomic multicast algorithm, dubbed FastCast, which can deliver a multi-partition operation in unprecedented four communication steps, is presented with algorithms and extensive performance evaluation. Single-partition operations are delivered in three communications steps, which matches the lower bound defined for atomic broadcast algorithms [50].

The proposed atomic multicast algorithm explores the increasing number of processors on recent computers and extends a traditional atomic multicast algorithm [15] executing an additional parallel optimistic ordering path. Under the assumptions presented in the next chapter, the original and the optimistic paths merge and order operations within only four communication delays.

The second contribution comes from the observation that operation ordering and execution can be decoupled [87]. Under this assumption, this thesis introduces GeoPaxos, a SMR protocol that scales similarly to partially replicated systems while keeping the data fully replicated, thus eliminating the necessity of additional data exchange and synchronization between partitions. While each replica keeps a full copy of all the information, the ordering responsibility is distributed among groups of servers. This distribution is performed in a way that maximizes the proximity between the clients and the data they access more often, avoiding the global latency of hundreds of milliseconds in geographically distributed scenarios.

GeoPaxos can also improve locality in such environments. The dynamics of client-server interactions can lead to unbalanced load and loss of locality, a situation where data needs to be redistributed. In GeoPaxos, such redistribution imposes no overhead on the infrastructure and no data migration. Once the data is fully replicated, only the ordering information needs to be updated.

The last mechanism to speed up replicated systems proposed in the thesis focuses on its core, the ordering protocol, typically represented by a consensus algorithm. Among the various consensus algorithms proposed, Paxos [48] stands out for its optimized resilience and communication. Much effort has been placed on implementing Paxos efficiently. Existing solutions make use of special network topologies, rely on specialized hardware, or exploit application semantics. Instead of proposing yet another variation of the original Paxos algorithm, this work proposes a new strategy to increase performance of Paxos-based state machine replication. The solution, dubbed Kernel Paxos, is an implementation of Paxos that significantly reduces communication overhead by avoiding system calls and TCP/IP stack. Besides, it reduces the number of context switches related to system calls, by providing Paxos as a kernel module.

1.1.2 Contribution in the Byzantine-failure model

Existing atomic multicast protocols only target benign failures. This section introduces ByzCast, the first Byzantine Fault-Tolerant (BFT) atomic multicast. Byzantine fault tolerance has become increasingly appealing as services can be deployed in inexpensive hardware (e.g., cloud environments) and new applications (e.g., blockchain [19]) become more sensitive to malicious behavior. ByzCast has two important characteristics: it was designed to use existing BFT abstractions and it scales with the number of groups, for messages addressed to a single group.

The main contributions of ByzCast are: (i) definition of a new class of atomic multicast algorithms, denominated partially genuine; (ii) construction on top of multiple instances of atomic broadcast as an overlay tree; (iii) presentation of the problem of building the overlay tree as an optimization problem; (iv) development and evaluation of a prototype using BFT-SMaRt [12].

1.2 System model and definitions

This section details the system model (§1.2.1) and recalls the definitions of consensus (§1.2.2), and reliable and atomic multicast (§1.2.3 and §1.2.4).

1.2.1 Processes, groups, and links

The system comprises $\Pi = \{p_1, \ldots, p_n\}$ server processes. Processes communicate by exchanging messages and do not have access to a shared memory or a global clock. The system is asynchronous: messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds.

The set $\Gamma = \{g_1, \ldots, g_m\}$ represents process groups in the system. Groups are disjoint, non-empty, and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. In this thesis, there are two failure models with different assumptions for processes and communication links.

Crash-failure model

In the CFT model, it is assumed that processes may fail by crashing (i.e., no malicious behavior). A process that never crashes is *correct*; otherwise it is *faulty*. Each group contains $2f+1$ processes, where f is the maximum number of faulty processes per group.

Communication links are fair-lossy, i.e., links do not create, corrupt, or duplicate messages, and guarantee that for any two *correct* processes p and q, and any message m, if p sends m to q infinitely many times, then q receives m an infinite number of times.

Byzantine-failure model

In the BFT model, processes can be *correct* or *faulty*. A correct process follows its specification whilst a faulty process can present arbitrary (i.e., Byzantine) behavior. Each group contains $3f+1$ processes, where f is the maximum number of faulty server processes per group [22, 51].

Cryptographic techniques are used for authentication, and digest calculation. Adversaries (and Byzantine processes under their control) are assumed computationally bound so that they are unable, with very high probability, to subvert the cryptographic techniques used. Adversaries can coordinate Byzantine processes and delay correct processes in order to cause the most damage to the system. Adversaries cannot, however, delay correct processes indefinitely.

1.2.2 Consensus

Inside each group g there exists a consensus service. The consensus service allows processes to propose values and ensures that eventually one of the proposed values is decided. A process in group g proposes a message (or a set of messages) x in instance i by invoking $\text{propose}_g[i](x)$, and decides on y in instance i with $\text{decide}_g[i](y)$. In the CFT model, the uniform consensus service satisfies the following properties:

- *uniform integrity:* if a process decides x in instance i, then x was previously proposed by some process in i.
- *termination:* if a correct process in group g proposes a value in instance i, then every correct process in g eventually decides exactly one value in i.
- *uniform agreement:* if a process in group g decides x in instance i, then no process in g decides $y \neq x$ in i.

In the Byzantine model, the consensus service properties cannot be *uniform* because a faulty server can decide arbitrary values:

- *integrity:* if a correct process decides x in instance i, then x was previously proposed by some process in i.
- *termination:* if a correct process in group g proposes a value in instance i, then every correct process in g eventually decides exactly one value in i.
- *agreement:* if a correct process in group g decides x in instance i, then no correct process in g decides $y \neq x$ in i.

To make consensus solvable in each group [34], it is further assumed that processes at each group have access to a weak leader election oracle [24]. The oracle outputs a single process denoted $leader_{g,p}$ such that there is (a) a correct process l_g in g and (b) a time after which, for every p in g $leader_{g,p} = l_g$.

1.2.3 Reliable and atomic multicast in crash-failure model

For every message m, *m.dst* denotes the groups to which m is multicast. If $|m.dst| = 1$ then m is a *local* message; if $|m.dst| > 1$ then m is *global*. A process reliably multicasts a message m by invoking primitive r-multicast(m) and delivers m with primitive r-deliver(m). This non-uniform FIFO reliable multicast ensures the following properties:

- *validity*: if a correct process p r-multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, r-deliver m.
- *non-uniform agreement*: if a correct process p r-delivers a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, r-deliver m.
- *integrity*: for any process p and any message m, p r-delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously r-multicast.
- *FIFO order*: if a process r-multicasts a message m before it r-multicasts a message m', then no process r-delivers m' unless it has previously r-delivered m.

With uniform atomic multicast, a process atomically multicasts message m using primitive a-multicast(m) and delivers m with a-deliver(m). The relation $<$ on the set of messages processes a-deliver is defined as follows: $m < m'$ iff there exists a process that a-delivers m before m'. Atomic multicast satisfies the uniform integrity and validity properties of reliable multicast as well as the following properties:

- *uniform agreement*: if a process p a-delivers a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.
- *uniform prefix order*: for any two messages m and m' and any two processes p and q such that $p \in g$, $q \in h$ and $\{g,h\} \subseteq m.dst \cap m'.dst$, 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'.
- *uniform acyclic order*: the relation $<$ is acyclic.

Atomic broadcast is a special case of atomic multicast in which every message is addressed to all groups.

To avoid contacting unnecessary processes, a reliable and atomic multicast protocols should ideally be *genuine* [37]: an algorithm $\mathscr{A}$ solving reliable or atomic multicast is genuine if and only if for any admissible run R of $\mathscr{A}$ and for any process p in R, if p sends or receives a message, then some message m is r-multicast (resp., a-multicast), and either (a) p is the process that r-multicasts (resp., a-multicasts) m or (b) $p \in g$ and $g \in m.dst$. In [37], the authors show the impossibility of solving genuine atomic multicast with weak synchronous assumptions (i.e., unreliable failure detectors [24]) when groups intersect. Hence, the assumption is that groups are disjoint.

1.2.4 Atomic multicast in Byzantine-failure model

The definitions of $m.dst$, local and global messages are the same provided in the previous section.

A process atomically multicasts a message m by invoking primitive a-multicast(m) and delivers m with a-deliver(m).

We define the relation $<$ on the set of messages correct processes a-deliver as follows: $m < m'$ iff there exists a correct process that a-delivers m before m'.

In BFT model, atomic multicast cannot be *uniform* and hence satisfies the following properties [39]:

- *Validity*: If a correct process p a-multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, a-deliver m.
- *Agreement*: If a correct process p a-delivers a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, a-deliver m.
- *Integrity*: For any correct process p and any message m, p a-delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously a-multicast.
- *Prefix order*: For any two messages m and m' and any two correct processes p and q such that $p \in g$, $q \in h$ and $\{g,h\} \subseteq m.dst \cap m'.dst$, 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'.

- *Acyclic order*: The relation $<$ is acyclic.

The definition of a genuine atomic multicast algorithm also applies to this failure model.

FIFO Atomic Broadcast

Atomic broadcast is a special case of atomic multicast in which there is a single group of server processes. In the BFT model, the assumption is that each group implements FIFO atomic broadcast, which in addition to the properties presented in §1.2.4, also ensures the following property.

- *FIFO order*: If a correct process broadcasts a message m before it broadcasts a message m', no correct process delivers m' unless it has previously delivered m.

Chapter 2

Making atomic multicast faster

Atomic multicast is a communication building block that allows messages to be propagated to groups of processes with reliability and order guarantees. Intuitively, all non-faulty processes addressed by a message must deliver the message and processes must agree on the order of delivered messages. Atomic multicast offers strong communication guarantees and should not be confused with network-level communication primitives (e.g., IP-multicast), which offer "best-effort" guarantees. Because messages can be multicast to different sets of destinations and interleave in non-obvious ways, implementing message order in a distributed setting is challenging. Some atomic multicast protocols address this challenge by ordering all messages using a fixed group of processes or involving all groups, regardless of the destination of the messages. To be efficient, however, an atomic multicast algorithm must be *genuine*: only the message sender and destination processes should communicate to propagate and order a multicast message [37]. A genuine atomic multicast is the foundation of scalable systems, since it does not depend on a fixed group of processes and does not involve all processes.

This chapter introduces FastCast, a genuine atomic multicast algorithm that offers unprecedented low time complexity, measured in communication delays. FastCast can order and deliver *global messages* (i.e., messages addressed to multiple groups of processes) in four communication delays; *local messages* (i.e., messages addressed to a single group of processes) take three delays. In comparison, three communication delays is a lower bound on atomic broadcast (in the presence of collisions) [50], a communication abstraction where there is a single group of processes.

FastCast is an optimistic algorithm inspired by BaseCast, an earlier genuine atomic multicast algorithm that requires six communication delays [15, 35, 73].

FastCast's secret sauce is to decompose the procedure used by BaseCast to order global messages in two execution paths. There is a *fast path* that speculates about the order of messages and a *slow path*, similar to BaseCast. If the fast path's "guess" is correct, something that can be assessed after four communication delays, the slow path is abbreviated; otherwise the slow path continues and computes the final order of the message. FastCast provides a significant advantage to existing atomic multicast algorithms since the fast path is correct in most common cases, that is, when the message sender and destination processes do not fail and are not suspected to have failed.

In addition to proposing a novel atomic multicast algorithm with reduced number of communication delays to order global messages, FastCast has been fully implemented and had its performance compared to BaseCast and a non-genuine atomic multicast protocol. Experiments were conducted in three environments: a local-area network (LAN), an emulated wide-area network (emulated WAN), and a real wide-area network (WAN). FastCast evaluation comprises a microbenchmark with configurations involving different number of groups, up to 16 groups, and a social network application deployed in 48 servers distributed in 16 groups of 3 servers each.

In brief, our results show that in WAN environments, FastCast outperforms BaseCast and the non-genuine atomic multicast protocol under a large variety of conditions, with two exceptions: when messages are multicast to a single group (all protocols perform similarly) and to all destinations (the non-genuine protocol performs better). In LAN environments, FastCast performs better than the two other protocols when messages are multicast to few destinations.

The rest of the chapter is organized as follows. Sections 2.1 and 2.2 present BaseCast and FastCast, respectively. Section 2.3 discusses the correctness of the presented algorithms. Section 2.4 describes our experimental evaluation. Section 2.5 surveys related work and Section 2.6 concludes the chapter.

2.1 Baseline Atomic Multicast

Fast Atomic Multicast, the main contribution of this chapter, is inspired by earlier atomic multicast protocols [15, 35, 73]. This section describes the fault-tolerant version of an early atomic multicast protocol [15] hereafter dubbed *BaseCast*. It initially provides an overview of BaseCast (§2.1.1), then describes it in detail (§2.1.2), and reasons about its time complexity (§2.1.3).

2.1.1 Overview

In BaseCast, each process implements a logical clock [47] and assigns timestamps to messages based on the logical clock. The correctness of BaseCast stems from two basic properties: (i) processes in the destination of an a-multicast message first assign tentative timestamps to the message and eventually agree on the message's final timestamp; and (ii) processes a-deliver messages according to their final timestamp.

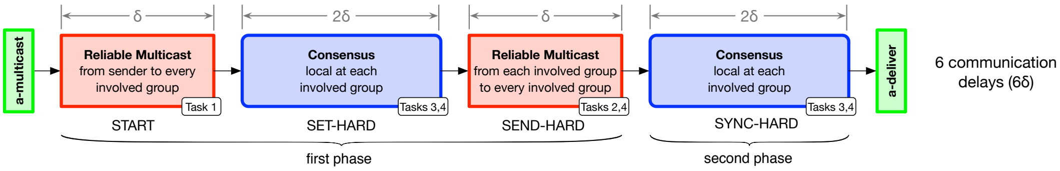
It is easier to understand how BaseCast guarantees properties (i) and (ii) by first considering the special case in which each group g_i in the system has a single and correct process p_i (i.e., $g_i = \{p_i\}$).

(i) To a-multicast a message m to a set of destinations (i.e., groups in $m.dst$), p_i r-multicasts m to the destinations in a START message. Upon r-delivering a START message with m, each destination updates its logical clock, assigns a hard tentative timestamp to m (the reason this timestamp is *hard* is explained in the next section), stores m and its timestamp in a buffer, and r-multicasts m's timestamp to all destinations in a SEND-HARD message. Upon r-delivering timestamps from all destinations in $m.dst$, a process p_i computes m's final timestamp as the maximum among all r-delivered hard tentative timestamps for m.

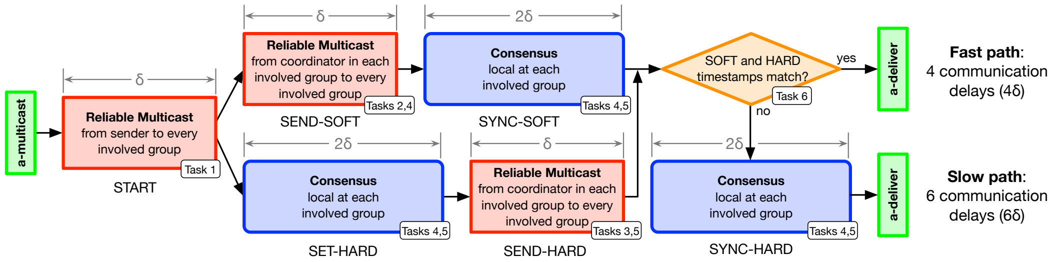
(ii) Messages are a-delivered respecting the order of their final timestamp. Thus, p_i a-delivers m when it can ascertain that m's final timestamp is smaller than the final timestamp of any messages p_i will a-deliver after m (intuitively, this holds because logical clocks are monotonically increasing).

To handle groups with any number of processes and thereby tolerate process failures, BaseCast uses consensus within each group to ensure that processes in the same group evolve through the same sequence of state changes and produce the same outputs [47, 77]. Consensus is needed within a group to order START messages (this consensus is called a SET-HARD step) and SEND-HARD messages (this consensus is called a SYNC-HARD step). Ordering SET-HARD and SYNC-HARD events within a group ensures that processes in the group assign the same hard tentative timestamp to an a-multicast message m and update their logical clock in the same deterministic way upon handling hard tentative timestamps from m's destination groups.

The propagation of the START message followed by the SET-HARD step and the propagation of SEND-HARD messages is referred as the *first phase* of the al-


δ
2δ
δ
2δ
a-multicast
Reliable Multicast
from sender to every involved group
Task 1
START
Consensus
local at each involved group
Tasks 3,4
SET-HARD
Reliable Multicast
from each involved group to every involved group
Tasks 2,4
SEND-HARD
Consensus
local at each involved group
Tasks 3,4
SYNC-HARD
a-deliver
6 communication delays (6δ)
first phase
second phase

(a) BaseCast


a-multicast
δ
Reliable Multicast
from sender to every involved group
Task 1
START
δ
Reliable Multicast
from coordinator in each involved group to every involved group
Tasks 2,4
SEND-SOFT
2δ
Consensus
local at each involved group
Tasks 4,5
SYNC-SOFT
SOFT and HARD timestamps match?
Task 6
yes
no
a-deliver
Fast path:
4 communication delays (4δ)
2δ
Consensus
local at each involved group
Tasks 4,5
SET-HARD
δ
Reliable Multicast
from coordinator in each involved group to every involved group
Tasks 3,5
SEND-HARD
2δ
Consensus
local at each involved group
Tasks 4,5
SYNC-HARD
a-deliver
Slow path:
6 communication delays (6δ)

(b) FastCast

Figure 2.1. Diagrammatic representations of BaseCast (top) and FastCast (bottom). Existing reliable multicast and consensus algorithms can propagate messages in one communication delay and reach a decision in two communication delays, respectively (e.g., [24] and [48]).

gorithm, and to the SYNC-HARD step as the *second phase* of the algorithm (see Figure 2.1(a)).

2.1.2 Detailed algorithm

Algorithm 1 contains five tasks that execute in isolation. The algorithm contains six variables: C_H implements a process's logical clock, used to assign hard tentative timestamps to messages; $\mathscr{B}$ contains timestamps assigned to messages not yet a-delivered; k_p and k_d index consensus instances; and sequences *ToOrder* and *Ordered* are used to totally order messages among processes in a group. Let S and R be two sequences. $S \oplus R$ denotes S followed by R and $S \setminus R$ denotes S without the entries that exist in R.

To a-multicast a message m, process p in group g r-multicasts m to m's destinations using a $(\text{START}, \bot, \bot, m)$ tuple. The information in such tuple represents, respectively, the type of the message, the group that r-multicasts the message, the timestamp and the message m. When p r-delivers message $(\text{START}, \bot, \bot, m)$ in Task 1 (respectively, $(\text{SEND-HARD}, h, x, m)$ in Task 2), p adds $(\text{SET-HARD}, g, \bot, m)$ (respectively, $(\text{SYNC-HARD}, h, x, m)$) to *ToOrder* to be ordered by consensus in Tasks 3 and 4. Consensus instances are independent and can execute concurrently. However, decision events are handled sequentially according to the order determined by k_d in Task 4.

Process p handles a $(\textsc{set-hard}, \perp, \perp, m)$ tuple in Task 4 by choosing a tentative hard timestamp for m, given by C_H, and propagating the chosen timestamp to m's destinations using a $(\textsc{send-hard}, g, C_H, m)$ message, if m is global; if m is local, p adds the chosen timestamp to $\mathscr{B}$ as a $(\textsc{sync-hard}, g, C_H, m)$ tuple. Process p handles a $(\textsc{sync-hard}, h, x, m)$ tuple by updating its local clock C_H and including the tuple in $\mathscr{B}$.

In Task 5, if p has received tentative timestamps from all groups in m's destinations (i.e., SYNC-HARD tuples in $\mathscr{B}$), p determines m's *final* timestamp as the maximum timestamp among the tentative timestamps assigned to m by m's destinations. Process p a-delivers m when it ascertains that no a-multicast message m' will have a final timestamp smaller than m's. This happens when no a-multicast message m' r-delivered by p has (a) a final timestamp smaller than m's and (b) a tentative timestamp smaller than m's final timestamp, if p has not received tentative timestamps from all of m''s destinations yet.

2.1.3 Time complexity

This section states the best-case time complexity between a-multicast(m) and a-deliver(m) in Algorithm 1 in terms of δ, the maximum communication delay.

Proposition 1 *Let Algorithm 1 use the reliable multicast implementation in [24] and the consensus implementation in [48]. Every atomically multicast global message is delivered in at least* 6δ.

2.2 Fast Atomic Multicast

This section introduces *FastCast*, a genuine atomic multicast algorithm that can deliver global messages *fast*, in four communication delays, providing a general description of FastCast (§2.2.1), then presenting it in detail (§2.2.2) and discussing its time complexity (§2.2.3).

2.2.1 Overview

Fast Atomic Multicast implements two execution paths for global messages, a *fast path* and a *slow path*. The two paths execute concurrently and are triggered after a message m is a-multicast (see Figure 2.1(b)).

Algorithm 1 Baseline Atomic Multicast - BaseCast (for process p in group g)

```
Initialization
    $C_H \leftarrow 0$    {p's logical clock}
    $\mathcal{B} \leftarrow \emptyset$    {tentative timestamps of undelivered messages}
    $k_p \leftarrow 0$; $k_d \leftarrow 0$    {the consensus instance for propose and decide events, respectively}
    $ToOrder \leftarrow \epsilon$; $Ordered \leftarrow \epsilon$    {sequences of tuples to be and already ordered, respectively}

To a-multicast message $m$:
    r-multicast $(\textsc{start}, \perp, \perp, m)$ to $m.dst$

when r-deliver$(\textsc{start}, \perp, \perp, m)$    {Task 1}
    $ToOrder \leftarrow ToOrder \oplus (\textsc{set-hard}, g, \perp, m)$    {add a tuple to be ordered by consensus}

when r-deliver$(\textsc{send-hard}, h, x, m)$    {Task 2}
    if $\forall y : (\textsc{sync-hard}, h, y, m) \notin ToOrder$ then    {if this tuple not already included...}
        $ToOrder \leftarrow ToOrder \oplus (\textsc{sync-hard}, h, x, m)$    {...add it to be ordered}

when $ToOrder \setminus Ordered \neq \emptyset$    {Task 3}
    propose$_g[k_p](ToOrder \setminus Ordered)$    {propose all tuples that haven't been ordered yet}
    $k_p \leftarrow k_p + 1$    {adjust counter for next propose instance}

when decide$_g[k_d](Decided)$    {Task 4}
    for each $z, h, x, m : (z, h, x, m) \in Decided \setminus Ordered$ in order do    {for each tuple}
        if $z = \textsc{set-hard}$ then    {set g's hard tentative timestamp}
            $C_H \leftarrow C_H + 1$    {increment logical clock}
            if $|m.dst| > 1$ then    {if m is a global message:}
                $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\textsc{send-hard}, g, C_H, m)\}$    {store g's hard tentative timestamp and...}
                r-multicast $(\textsc{send-hard}, g, C_H, m)$ to $m.dst$    {...send it to each m's destination}
            else    {if m is a local message:}
                $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\textsc{sync-hard}, g, C_H, m)\}$    {g's hard tentative timestamp is ordered}
        if $z = \textsc{sync-hard}$ then    {handle the receipt of h's hard tentative timestamp}
            $C_H \leftarrow max(\{C_H, x\})$    {Lamport's rule to update logical clocks}
            $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(\textsc{send-hard}, h, x, m)\}$    {no longer needed, will store ordered timestamp}
            $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\textsc{sync-hard}, h, x, m)\}$    {h's hard tentative timestamp is now ordered}
        $Ordered \leftarrow Ordered \oplus (z, h, x, m)$    {this tuple has been handled}
    $k_d \leftarrow k_d + 1$    {adjust counter for next decide instance}

when $\exists m\ \forall h \in m.dst\ \exists x : (\textsc{sync-hard}, h, x, m) \in \mathcal{B}$    {Task 5}
    $ts \leftarrow max(\{x : (\textsc{sync-hard}, h, x, m) \in \mathcal{B}\})$    {ts is m's final timestamp}
    for each $z, h, x : (z, h, x, m) \in \mathcal{B}$ do $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(z, h, x, m)\}$    {replace m's ...}
    $\mathcal{B} \leftarrow \mathcal{B} \cup \{(\textsc{final}, \perp, ts, m)\}$    {...tentative timestamps in B by m's final timestamp}
    while $\exists(\textsc{final}, \perp, ts, m) \in \mathcal{B}$ :
          $\forall(z, h, x, m') \in \mathcal{B}, m \neq m' : ts < x$ do    {for each deliverable message}
        a-deliver $m$    {deliver it!}
        $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(\textsc{final}, \perp, ts, m)\}$    {B only contains undelivered messages}
```

In the fast path, processes in each group $g \in m.dst$ assign a soft tentative timestamp to m, which is a "guess" of what g's hard tentative timestamp will be. It is called a guess because g's hard tentative timestamp is determined after processes in g execute consensus, in a SET-HARD step. Processes r-multicast m's soft tentative timestamp to destinations in $m.dst$ in a SEND-SOFT message. Upon r-delivering soft tentative timestamps, processes in g execute consensus to ensure that they act in the same way. This consensus is called a SYNC-SOFT step.

Processes in g can a-deliver m fast if two conditions are satisfied: (a) processes have r-delivered and ordered (through consensus) soft tentative timestamps from all groups in $m.dst$, and therefore they can compute m's final timestamp; and (b) the soft tentative timestamp of each $g \in m.dst$ matches g's hard tentative timestamp (determined in the slow path).

The slow path resembles BaseCast, with the exception that processes may skip the second phase of the protocol. When processes in g have collected soft tentative timestamps, as a result of the fast path, and hard tentative timestamps, as a result of the first phase of the slow path, from all groups in $m.dst$, they compare the timestamps. If the soft and hard tentative timestamps for each group in $m.dst$ match, then processes can a-deliver m without executing the second phase of the slow path. If the timestamps do not match for some group, processes continue the execution of the second phase of the slow path, just like in BaseCast.

The procedure is modified above so that processes in a group can make "educated guesses" of their hard tentative timestamps, and thereby a-deliver messages fast.

- In each group g, consensus is implemented with Paxos [48]. In each Paxos instance i, the leader in g proposes the value to be decided in instance i. In runs in which there is a single and correct leader in g, processes will decide on the value proposed by the leader [48].
- The leader is the only process in a group to propagate soft tentative timestamps (SEND-SOFT messages) to the destination groups of a message and uses the chosen timestamp as the proposed hard tentative timestamp in consensus (SET-HARD step).

As a result, in executions in which there is a single and correct leader in each group, a-multicast messages will be a-delivered fast.

2.2.2 Detailed algorithm

Algorithm 2 presents seven tasks that execute in isolation. In addition to the variables used in Algorithm 1, it also implements a logical clock C_S used to assign soft tentative timestamps to messages. In the following, execution of global messages is first described and then the execution of local messages.

A process a-multicasts global message m by r-multicasting $(\textsc{start}, \bot, \bot, m)$ to m's destinations (i.e., groups in $m.dst$). In Task 1, process p in group g r-delivers the START message and requests processes in g to compute a hard tentative timestamp for m, by adding $(\textsc{set-hard}, g, \bot, m)$ to sequence *ToOrder*. Entries in *ToOrder* are ordered by consensus in Tasks 4 and 5. In Task 4, if p is the leader of g, it proposes tuples in *ToOrder* in the next available consensus instance; p also chooses a soft tentative timestamp for m and r-multicasts this timestamp in tuple $(\textsc{send-soft}, h, C_S, m)$ to m's destinations.

Soft tentative timestamps are r-delivered in Task 2, where p adds $(\textsc{sync-soft}, h, x, m)$ to *ToOrder* to be ordered by consensus in Task 4. Task 5 is responsible for consensus decisions, handling them sequentially and in order. To handle a $(\textsc{set-hard}, h, x, m)$ tuple, p updates its logical clock C_H, picks a hard tentative timestamp for m and r-multicasts this timestamp to m's destinations in a SEND-HARD message. Finally, p handles $(\textsc{sync-hard}, h, x, m)$ and $(\textsc{sync-soft}, h, x, m)$ tuples by updating its logical clock C_H and adding the tuples to $\mathscr{B}$.

A tuple $(\textsc{send-hard}, h, x, m)$, with m's hard tentative timestamp from group h, is r-delivered in Task 3. As a result, p adds $(\textsc{sync-hard}, h, x, m)$ to *ToOrder* so that it is ordered by consensus, as described above. Task 6 checks whether the soft and hard tentative timestamps for m match, in which case it sets m as ordered and places m in $\mathscr{B}$, where it will be eventually a-delivered. The condition for a-delivering m is similar in Algorithm 1 (Task 5) and Algorithm 2 (Task 7).

Local a-multicast messages experience a shorter execution path than global messages. When a message m is a-multicast to a single group g, it is r-multicast to the members of g in a $(\textsc{start}, \bot, \bot, m)$ message. Upon r-delivering m, g's members execute a round of consensus (SET-HARD step) to agree on the hard tentative timestamp to assign to m and ensure that future messages will be assigned a timestamp greater than the one assigned to m. When the proposed timestamp is decided, it becomes m's final timestamp.

2.2.3 Time complexity

Proposition 2 states that Algorithm 2 can deliver messages fast, in four communication delays.

Proposition 2 *Assume Algorithm 2 uses the reliable multicast implementation in [24] and the consensus implementation in [48]. If $\mathscr{R}$ is a set of runs of Algorithm 2*

Algorithm 2 Fast Atomic Multicast - FastCast (for process p in group g)

```
Initialization
    $C_H \leftarrow 0$; $C_S \leftarrow 0$    {p's logical clock for hard and soft tentative timestamps, respectively}
    $\mathscr{B} \leftarrow \emptyset$    {tentative timestamps of undelivered messages}
    $k_p \leftarrow 0$; $k_d \leftarrow 0$    {the consensus instance for propose and decide events, respectively}
    $ToOrder \leftarrow \epsilon$; $Ordered \leftarrow \epsilon$    {sequences of tuples to be and already ordered, respectively}

To a-multicast message $m$:
    r-multicast $(\text{START}, \bot, \bot, m)$ to $m.dst$

when r-deliver$(\text{START}, \bot, \bot, m)$    {Task 1}
    $ToOrder \leftarrow ToOrder \oplus (\text{SET-HARD}, g, \bot, m)$    {add a tuple to be ordered by consensus}

when r-deliver$(\text{SEND-SOFT}, h, x, m)$    {Task 2}
    if $\forall y : (\text{SYNC-SOFT}, h, y, m) \notin ToOrder$ then    {if this tuple not already included...}
        $ToOrder \leftarrow ToOrder \oplus (\text{SYNC-SOFT}, h, x, m)$    {...add it to be ordered}

when r-deliver$(\text{SEND-HARD}, h, x, m)$    {Task 3}
    if $\forall y : (\text{SYNC-HARD}, h, y, m) \notin ToOrder$ then    {if this tuple hasn't been included already...}
        $ToOrder \leftarrow ToOrder \oplus (\text{SYNC-HARD}, h, x, m)$    {...add it to be ordered}

when $ToOrder \setminus Ordered \neq \emptyset$ and $leader_{g,p} = p$    {Task 4}
    $C_S \leftarrow \max(\{C_H, C_S\})$    {soft timestamp should not be smaller than hard timestamp}
    for each $(z, h, x, m) \in ToOrder \setminus Ordered$ in delivery order do    {for unordered msgs}
        if z = SET-HARD then
            $C_S \leftarrow C_S + 1$    {increment logical clock and propagate soft tentative timestamp}
            if $|m.dst| > 1$ then r-multicast $(\text{SEND-SOFT}, h, C_S, m)$ to $m.dst$
        else
            $C_S \leftarrow \max(\{C_S, x\})$    {soft timestamp should not be smaller than unordered ones}
    propose$_g[k_p](ToOrder \setminus Ordered)$    {propose all tuples that haven't been ordered yet}
    $k_p \leftarrow k_p + 1$    {adjust counter for next propose instance}

when decide$_g[k_d](Decided)$    {Task 5}
    for each $(z, h, x, m) \in Decided \setminus Ordered$ in order do    {for each ordered tuple}
        if $z = \text{SET-HARD}$ then    {set g's hard tentative timestamp}
            $C_H \leftarrow C_H + 1$    {increment logical clock and send hard tentative timestamp}
            if $|m.dst| > 1$ then r-multicast $(\text{SEND-HARD}, g, C_H, m)$ to $m.dst$
            else $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SYNC-HARD}, g, C_H, m)\}$    {if m is local, hard timestamp is ordered}
        if $z = \text{SYNC-SOFT}$ then
            $C_H \leftarrow \max(\{C_H, x\})$    {Lamport's rule to update logical clocks}
            $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SYNC-SOFT}, h, x, m)\}$    {h's soft tentative timestamp is now ordered}
        if $z = \text{SYNC-HARD}$ then
            $C_H \leftarrow \max(\{C_H, x\})$    {Lamport's rule to update logical clocks}
            $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SYNC-HARD}, h, x, m)\}$    {h's hard tentative timestamp is now ordered}
        $Ordered \leftarrow Ordered \oplus \{(z, h, x, m)\}$    {this tuple has been handled}
    $k_d \leftarrow k_d + 1$    {adjust counter for next decide instance}

when $\exists h, x, m : (\text{SYNC-SOFT}, h, x, m) \in \mathscr{B}$ and
                    $(\text{SYNC-HARD}, h, x, m) \in ToOrder \setminus Ordered$    {Task 6}
    $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{SYNC-HARD}, h, x, m)\}$    {h's hard tentative timestamp is now ordered}
    $Ordered \leftarrow Ordered \oplus \{(\text{SYNC-HARD}, h, x, m)\}$    {this tuple has been handled}

when $\exists m \; \forall h \in m.dst \; \exists x : (\text{SYNC-HARD}, h, x, m) \in \mathscr{B}$    {Task 7}
    $ts \leftarrow max(\{x : (\text{SYNC-HARD}, h, x, m) \in \mathscr{B}\})$    {ts is m's final timestamp}
    for each $z, h, x : (z, h, x, m) \in \mathscr{B}$ do $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(z, h, x, m)\}$    {replace m's...}
    $\mathscr{B} \leftarrow \mathscr{B} \cup \{(\text{FINAL}, \bot, ts, m)\}$    {...tentative timestamps in $\mathscr{B}$ by m's final timestamp}
    while $\exists(\text{FINAL}, \bot, ts, m) \in \mathscr{B} \; \forall(z, h, x, m') \in \mathscr{B}, m \neq m' : ts < x$ do
        a-deliver $m$    {deliver each deliverable m in $\mathscr{B}$ }
        $\mathscr{B} \leftarrow \mathscr{B} \setminus \{(\text{FINAL}, \bot, ts, m)\}$    {$\mathscr{B}$ only contains undelivered messages}
```

in which for every group g there is a correct leader l_g *and for each* $p \in g$ $leader_{g,p} = l_g$*, then there are runs in* $\mathscr{R}$ *in which atomically multicast global messages are delivered fast, in four communication delays.*

2.3 Proofs of Correctness

This sections details the proofs of correctness for the presented algorithms.

2.3.1 Proofs for Propositions 1 and 2

Proposition 1 *Let Algorithm 1 use the reliable multicast implementation in [24] and the consensus implementation in [48]. Every atomically multicast global message is delivered in at least* 6δ*.*

PROOF: Let m be a global message a-multicast using Algorithm 1. It takes one communication delay for m to be r-multicast to its destinations. Each group in $m.dst$ relies on consensus to assign m a hard tentative timestamp. Executing an instance of consensus within m's destination groups takes 2δ in the best case [50]. Processes then exchange their timestamps in another communication delay and finally execute one more instance of consensus before a-delivering m. Thus, m can be a-delivered in (at least) 6δ. □

Proposition 2 *Assume Algorithm 2 uses the reliable multicast implementation in [24] and the consensus implementation in [48]. If* $\mathscr{R}$ *is a set of runs of Algorithm 2 in which for every group g there is a correct leader* l_g *and for each* $p \in g$ $leader_{g,p} = l_g$*, then there are runs in* $\mathscr{R}$ *in which atomically multicast global messages are delivered fast, in four communication delays.*

PROOF: Consider run $R \in \mathscr{R}$ where a process r a-multicasts a global message m. From Algorithm 2, r r-multicasts $(\text{START}, \bot, \bot, m)$ to $m.dst$. For every g in $m.dst$ and each p in g, p r-delivers m at Task 1 within one communication delay and adds tuple $(\text{SET-HARD}, g, \bot, m)$ to *ToOrder*. The following part consider now what happens in the fast and in the slow paths.

- In the fast path, from the hypothesis and Task 4, for each g in $m.dst$, process l_g (and no other process in g) assigns a soft timestamp C_s to m and r-multicasts it to $m.dst$ in a SEND-SOFT message, which processes r-deliver after one delay (Task 2) and as a result include tuple $(\text{SYNC-SOFT}, h, y, m)$ in *ToOrder*. Then, l_g in g proposes a sequence that contains the SYNC-SOFT message and after two delays, processes decide on the value proposed and include $(\textsc{sync-soft}, h, y, m)$ in $\mathscr{B}$. In total, four communication delays are needed since m is a-multicast.

- In the slow path, after including $(\textsc{set-hard}, g, \perp, m)$ in *ToOrder*, l_g (and no other process in g) proposes a sequence that includes the SET-HARD tuple, which is decided by processes in g after two delays. After deciding on the SET-HARD tuple, processes in g assign hard timestamp C_h to m and r-multicast $(\textsc{send-hard}, g, C_h, m)$ to all processes in $m.dst$ (Task 5). This message is r-delivered within one delay and upon r-delivering it (Task 3), processes include $(\textsc{set-hard}, g, C_h, m)$ in $\mathscr{B}$. In total, four communication delays since m is a-multicast.

Thus, for each process in m's destination and each group $g \in m.dst$, after four delays tuple $(\textsc{sync-soft}, g, x, m)$ is in $\mathscr{B}$ and tuple $(\textsc{sync-hard}, g, x, m)$ is in *ToOrder* and p includes $(\textsc{sync-hard}, g, x, m)$ in $\mathscr{B}$ (Task 6) and a-delivers m (Task 7), after four communication delays. □

2.3.2 Proof of correctness for Algorithm 1

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

PROOF: From p to a-deliver m (Task 5), p assessed that for each h in $m.dst$, there is a tuple $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$. It is shown that SYNC-HARD tuples for m are included only once in $\mathscr{B}$. There are two cases in which tuple $(\textsc{sync-hard}, h, x, m)$ is included in $\mathscr{B}$. In Task 4, $(\textsc{sync-hard}, h, x, m)$ is included in $\mathscr{B}$ upon deciding on tuple $(\textsc{set-hard}, h, x, m)$ if m is local and upon deciding on $(\textsc{sync-hard}, h, x, m)$ if m is global. For the first case to happen, $(\textsc{set-hard}, h, x, m)$ must be included in *ToOrder* in Task 1, as the result of the r-delivery of $(\textsc{start}, \perp, \perp, m)$. From the properties of reliable broadcast, such a message is delivered only once by p. The second case happens when a SYNC-HARD tuple is in *ToOrder*, which from Task 2, it is included only once. From Algorithm 2, it follows immediately that p only a-delivers m if p is part of m's addresses and m is a-multicast. □

Lemma 1 *If a correct process p in g includes tuple T in ToOrder, then eventually processes in g decide on a sequence of tuples that contains T.*

PROOF: Process p includes T in *ToOrder* either in Task 1 or in Task 2. In both cases, T was r-delivered by p and from the properties of reliable broadcast, every correct process in g will r-deliver T and include it in *ToOrder*. Let t be a time after which all faulty processes have failed. Thus, after t there is a time when all *ToOrder* sequences from processes that propose in consensus contain T. By the uniform integrity property of consensus, T is eventually included in a decision of consensus. □

Lemma 2 *For each correct process p that has tuple* $(\text{SYNC-HARD}, h, x, m)$ *in* $\mathscr{B}$*, p eventually replaces the entry by* $(\text{FINAL}, \perp, ts, m)$ *in* $\mathscr{B}$ *where ts is the maximum timestamp x in the* SYNC-HARD *tuples that concern m.*

PROOF: To include $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$, p has decided on a sequence that contains either (a) a $(\text{SET-HARD}, h, x, m)$ tuple if m is local, or (b) a $(\text{SYNC-HARD}, h, x, m)$ tuple if m is global. In case (a), $(\text{SYNC-HARD}, h, x, m)$ will be trivially replaced by $(\text{FINAL}, \perp, x, m)$ in Task 5. In case (b), some process proposed a *ToOrder* \ *Ordered* sequence that contains $(\text{SYNC-HARD}, h, x, m)$. The SYNC-HARD tuple is included in *ToOrder* in Task 2 upon r-delivering tuple $(\text{SEND-HARD}, h, x, m)$, which was r-multicast in Task 4, upon the decision of a sequence with $(\text{SET-HARD}, h, x, m)$. Thus, $(\text{SET-HARD}, h, x, m)$ was included in *ToOrder* at Task 1, as a result of the r-delivery of $(\text{START}, \perp, \perp, m)$, which is r-multicast to all of m's destinations. Every group h in $m.dst$ upon r-delivering $(\text{START}, \perp, \perp, m)$ adds tuple $(\text{SET-HARD}, h, x, m)$ to *ToOrder*, which from Lemma 1 is eventually included in a consensus decision and results in the r-multicast of $(\text{SEND-HARD}, h, x, m)$ to members of $m.dst$. When a process r-delivers $(\text{SEND-HARD}, h, x, m)$, it adds $(\text{SYNC-HARD}, h, x, m)$ to *ToOrder* and, from Lemma 1, the tuple is decided in an instance of consensus, leading to the inclusion of $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. Once there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$ for each group h in $m.dst$, p replaces the SYNC-HARD tuples by $(\text{FINAL}, \perp, ts, m)$. □

Lemma 3 *If a correct process p includes* $(\text{FINAL}, \perp, ts, m)$ *in* $\mathscr{B}$*, then p eventually a-delivers m.*

PROOF: Assume for a contradiction that q does not a-deliver m. Thus, there is some tuple (z, h, y, m') in $\mathscr{B}$ such that $m \neq m'$ and $y < ts$. It is first shown that eventually any entry (z, h, y, m') added in $\mathscr{B}$ after $(\text{FINAL}, \perp, ts, m)$ is in $\mathscr{B}$ has a timestamp larger than ts. Message m only has a FINAL tuple in $\mathscr{B}$ after it received SYNC-HARD tuples from all of m's destinations. When q includes $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$ in Task 4, q updates C_h such that it contains the maximum between it current value and x. Since the next SET-HARD event that q

handles for a message m'' will increment C_h, it follows that m'' will have a timestamp larger than ts.

It is then shown that every message that contains a timestamp smaller than m's final timestamp ts is eventually a-delivered and removed from $\mathscr{B}$. Let (z, h, y, m') be an entry in $\mathscr{B}$ such that $y < ts$. Either z is FINAL or it is SYNC-HARD and from Lemma 2 z the tuple will eventually be replaced by a FINAL tuple. Thus, from Task 5 message m' will be eventually a-delivered and removed from $\mathscr{B}$, a contradiction. The conclusion is that q eventually a-delivers m. ☐

Proposition 4 *Validity: If a correct process p a-multicasts a message m, then eventually all correct processes q ∈ m.dst a-deliver m.*

PROOF: Upon a-multicasting m, p r-multicasts $(\text{START}, \perp, \perp, m)$ to all processes in $m.dst$ and from the validity and agreement properties of reliable broadcast, every correct q in $m.dst$ will r-deliver $(\text{START}, \perp, \perp, m)$ in Task 1. By Task 1, q includes $(\text{SET-HARD}, g, \perp, m)$ in *ToOrder* and from Lemma 1, the tuple is eventually decided in some instance of consensus. If m is global, q r-multicasts $(\text{SEND-HARD}, g, \perp, m)$ to all processes in $m.dst$; if m is local, q includes $(\text{SYNC-HARD}, g, \perp, m)$ in $\mathscr{B}$. In the first case, every correct process $r \in m.dst$ eventually r-delivers $(\text{SEND-HARD}, g, \perp, m)$ and includes $(\text{SYNC-HARD}, g, \perp, m)$ in *ToOrder*. From Lemma 1 the SYNC-HARD tuple is eventually included in a consensus decision and from Task 4 in $\mathscr{B}$. Therefore, eventually SYNC-HARD tuples from every group in $m.dst$ are in $\mathscr{B}$ and q replaces them by $(\text{FINAL}, \perp, ts, m)$, where ts is the maximum among the timestamps in the SYNC-HARD tuples. From Lemma 3, q eventually a-delivers m. ☐

Proposition 5 *Uniform agreement: If a process p a-delivers a message m, then eventually all correct processes q ∈ m.dst a-deliver m.*

PROOF: For process p to a-deliver m, from Task 5 for every group h in $m.dst$, there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. Moreover, there is no entry (z, h', y, m') in $\mathscr{B}$ such that $m \neq m'$ and $ts < y$, where ts is the maximum timestamp in the SYNC-HARD tuples that concern m. There are two cases to consider.

Case (a): p and q are in the same group g. Process p adds tuples to $\mathscr{B}$ in Task 4 only, and every tuple added to $\mathscr{B}$, which can be of type SET-HARD and SYNC-HARD, is created from an entry in *Decided* and variable C_h. Moreover, C_h is modified only in Task 4, based on tuples in *Decided*. Since p and q decide on the same sequence of consensus values, it follows that unless q fails, it executes the same sequence of steps as p and eventually a-delivers m.

Case (b): p and q are in different groups. To include (SYNC-HARD, h, x, m) in $\mathscr{B}$, p has decided on a sequence that includes the tuple, for each group $h \in m.dst$. From consensus, some process r proposed a sequence that contains (SYNC-HARD, h, x, m) in *ToOrder* (Task 3). So, r r-delivered tuple (SEND-HARD, h, x, m), which was r-multicast to all of m's destinations in Task 4. As a consequence, processes in q's group r-deliver message (SEND-HARD, h, x, m) and decide on a sequence that includes (SYNC-HARD, h, x, m). The conclusion is that r adds (SYNC-HARD, h, x, m) to $\mathscr{B}$, for each of m's destinations h. From Lemma 3, q eventually a-delivers m. □

Proposition 6 *Uniform prefix order: For any two messages m and m' and any two processes p and q such that $\{p, q\} \subseteq m.dst \cap m'.dst$, 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: The proposition trivially holds if p and q are in the same group, so assume p is in group g and q is in group h and suppose, by way of contradiction, that p does not a-deliver m' before m nor does q a-deliver m before m'. Without loss of generality, suppose that $m.ts < m'.ts$.

A initial claim is that q inserts m into $\mathscr{B}$ before a-delivering m'. In order for m (respectively, m') to be a-delivered by p (resp., q), p's (resp., q's) $\mathscr{B}$ must contain tuples (SYNC-HARD, g, x, m) and (SYNC-HARD, h, y, m) (resp., (SYNC-HARD, g, x', m') and (SYNC-HARD, h, y', m')). From Task 4, for p to include a SYNC-HARD tuple in $\mathscr{B}$, p must have decided a sequence that contains (SYNC-HARD, g, x, m) (recall that m is a global message). Thus, some process in g included (SYNC-HARD, g, x, m) in *ToOrder*, after r-delivering tuple (SEND-HARD, g, x, m). With a similar argument, some process in g included (SYNC-HARD, g, x', m') in *ToOrder*, after r-delivering tuple (SEND-HARD, g, x', m'). Let r and s be the processes that r-multicast messages (SEND-HARD, g, x, m) and (SEND-HARD, g, x', m'), respectively, at Task 4. Therefore, r and s decided sequences that include the SET-HARD tuples. Assume that (SET-HARD, g, x, m) is decided before (SET-HARD, g, x', m'). Therefore, before r-multicasting (SEND-HARD, g, x', m'), s r-multicast (SEND-HARD, g, x, m). From the FIFO properties of reliable multicast, q r-delivered the tuples in the order above and it can be shown that (SYNC-HARD, g, x, m) appears in $\mathscr{B}$ before (SET-HARD, g, x', m'), which proves our claim.

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

Proposition 7 *Uniform acyclic order: The relation $<$ is acyclic.*

PROOF: Suppose, by way of contradiction, that there exist messages $m_1, ..., m_k$ such that $m_1 < m_2 < ... < m_k < m_1$. From Task 5, processes a-deliver messages following the order of their final timestamps. Thus, there must be processes p and q such that the final timestamps they assign to m_1, $m_1.ts_p$ and $m_1.ts_q$, satisfy $m_1.ts_p < m_1.ts_q$, a contradiction since both p and q receive the same SYNC-HARD tuples used to calculate m_1's final timestamp in Task 5.

□

Theorem 1 *Algorithm 1 implements atomic multicast.*

PROOF: This follows directly from Propositions 3 through 7. □

2.3.3 Proof of correctness for Algorithm 2

Proposition 8 *Uniform integrity: For any process p and any message m, p a-delivers m at most once, and only if $p \in m.dst$ and m was previously a-multicast.*

PROOF: Assume p a-delivers m. From Task 7, p assessed that for each h in $m.dst$, there is a tuple $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$. It is shown that SYNC-HARD tuples for m are included only once in $\mathscr{B}$. Since SYNC-HARD tuples for m are removed from $\mathscr{B}$ when m is a-delivered (in Task 7), this shows that m is a-delivered only once. There are three cases in which tuple $(\text{SYNC-HARD}, h, x, m)$ is included in $\mathscr{B}$ in Algorithm 2. In Task 5, $(\text{SYNC-HARD}, h, x, m)$ is included in $\mathscr{B}$ (a) upon deciding on tuple $(\text{SET-HARD}, h, x, m)$ if m is local and (b) upon deciding on $(\text{SYNC-HARD}, h, x, m)$ if m is global. (c) In Task 6 if the soft and hard timestamps for h (i.e., $(\text{SYNC-SOFT}, h, x, m)$ and $(\text{SYNC-HARD}, h, x, m)$) in *ToOrder* match.

In all cases, after including $(\text{SYNC-HARD}, h, x, m)$ in $\mathscr{B}$, p added the tuple to *Ordered* and so, the tuple will not be in any future consensus decision. Moreover, in case (a), $(\text{SET-HARD}, h, x, m)$ is included in *ToOrder* upon r-delivering a START message for m (Task 1). From uniform integrity of reliable multicast, p does not r-delivers the START message for m more than once. In cases (b) and (c), since the SYNC-HARD tuple is in *ToOrder*, from Task 3, no other SYNC-HARD tuple for m from h will be included in *ToOrder*.

From Algorithm 2, it follows immediately that p only a-delivers m if p is part of m's addresses and m is a-multicast. □

Lemma 4 *If a correct process p in g includes tuple T in ToOrder, then eventually processes in g decide on a sequence of tuples that contains T.*

PROOF: Process p includes T in *ToOrder* in Tasks 1, 2 and 3. In all cases, T was r-delivered by p and from the properties of reliable broadcast, every correct process in g will r-deliver T and include it in *ToOrder*. Let t be a time after which all faulty processes have failed and t' the time after which for every process p in g, $leader_{g,p} = l_g$, where l_g is a correct process. Thus, after $t'' = max(t,t')$ there is a time when all *ToOrder* sequences contain T and one correct process, l_g, proposes this sequence in consensus. By the uniform integrity property of consensus, T is eventually included in a decision of consensus. □

Lemma 5 *For each correct process p that has tuple $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$, p eventually replaces the entry by $(\textsc{final}, \bot, ts, m)$ in $\mathscr{B}$ where ts is the maximum timestamp x in the* SYNC-HARD *tuples that concern m.*

PROOF: To include $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$, (a) p has decided on a sequence that contains a $(\textsc{set-hard}, h, x, m)$ tuple if m is local (Task 5), or (b) p has decided on a sequence that contains a $(\textsc{sync-hard}, h, x, m)$ tuple if m is global (Task 5), or (c) p has decided on a sequence that contains $(\textsc{sync-soft}, h, x, m)$ and p has included $(\textsc{sync-hard}, h, x, m)$ in *ToOrder*, if m is global (Task 6).

In case (a), $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$ will be trivially replaced by $(\textsc{final}, \bot, x, m)$ in Task 5. In case (b), some process proposed a *ToOrder*$\setminus$*Ordered* sequence that contains $(\textsc{sync-hard}, h, x, m)$. So, for cases (b) and (c), the SYNC-HARD tuple is included in *ToOrder* in Task 3 upon r-delivering tuple $(\textsc{send-hard}, h, x, m)$, which was r-multicast in Task 5, upon the decision of a sequence with $(\textsc{set-hard}, h, x, m)$. Therefore, $(\textsc{set-hard}, h, x, m)$ was included in *ToOrder* at Task 1, as a result of the r-delivery of $(\textsc{start}, \bot, \bot, m)$, which is r-multicast to all of m's destinations. Every group h in $m.dst$ upon r-delivering $(\textsc{start}, \bot, \bot, m)$ adds tuple $(\textsc{set-hard}, h, x, m)$ to *ToOrder*, which from Lemma 4 is eventually included in a consensus decision and results in the r-multicast of $(\textsc{send-hard}, h, x, m)$ to members of $m.dst$. When a process r-delivers $(\textsc{send-hard}, h, x, m)$, it adds $(\textsc{sync-hard}, h, x, m)$ to *ToOrder* and, from Lemma 4, the tuple is decided in an instance of consensus, leading to the inclusion of $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$.

Once for each group h in $m.dst$ there is a tuple $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$, p replaces the SYNC-HARD tuples by $(\textsc{final}, \bot, ts, m)$. □

Lemma 6 *If a correct process p includes $(\textsc{final}, \bot, ts, m)$ in $\mathscr{B}$, then p eventually a-delivers m.*

PROOF: Assume for a contradiction that q does not a-deliver m. Thus, there is some tuple (z, h, y, m') in $\mathscr{B}$ such that $m \neq m'$ and $y < ts$.

It is first shown that eventually any entry (z, h, w, m') added in $\mathscr{B}$ after $(\textsc{final}, \bot, ts, m)$ is in $\mathscr{B}$ has a timestamp larger than ts. Process q only includes $(\textsc{final}, \bot, ts, m)$ in $\mathscr{B}$ (Task 7) after from each $h \in m.dst$ (a) q decided on a sequence with $(\textsc{sync-hard}, h, x, m)$ (Task 5) or (b) q decided on a sequence with $(\textsc{sync-soft}, h, x, m)$ (Task 5) and included $(\textsc{sync-hard}, h, x, m)$ in *ToOrder*, after r-delivering $(\textsc{send-hard}, h, x, m)$ (Task 3). Before q includes $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$ in Tasks 5 and 6, q updates C_h such that it contains the maximum between its current value and x. Since the next SET-HARD event that q handles for a message m'' will increment C_h, m'' will have a timestamp larger than ts.

It is now shown that every message that contains a timestamp smaller than m's final timestamp ts is eventually a-delivered and removed from $\mathscr{B}$. Let (z, h, y, m') be an entry in $\mathscr{B}$ such that $y < ts$. Either z is FINAL or it is SYNC-HARD and from Lemma 5 the tuple will eventually be replaced by a FINAL tuple. Thus, from Task 7 message m' will be eventually a-delivered and removed from $\mathscr{B}$, a contradiction. The conclusion is that q eventually a-delivers m. ☐

Proposition 9 *Validity: If a correct process p a-multicasts a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.*

PROOF: Upon a-multicasting m, p r-multicasts $(\textsc{start}, \bot, \bot, m)$ to all processes in $m.dst$ and from the validity and agreement properties of reliable broadcast, every correct q in $m.dst$ will r-deliver $(\textsc{start}, \bot, \bot, m)$ in Task 1. By Task 1, q includes $(\textsc{set-hard}, g, \bot, m)$ in *ToOrder* and from Lemma 4, the tuple is eventually decided in some instance of consensus. If m is global, q r-multicasts $(\textsc{send-hard}, g, \bot, m)$ to all processes in $m.dst$; if m is local, q includes $(\textsc{sync-hard}, g, \bot, m)$ in $\mathscr{B}$. In the first case, every correct process $r \in m.dst$ eventually r-delivers $(\textsc{send-hard}, g, \bot, m)$ and includes $(\textsc{sync-hard}, g, \bot, m)$ in *ToOrder*. From Lemma 4 the SYNC-HARD tuple is eventually included in a consensus decision and from Task 5 in $\mathscr{B}$. Therefore, eventually SYNC-HARD tuples from every group in $m.dst$ are in $\mathscr{B}$ and q replaces them by $(\textsc{final}, \bot, ts, m)$, where ts is the maximum among the timestamps in the SYNC-HARD tuples. From Lemma 6, q eventually a-delivers m. ☐

Proposition 10 *Uniform agreement: If a process p a-delivers a message m, then eventually all correct processes $q \in m.dst$ a-deliver m.*

PROOF: For process p to a-deliver m, from Task 7, for every group h in $m.dst$, there is a tuple $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$. Process p includes $(\textsc{sync-hard}, h, x, m)$ in $\mathscr{B}$ in the following situations: (a) after p decides on a sequence that contains (SET-HARD, h, x, m) and m is local (Task 5); (b) after p decides on a sequence that includes (SYNC-HARD, h, x, m) and m is global (Task 5); and (c) after p decides on a sequence that includes (SYNC-SOFT, h, x, m) and (SYNC-HARD, h, x, m) is in *ToOrder* (Task 6).

In case (a), from consensus's uniform agreement property, q decides on a sequence that contains (SET-HARD, h, x, m) and since m is local, q includes (SYNC-HARD, h, x, m) in $\mathscr{B}$. From Lemma 6, q eventually a-delivers m. In case (b), following a similar argument as item (a), q decides on a sequence that includes (SYNC-HARD, h, x, m) and includes the tuple in $\mathscr{B}$. From Lemma 6, q eventually a-delivers m.

In case (c), to include (SYNC-HARD, h, x, m) in *ToOrder*, p has r-delivered message (SEND-HARD, h, x, m) (Task 3), which was r-multicast to all of m's destinations in Task 5 upon the decision of a sequence containing tuple (SET-HARD, h, x, m) at group h. From consensus's uniform agreement property, every correct process in h also decides on a sequence with (SET-HARD, h, x, m) and r-multicasts (SEND-HARD, h, x, m) to every process in $m.dst$. Upon r-delivering (SEND-HARD, h, x, m), processes in q's group include tuple (SYNC-HARD, h, x, m) in *ToOrder* and from Lemma 4 decide on a sequence that includes the SYNC-HARD tuple. Then, q includes (SYNC-HARD, h, x, m) in $\mathscr{B}$ and from Lemma 6, q eventually a-delivers m. □

Proposition 11 *Uniform prefix order: For any two messages m and m' and any two processes p and q such that $\{p, q\} \subseteq m.dst \cap m'.dst$, 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: The proposition trivially holds if p and q are in the same group, so assume p is in group g and q is in group h and suppose, by way of contradiction, that p does not a-deliver m' before m nor does q a-deliver m before m'. Without loss of generality, suppose that $m.ts < m'.ts$.

The claim is that q inserts m into $\mathscr{B}$ before a-delivering m'. In order for m (respectively, m') to be a-delivered by p (resp., q), p's (resp., q's) $\mathscr{B}$ must contain tuples (SYNC-HARD, g, x, m) and (SYNC-HARD, h, y, m) (resp., (SYNC-HARD, g, x', m') and (SYNC-HARD, h, y', m')).

For p to include a SYNC-HARD tuple in $\mathscr{B}$, p must have (a) decided a sequence that contains (SYNC-HARD, g, x, m) (recall that m is a global message) (Task 5) or (b) decided a sequence that contains (SYNC-SOFT, g, x, m) and (SYNC-HARD, h, x, m) is in *ToOrder* (Task 6).

In case (a), some process in g included (SYNC-HARD, g, x, m) in *ToOrder*, after r-delivering tuple (SEND-HARD, g, x, m). With a similar argument, some process in g included ($\text{SYNC-HARD}, g, x', m'$) in *ToOrder*, after r-delivering tuple ($\text{SEND-HARD}, g, x', m'$). In cases (a) and (b), let r and s be the processes that r-multicast messages ($\text{SEND-HARD}, g, x, m$) and ($\text{SEND-HARD}, g, x', m'$), respectively, at Task 4. Therefore, r and s decided sequences that include the SET-HARD tuples. Assume that ($\text{SET-HARD}, g, x, m$) is decided before ($\text{SET-HARD}, g, x', m'$). Therefore, before r-multicasting ($\text{SEND-HARD}, g, x', m'$), s r-multicast ($\text{SEND-HARD}, g, x, m$). From the FIFO properties of reliable multicast, q r-delivered the tuples in the order above and it can be shown that ($\text{SYNC-HARD}, g, x, m$) appears in $\mathscr{B}$ before ($\text{SET-HARD}, g, x', m'$), which proves our claim.

Consequently, from the claim, q a-delivers m before m' since $m.ts_q < m'.ts_q$, a contradiction that concludes the proof. ☐

Proposition 12 *Uniform acyclic order: The relation $<$ is acyclic.*

PROOF: Suppose, by way of contradiction, that there exist messages $m_1, ..., m_k$ such that $m_1 < m_2 < ... < m_k < m_1$. From Task 7, processes a-deliver messages following the order of their final timestamps. Thus, there must be processes p and q such that the final timestamps they assign to m_1, $m_1.ts_p$ and $m_1.ts_q$, satisfy $m_1.ts_p < m_1.ts_q$, a contradiction since both p and q receive the same SYNC-HARD tuples used to calculate m_1's final timestamp in Task 7. ☐

Theorem 2 *Algorithm 2 implements atomic multicast.*

PROOF: This follows directly from Propositions 8 through 12. ☐

2.4 Performance evaluation

This section describes the main motivations that guided the experiments design (§2.4.1), details the environments in which experiments were conducted (§2.4.2), explains how a social network benchmark was implemented (§2.4.3), and then presents and discusses the results (§2.4.4–2.4.7). It concludes with a summary of the main findings (§2.4.8).

2.4.1 Evaluation rationale

In the following, choices for environments, benchmarks, and protocols are explained.

Environments. They are three: a LAN, an emulated WAN, and a real WAN. The LAN and emulated WAN provide controlled environments, where experiments can run in isolation; the real WAN represents a setting in which FastCast is expected to be used in practice. FastCast optimizes for the number of communication delays. Thus, the protocol will most likely perform well in environments in which latencies are high. Such conjecture is tested in emulated and real WANs. It is also desirable to understand how FastCast performs in less favorable environments, where the difference between communication and processing delays is less significant (i.e., LAN).

Benchmarks. The microbenchmark consists of 64-byte messages to evaluate particular scenarios in isolation. The number of groups is variable (up to 16 groups, the largest configuration our local infrastructure can accommodate) as well as the number of message destinations. The second benchmark is a social network service which defines message destinations according to user connections in a social network graph. In these two benchmarks, there are executions with a single client to understand the performance of FastCast without queueing effects and executions with multiple clients to evaluate FastCast under stress.

Protocols. FastCast is compared to another genuine atomic multicast protocol, BaseCast, and to a non-genuine atomic multicast protocol that uses a fixed group of processes to order messages, regardless of the message destinations. The fixed group of processes uses Multi-Paxos to order atomically multicast messages. Although the non-genuine protocol does not scale, under low load (i.e., with a single client) it provides a performance reference since messages can be ordered in three communication delays.

2.4.2 Implementation and environments

The prototypes of BaseCast, FastCast and the non-genuine atomic multicast are provided in C. For brevity, hereafter the non-genuine atomic multicast protocol is referred as Multi-Paxos. Libpaxos,[1] a Multi-Paxos C library, provides consensus, and point-to-point communication relies on TCP. A stable leader for each group is defined prior to the execution, which is expected to be the common case.

Local-area network (LAN). This environment consisted of a set of nodes, each node with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8GB of memory, SATA SSD disks, and 1Gbps ethernet card. Each node runs CentOS 7.1 64 bits. The RTT (round-trip time) between nodes in the cluster is around 0.1ms.

Emulated wide-area network (emulated WAN). These experiments use the LAN environment and divides nodes in three "regions", R1, R2 and R3. The latencies between nodes in different regions were emulated using Linux traffic control tools. The used latency values were measured in a real WAN (see below), with average RTT of 70ms (R1⟷R2), 70ms (R2⟷R3), and 144ms (R1⟷R3), and standard deviation of 5%.

---

[1] http://libpaxos.sourceforge.net/paxos_projects.php#libpaxos3

Real wide-area network (WAN). The experiments run on Amazon EC2, a public wide-area network. All the nodes are m3.large instances, with 2 vCPUs and 7.5GB of memory, allocated in three regions: California (R1), North Virginia (R2) and Ireland (R3).

In all experiments, groups contain three processes, each process running in a different node. In WAN setups, clients are distributed in three regions and each process deployed in a group in a different region (see Figure 2.2). Consequently, each group can tolerate the failure of a whole datacenter.

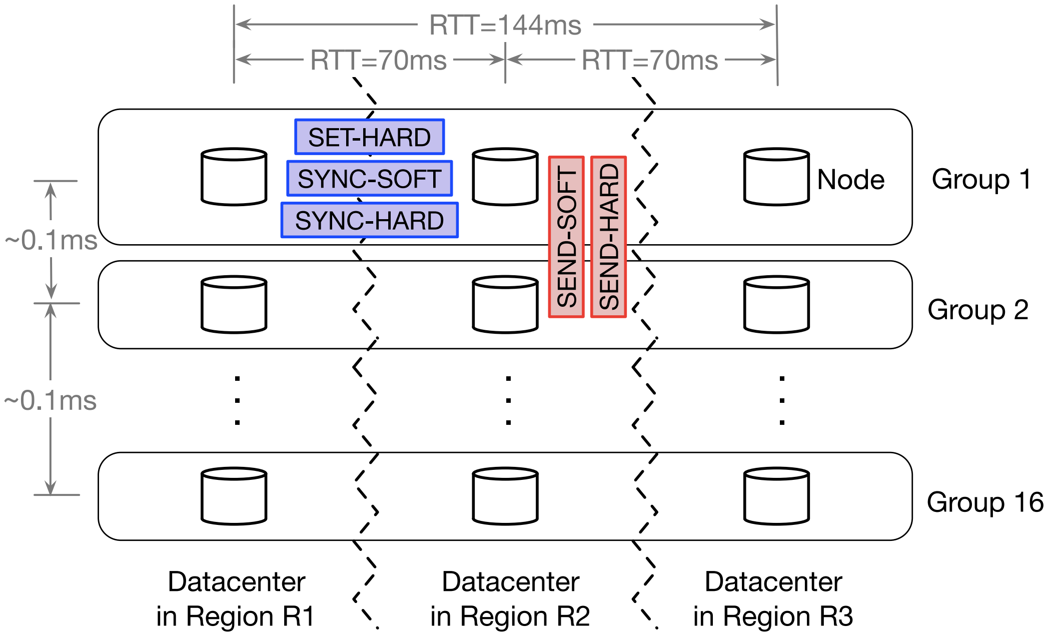

RTT=144ms
RTT=70ms
RTT=70ms
SET-HARD
SYNC-SOFT
SYNC-HARD
SEND-SOFT
SEND-HARD
Node
Group 1
Group 2
Group 16
~0.1ms
~0.1ms
Datacenter in Region R1
Datacenter in Region R2
Datacenter in Region R3

Figure 2.2. Configuration in WAN.

2.4.3 Social network benchmark

The developed social network service is similar to Twitter. Users can follow and unfollow other users, post a message, and read the last messages posted by the users they follow. The social network graph has ten thousand users partitioned in

16 partitions using METIS,[2] a popular graph partitioner. METIS strives to balance the number of users per partition while minimizing the number of edges across partitions. In the social graph, 7110 users have followers in the same partition, 2474 users have followers in two partitions, 376 users have followers in three partitions, and the remaining 40 users have followers in four or five partitions. When a user posts a message (64 bytes), the message is atomically multicast to all the groups that contain followers of the user. This ensures that reads are single-group operations. Since reads and posts are implemented with atomic multicast, this social network service offers strong consistency guarantees. In the experiments, there are only posts, since as mentioned above, reads are always local to a partition.

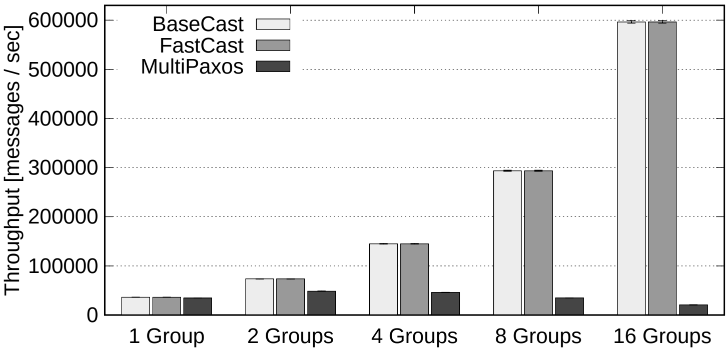


Figure 2.3. Throughput for single-group messages in a LAN.

2.4.4 Microbenchmark in LAN

The first experiment explores the advantage of genuine atomic multicast algorithms over non-genuine approaches like Multi-Paxos. Figure 2.3 shows the throughput in messages per second versus the number of groups, when 200 clients per group multicast local messages only. The results show that genuine atomic multicast protocols result in throughput that increases linearly with the number of groups, from 36000 messages per second with 1 group to almost 600000 messages per second with 16 groups. This happens because genuine protocols only involve the sender of a message and the destination group. Because Multi-Paxos has to order all the messages and is nearly saturated with 200 client, performance with 2 groups is only slightly higher than with 1 group, reaching

---

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

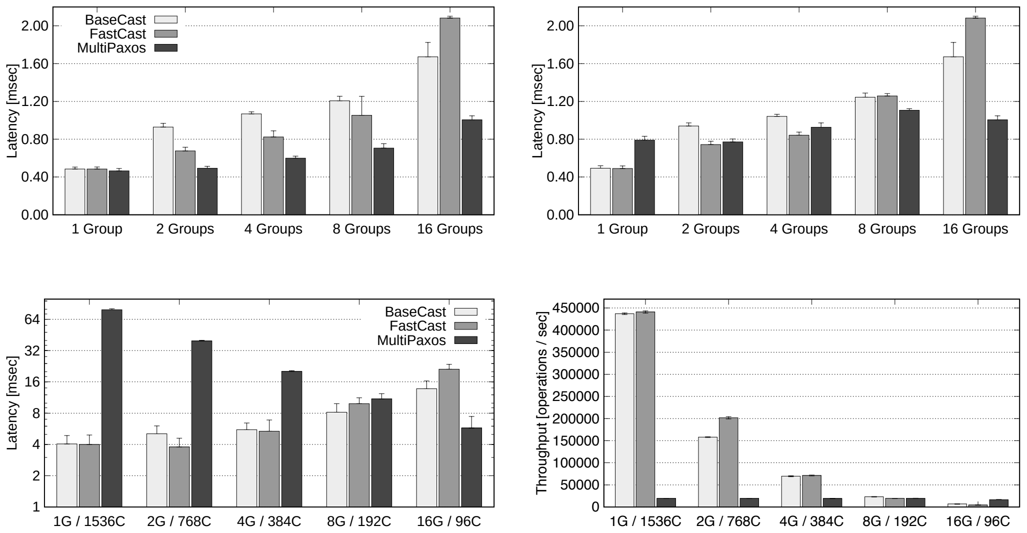


Figure 2.4. Atomic multicast in a LAN. Bars show median latency or average throughput, and whiskers show 95-th percentile or 95% confidence interval. (Top left.) Latency when one client multicasts to all groups versus number of groups in configuration. (Top right.) Latency when one client multicasts to k groups in a system with 16 groups. (Bottom left and right.) Latency and throughput when multiple clients multicast to increasing number of destination groups in a system with 16 groups. Legend in x-axis shows number of destination groups k_g in a multicast message and number of clients k_c in the configuration, where $k_g \times k_c = 1536$.

a maximum of 48000 messages per second. BaseCast and FastCast have identical results because the mechanism to order messages addressed to one group is similar in both algorithms. Even though Multi-Paxos does not scale, it provides a useful reference for latency with few clients (i.e., low load), since it orders messages in three communication delays.

The next experiments assess latency with a single client. This setup aims to check if the reduction in communication delays introduced by FastCast has the expected impact on latency. The considered configurations have an increasing number of groups and clients multicast messages to all groups in the configuration. FastCast's advantage is more noticeable when the number of groups is smaller than 8, with a reduction from 0.928 ms to 0.691ms with 2 groups when compared to BaseCast, and from 1.068 ms to 0.847ms with 4 groups (Figure 2.4, top left). With 16 groups, the overhead introduced by the slow and fast paths in

FastCast impacts latency negatively. Multi-Paxos has lower latency in almost all the cases, an expected result that reflects the advantage of atomic broadcast over atomic multicast when the number of destinations is equal or close to the total number of groups [75]. Figure 2.4 (top right) shows similar results for a fixed number of groups with increasing destinations. Multi-Paxos has higher latency in this configuration since all processes must be reached, even if the destinations are a subset of all groups.

The protocols are now assessed under “operational load”, that is, with enough clients to stress the system, without saturating resources. In these experiments, the number of groups is fixed to 16 and the number of destinations varies in multicast messages. The product *number of destination groups* × *number of clients* is constant and equal to 1536. The rationale is that the cost to multicast a message grows with the number of destinations. Thus, to avoid overloading the system, the number of clients is decreased as the number of destination groups increases.

Figure 2.4 shows latency and throughput (graphs at the bottom). FastCast outperforms BaseCast for 2 destination groups, with a latency and throughput of 3.8 ms and 202000 messages per second against 5.1 ms and 158000 messages per second for BaseCast. With more destinations, the overhead introduced by FastCast’s parallel paths execution make it less efficient than BaseCast, which has a single execution path. Multi-Paxos is CPU-bound with more than 200 clients.

2.4.5 Microbenchmark in emulated WAN

The following experiments test the conjecture that FastCast is more suitable for WANs. The first experiment assesses the latency of the three protocols in the absence of queuing effects. A single client atomically multicasts, in closed loop, global messages. The expected latency for the execution of a Paxos instance in libpaxos is around 70 ms, which is the round-trip time (RTT) between the two closest regions. This is time enough for the Paxos coordinator to receive responses from a quorum of acceptors.

Figure 2.5 (top left) shows that for any number of destination groups, Multi-Paxos and FastCast have a median latency around 1 RTT, while BaseCast has always 2 RTT latency. Although FastCast executes two consensus instances for each global message, just like BaseCast, one consensus is executed in the fast path and the other consensus in the slow path. Since both paths are executed in parallel, FastCast has similar latency as Multi-Paxos. The reduced communication steps, in such scenario, makes FastCast twice as fast as BaseCast. Differently from the results found in a LAN, FastCast largely outperforms BaseCast in configurations with 16 groups when increasing the number of destinations (Figure 2.5,

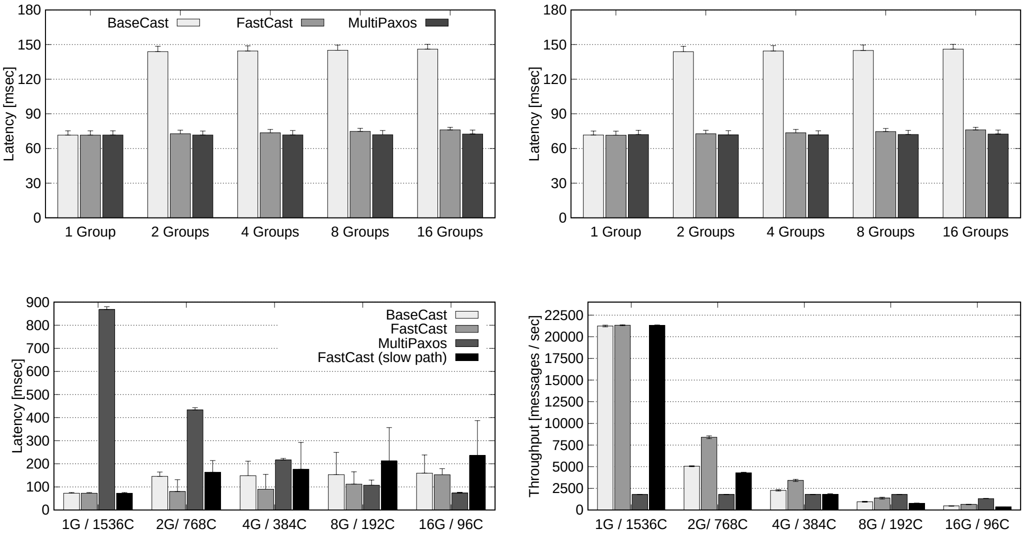


Figure 2.5. Atomic multicast in emulated WAN. Bars show median latency or average throughput, and whiskers show 95-th percentile or 95% confidence interval. (Top left.) Latency when one client multicasts to all groups versus number of groups in configuration. (Top right.) Latency when one client multicasts to k groups in a system with 16 groups. (Bottom left and right.) Latency and throughput when multiple clients multicast to increasing number of destination groups in a system with 16 groups. Legend in x-axis shows number of destination groups k_g in a multicast message and number of clients k_c in the configuration, where $k_g \times k_c = 1536$.

top right).

The next experiments consider multiple clients and 16 groups (Figure 2.5, bottom graphs). As in the LAN experiments, the *number of destinations* × *number of clients* factor is constant for each configuration. These experiments also evaluate the performance of FastCast when messages are ordered through the slow path. For this case, the protocol has been changed to force the leader to make wrong timestamp guesses.

FastCast consistently outperforms BaseCast for global messages up to 8 destination groups. With 2 destination groups, the throughput of FastCast is 70% higher than BaseCast's (8400 versus 5050 messages per second on average). When messages are multicast to all 16 groups, FastCast and BaseCast display similar latency, although FastCast has higher throughput than BaseCast, 647 versus 475 messages per second. Both genuine atomic multicast protocols perform much better than the non-genuine protocol up to 8 destination groups, although the advantage decreases as the number of destinations increases. With 8 destination groups, FastCast and Multi-Paxos have similar latency, and BaseCast performs worse than the two other protocols. With 16 destination groups, Multi-Paxos has lower latency and higher throughput than the other protocols. This fact confirms findings in other works in the literature, which show that when messages address all groups, atomic broadcast protocols have superior performance than atomic multicast protocols [75].

Up to 4 destination groups in a message, FastCast can order messages through the slow path with latency slightly higher than BaseCast's. With 8 and 16 destination groups, the additional overhead of executing three consensus instances in the fast and slow paths degrades the performance of FastCast.

These results establish that, as anticipated, FastCast shines in environments where communication latency is high. In these environments, FastCast can take full advantage of its reduced number of communication delays. Next section aims to confirm these observations in a real WAN.

2.4.6 Microbenchmark in real WAN

To confirm the results obtained with the emulated WAN, the same set of experiments is executed on Amazon EC2. The experiments used up to 48 nodes for the maximum of 16 groups and one additional nodes per group to run the clients.

The results for executions with a single client (Figure 2.6, graphs on the top) are very similar to those found with the emulated WAN. (recall that in the emulated WAN the RTT values between regions correspond to measurements in the real WAN.) For the experiments with increased load, FastCast had slightly better performance in the real WAN than in the emulated WAN for configurations with 8 and 16 groups. This improvement is due to the fact that m3.large instances have better processors than the ones in our LAN and emulated WAN, which reduces FastCast's CPU overhead.

FastCast beats BaseCast in all configurations, with latency of 84 ms for 2, 4 and 8 destination groups and 101 ms for 16 groups. BaseCast has a latency between 163 ms and 170 ms in all cases (Figure 2.6, bottom left). Regarding throughput, Figure 2.6 (bottom right) shows that FastCast is 80% faster than BaseCast with 2 destination groups (4600 vs. 8350 messages per second) and more than 60% faster when messages are addressed to all the available groups (565 vs. 913 messages per second). Multi-Paxos outperforms the other two protocols when all the groups are in the destination of the messages, delivering 1213 messages per second against the 913 messages per second of FastCast.

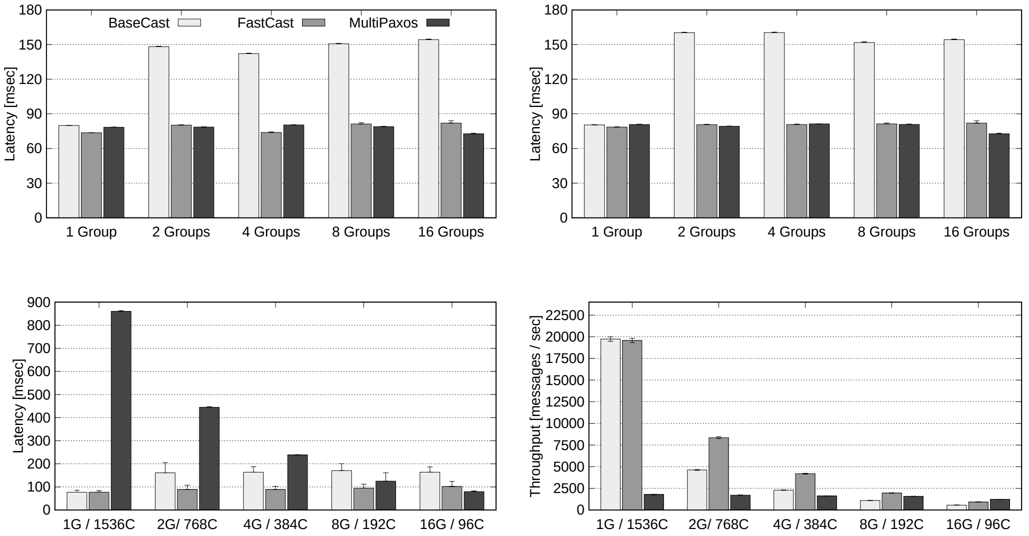


Figure 2.6. Atomic multicast in a real WAN. Bars show median latency or average throughput, and whiskers show 90-th percentile or 95% confidence interval. (Top left.) Latency when one client multicasts to all groups versus number of groups in configuration. (Top right.) Latency when one client multicasts to k groups in a system with 16 groups. (Bottom left and right.) Latency and throughput when multiple clients multicast to increasing number of destination groups in a system with 16 groups. Legend in x-axis shows number of destination groups k_g in a multicast message and number of clients k_c in the configuration, where $k_g \times k_c = 1536$.

2.4.7 Social network in emulated WAN

In the first experiment, a single client posts messages in a closed loop. Each post operation results in the posted message atomically multicast to all the groups in which the poster has at least one follower. Figure 2.7 (top left) shows that FastCast performs similarly to Multi-Paxos, despite the additional consensus instance needed by FastCast to order global messages. FastCast's latency is in the range 73–76ms, which approximates 1 RTT). BaseCast, on the contrary, has latency 2 times greater because of the two sequential consensus executions needed to order global messages.

The next experiments consider executions with an increasing number of clients. Figure 2.7 (top right) compares the throughput of the three protocols. Up to 3200 clients, that is, 200 clients per group, FastCast outperforms both BaseCast and Multi-Paxos. After 3200 clients, FastCast saturates with a throughput of 12500 posts per second. BaseCast eventually reaching FastCast with similar throughput before both systems reach saturation, with 4000 clients.

Figure 2.7 (both graphs at the bottom) shows the latencies for executions with 800 and 1600 clients. Multi-Paxos is overwhelmed in both cases. With 800 clients, the latency of FastCast is close to 1 RTT, between 80ms and 90ms, while the latency of BaseCast is around 150ms. With 1600 clients (100 clients per group), FastCast's latency increases to 130ms, while the latency of BaseCast reaches 200ms.

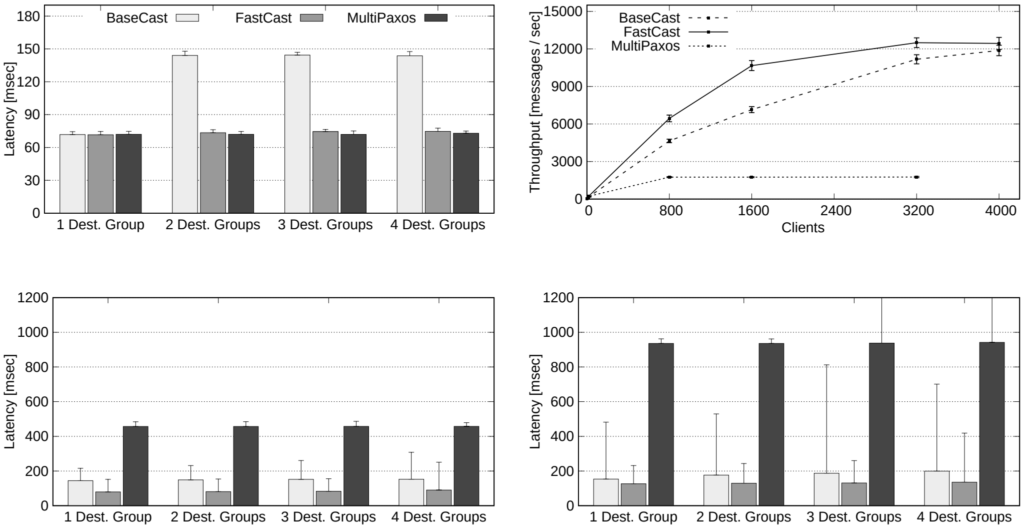


Figure 2.7. Social network application in an emulated WAN. Bars show median latency, lines show average throughput, and whiskers show 90-th percentile or 95% confidence interval. (Top left.) Latency when one client executes 'post' operations versus number of groups in the client's followers list. (Top right.) Throughput versus number of clients executing 'post' operations. (Bottom left.) Latency when 800 clients execute 'post' operations versus number of groups in the clients' followers list. (Bottom right.) Latency when 1600 clients execute 'post' operations versus number of groups in the clients' followers list.

2.4.8 Summary

This section summarizes the results of the performance evaluation. Table 2.1 compares the protocols based on the results presented in the previous sections.

The following conclusions are drawn from the evaluation.

- FastCast outperforms other protocols in most configurations in a WAN (emulated and real), with a few exceptions: (a) For local messages, under low load all protocols perform the same and under high load FastCast and BaseCast have similar performance, better than Multi-Paxos's. (b) For messages addressed to all destinations, Multi-Paxos performs best.
- In LAN under high load, no protocol stands out. FastCast performs best with few destination groups, BaseCast with many destination groups, but not all, when Multi-Paxos is better than the other two.
- Multi-Paxos outperforms the genuine protocols in the LAN with one client, and in all cases when messages are multicast to all destinations, with one exception (microbenchmark in emulated WAN, single client and 16 destinations).

Benchmark	Environment	Load	Destinations in system with 16 groups				
			1	2	4	8	16
Micro bechmark	LAN	low	all equal	MP	MP	MP	MP
		high	BC **FC**	**FC**	BC **FC**	BC	MP
	emulated WAN	low	all equal	**FC** MP	**FC** MP	**FC** MP	MP
		high	BC **FC**	**FC**	**FC**	**FC**	MP
	real WAN	low	all equal	**FC** MP	**FC** MP	**FC** MP	MP
		high	BC **FC**	**FC**	**FC**	**FC**	MP
Social Network	emulated WAN	low	all equal	**FC** MP	**FC** MP	Not applicable	
		high	**FC**	**FC**	**FC**	Not applicable	

Table 2.1. How atomic multicast protocols compare. FC = FastCast, BC = BaseCast, MP = Multi-Paxos; each cell shows the best performing protocol(s) in the given configuration.

2.5 Related work

Several multicast and broadcast algorithms have been proposed in the literature [31]. Moreover, many systems ensure strong consistency with “ad hoc” ordering protocols that do not implement all the properties of atomic multicast (e.g., [28, 44, 79]). The focus of this section is on atomic multicast algorithms.

Existing atomic multicast algorithms fall into one of three categories: *timestamp-based*, *round-based*, and *ring-based*.

Algorithms based on timestamps (i.e., [35, 70, 74] and the algorithms proposed in this work) are genuine and variations of an early atomic multicast algorithm [15], designed for failure-free systems. In these algorithms, processes assign timestamps to messages, ensure that destinations agree on the final timestamp assigned to each message, and deliver messages following this timestamp order. The algorithms in [35, 74] have a best-case time complexity of 6δ for the delivery of global messages. The algorithm in [70] can deliver global messages in 5δ and it ensures another property besides genuineness called *message-minimality*. This property states that the messages of the algorithm have a size proportional to the number of destination groups of the multicast message, and not to the total number of processes. The two algorithms proposed in this chapter verify this property as well.

In round-based algorithms, processes execute an unbounded sequence of rounds and agree on messages delivered at the end of each round. A round-based atomic multicast algorithm that can deliver messages in 4δ is presented in [74]. Differently from FastCast, which can also deliver global messages in 4δ, the algorithm in [74] is not genuine.

Ring-based algorithms propagate messages along a predefined ring overlay and ensure atomic multicast properties by relying on this topology. An atomic multicast algorithm in this category is proposed in [32], where consensus is run among the members of each group. The time complexity of this algorithm is proportional to the number of destination groups.

Multi-Ring Paxos [56], Spread [2, 9], and Ridge [14] are ring-based non-genuine atomic multicast protocols. On the one hand, to deliver a message m, they require communication with processes outside of the destination groups of m. On the other hand, these protocols do not require disjoint groups.

Although FastCast is the first optimistic atomic multicast protocol, optimistic execution to improve performance has been explored before in other contexts, such as atomic broadcast (e.g., [64, 65, 88]) and quorum systems [38].

2.6 Conclusion

Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available strongly consistent distributed systems. This thesis proposes FastCast, the first genuine atomic multicast algorithm that can order local messages, addressed to a single group, in three communication delays and global messages, addressed to multiple groups of processes, in four communication delays. In addition to introducing a novel genuine atomic multicast algorithm, its performance has been assessed in three different environments. The results show that FastCast largely outperforms other genuine and non-genuine atomic multicast protocols.

Chapter 3

Making atomic multicast safer

Although research on efficient atomic multicast protocols is relatively mature [2, 32, 35, 70], to date all existing protocols target benign failures (e.g., crash failures) [9, 14, 26, 56]. This chapter introduces ByzCast, the first Byzantine Fault-Tolerant (BFT) atomic multicast protocol. Byzantine fault tolerance has become increasingly appealing as service providers can deploy their systems in increasingly inexpensive hardware (e.g., cloud environments) and new applications become more and more sensitive to malicious behavior (e.g., blockchain [19]).

ByzCast's design was motivated by two driving forces: (i) The desire to reuse existing BFT tools and libraries, instead of coming up with protocols that would require an implementation from scratch. (ii) The perception that the usefulness of atomic multicast lies in its ability to deliver scalable performance. On the one hand, much effort has been put into designing, implementing, debugging and performance-tuning BFT atomic broadcast protocols (i.e., a special case of atomic multicast in which messages always address the same set of destinations) [5, 12, 21, 58, 81]. One could build on these solutions and thereby shorten the development cycle of BFT atomic multicast protocol. On the other hand, it would not be difficult to achieve the first goal above with a naive atomic multicast protocol that trivially relies on atomic broadcast. For example, one could use a fixed group of processes to order all multicast messages (using atomic broadcast) and then relay the ordered messages to their actual destinations. Instead, one should aim at atomic multicast protocols that are *genuine*, that is, only the message sender and the message destinations should communicate to order multicast messages [37]. A genuine atomic multicast is the foundation of scalable systems, since it does not depend on a fixed group of processes and does not involve processes unnecessarily.

ByzCast conciliates these goals with a compromise between reusability and scalability: the resulting protocol is more complex than the naive variant described above and *partially genuine*. ByzCast is partially genuine in that messages atomically multicast to a single group of processes only require coordination between the message sender and the destination group; messages addressed to multiple groups of processes, however, may involve processes that are not part of the destination (i.e., these processes help order the messages though). The motivation for partially genuine atomic multicast protocols comes from the observation that when sharding a service state for performance, service providers strive to maximize the number of requests that can be served by a single shard alone.

ByzCast is a hierarchical protocol. It uses an overlay tree where a node in the tree is a group of processes. Each group of processes runs an instance of atomic broadcast that encompasses the processes in the group. Hence, ordering messages multicast to a single group is easy enough: it suffices to use the atomic broadcast instance implemented by the destination group. Ordering messages that address multiple groups is trickier. First, it requires ordering such a message in the lowest common ancestor group of the message's destinations (in the worst case the root). Then, the message is successively ordered by the lower groups in the tree until it reaches the message's destination groups. The main invariant of ByzCast is that the lower groups in the tree preserve the order induced by the higher groups.

In addition to proposing a partially genuine atomic multicast protocol that builds on multiple instances of atomic broadcast (one instance per group of processes), this work also considers the problem of building an efficient overlay tree. The structure of the overlay tree is mostly important for messages that address multiple groups. In its simplest form, one could have a two-level tree: any messages that address more than one destination would be first ordered by the root group and then by the destination groups, the leaves of the tree. In this simple tree, however, the root could become a performance bottleneck. More efficient solutions, based on more complex trees, are possible if one accounts for the workload when computing ByzCast's overlay tree. This discussion is framed as an optimization problem.

The work described in this chapter makes the following contributions:

- it presents a partially genuine atomic multicast protocol that builds on multiple instances of atomic broadcast, a problem that has been extensively studied and for which efficient libraries exist (e.g. [12, 59, 61, 84]).
- it defines the problem of building an overlay tree as an optimization problem. The optimization model takes into account the frequency of messages

per destination and the performance of a group alone.

- it describes a prototype of ByzCast developed using BFT-SMaRt [12], a well-established library that implements BFT atomic broadcast.
- it provides a detailed experimental evaluation of ByzCast and compares ByzCast to a naive atomic multicast solution.

The rest of the chapter is organized as follows. Section 3.1 presents ByzCast, its performance optimizer, and correctness proof. Section 3.3 details the prototype. Section 3.4 describes the experimental evaluation. Section 3.5 surveys related work and Section 3.6 concludes the chapter.

3.1 Byzantine Fault Tolerant Atomic Multicast

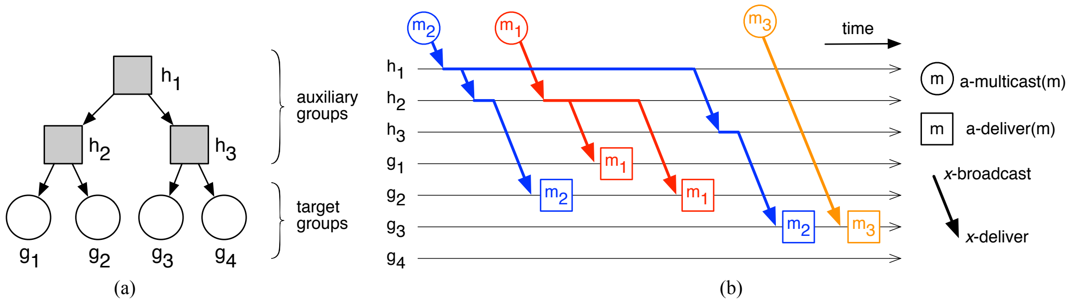


Figure 3.1. (a) An overlay tree used in ByzCast with four target groups and three auxiliary groups. (b) An execution of ByzCast with three messages: m_1 is a-multicast to $\{g_1, g_2\}$, m_2 to $\{g_2, g_3\}$, and m_3 to g_3. For clarity, each group has one (correct) process.

This section explains the rationale behind the design of ByzCast (§3.1.1), presents the protocol in detail (§3.1.2), shows how to optimize ByzCast for different workloads (§3.1.3), and then argues about its correctness (§3.2).

3.1.1 Rationale

The design of ByzCast was guided by two high-level goals:

Building on existing solutions

Research on Byzantine Fault Tolerant agreement protocols is mature (see §3.5). One of the main goals was to devise an atomic multicast protocol that could reuse existing BFT software, instead of designing a protocol that would require an implementation completely from scratch.

Striving for scalable protocols

Genuineness is the property that best captures scalability in atomic multicast. By requiring only the groups in the destination of a message to coordinate to order the message, a genuine atomic multicast protocol can scale with the number of groups while saving resources.

3.1.2 Protocol

For clarity, the following description represents a version of ByzCast that uses additional groups of servers to help order messages. Hereafter, the groups in $\Gamma = \{g_1, ..., g_m\}$ are referred as *target groups* and the additional server groups in $\Lambda = \{h_1, ..., h_n\}$ as *auxiliary groups*. As with target groups, each auxiliary group has $3f + 1$ processes, with at most f faulty processes, a necessary condition for solving consensus with Byzantine adversaries.

Each group x in ByzCast (both target and auxiliary) implements a FIFO atomic broadcast, as defined in §1.2.4. The atomic broadcast in group x is implemented by x's members and independent from the atomic broadcast of other groups. There is a distinction between the primitives of atomic multicast, denoted as a-multicast and a-deliver, and the primitives of the atomic broadcast of group x, denoted as x-broadcast and x-deliver.

ByzCast arranges groups in a tree overlay where the leaves of the tree are target groups and the inner nodes of the tree are auxiliary groups. The *reach* of a group x, $\mathit{reach}(x)$ is defined as the set of target groups that can be reached from x by walking down the tree. In Figure 3.1 (a), $\mathit{reach}(h_1) = \{g_1, g_2, g_3, g_4\}$, $\mathit{reach}(h_2) = \{g_1, g_2\}$, and $\mathit{reach}(h_3) = \{g_3, g_4\}$. The set $\mathit{children}(x)$ denotes the children of a group x in the tree.

To a-multicast a message m to a set of target groups in $m.dst$ (see Algorithm 3), a process first x_0-broadcasts m in the *lowest common ancestor* group x_0 of (the groups in) $m.dst$, denoted $lca(m.dst)$.

When m is x_k-delivered by processes in x_k, each process x_{k+1}-broadcasts m in x_k's child group x_{k+1} if x_{k+1}'s reach intersects $m.dst$. This procedure continues until target groups in $m.dst$ x_k-deliver m, which triggers the a-deliver of m.

To account for Byzantine processes in group x_k, processes in x_{k+1} only handle m once they x_{k+1}-deliver m $f + 1$ times. This ensures that m was x_{k+1}-broadcast by at least one correct process in x_k and, by inductive reasoning, m

Algorithm 3 ByzCast

```
Initialization
  $\mathcal{T}$ is an overlay tree with groups $\Gamma \cup \Lambda$
  $A\text{-}delivered \leftarrow \emptyset$

To a-multicast message $m$:
  $x_0 \leftarrow lca(m.dst)$
  $x_0$-broadcast($m$)

Each server process $p$ in group $x_k$ executes as follows:
  when $x_k$-deliver($m$)
    if $k = 0$ or $x_k$-delivered $m$ $(f+1)$ times then
      for each $x_{k+1} \in children(x_k)$ such that
                  $m.dst \cap reach(x_{k+1}) \neq \emptyset$ do
        $x_{k+1}$-broadcast($m$)
      if $x_k \in m.dst$ and $m \notin A\text{-}delivered$ then
        a-deliver($m$)
        $A\text{-}delivered \leftarrow A\text{-}delivered \cup \{m\}$
```

was a-multicast by a client (and not fabricated by a malicious server).

Intuitively, ByzCast atomic order is a consequence of two invariants:

1. Any two messages m and m' atomically multicast to common destinations are ordered by at least one inner group x_k in the tree.
2. If m is ordered before m' in x_k, then m is ordered before m' in any other group that orders both messages (thanks to the FIFO atomic broadcast used in each group).

Figure 3.1 (b) illustrates an execution of ByzCast with messages m_1, m_2 and m_3 a-multicast to groups $\{g_1, g_2\}$, $\{g_2, g_3\}$, and $\{g_3\}$, respectively. Assuming the overlay tree shown in Figure 3.1 (a), m_1 is first h_2-broadcast in group h_2. Upon h_2-delivering m_1, processes in h_2 atomically broadcast m_1 in g_1 and in g_2. Message m_2 is first h_1-broadcast, and then it continues down the tree until it is delivered by g_2 and g_3, its destination target groups. Message m_3 is g_3-broadcast in g_3 directly since it is addressed to a single group. The order between m_1 and m_2 is determined by their delivery order at h_2 since h_2 is the highest group to deliver both messages.

ByzCast is a partially genuine atomic multicast protocol. While messages addressed to a single group are ordered by processes in the destination group only, messages addressed to multiple groups may involve auxiliary groups. For example, in Figure 3.1, the atomic multicast of m_1 (resp., m_2) involves h_2 (resp., h_1, h_2

and h_3), which is not a destination of m_1 (resp., m_2). Since m_3 involves a single destination group, only m_3's sender and g_3, m_3's destination, must coordinate to order the message.

Finally, even though ByzCast has been described with auxiliary groups as inner nodes of the tree, Algorithm 3 does not need this restriction: target groups can be inner nodes in the overlay tree, i.e., a tree can contain target groups only.

3.1.3 Optimizations

The performance of messages multicast to multiple groups largely depends on ByzCast overlay tree. Laying out the overlay tree is an optimization problem with conflicting goals: on the one hand, the aim at short trees to reduce the latency of global messages; on the other hand, when laying out the tree, one must avoid overloading groups. For example, in Figure 3.1, the height of the lowest common ancestor of m_1 and m_2 are two and three, respectively. A two-level tree where the four target groups descend directly from one auxiliary group would improve the latency of global messages. However, in a two-level tree all global messages must start at the root group, which could become a performance bottleneck.

The problem of laying out an optimized ByzCast tree is formulated next. The following parameters are input:

- Γ and Λ as already defined, and $\mathcal{N} = \Gamma \cup \Lambda$;
- $D \subseteq \mathcal{P}(\Gamma)$: all possible destinations of a message, where $\mathcal{P}(\Gamma)$ is the power set of Γ;
- $F(d)$: maximum load in messages per second multicast to destinations d in the workload, where $d \in D$; and
- $K(x)$: maximum performance in messages per second that group x can sustain, $\forall x \in \mathcal{N}$.

Given this input, the problem consists in finding the directed edges $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ of the optimized overlay tree $\mathcal{T} = (\mathcal{N}, \mathcal{E})$. To more precisely state the optimization function with constraints, we introduce additional definitions.

- $P(\mathcal{T}, d)$: the set of groups involved in a multicast to d (i.e., groups in the paths from $lca(d)$ to all groups in d);
- $H(\mathcal{T}, d)$: the height of the lowest common ancestor of groups in d;
- $T(\mathcal{T}, x) = \{d \mid d \in D \text{ and } x \in P(\mathcal{T}, d)\}$: set of destinations that involve group x; and
- $L(\mathcal{T}, x) = \sum_{d \in T(\mathcal{T}, x)} F(d)$: load imposed on group x.

Among the candidate overlay trees, respecting the above restrictions, we are interested in those that minimize the height of the various destinations.

$$\text{minimize} \sum_{d \in D} H(\mathscr{T}, d)$$

In addition to topological constraints, the load imposed to each group must respect its capacity.

$$\text{subject to } \forall x : L(\mathscr{T}, x) \leq K(x)$$

3.2 Proof of Correctness

This section presents the proof that ByzCast satisfies all the properties of atomic multicast (§1.2.3).

Lemma 7 *For any message m atomically multicast to multiple groups, let group* x_0 *be the lowest common ancestor of* $m.dst$*. For all* $x_d \in m.dst$*, if correct process p in* x_0 x_0*-delivers m, then all correct processes in the path* $x_1, ..., x_d$*,* x_k*-deliver m* $(f+1)$ *times, where* $1 \leq k \leq d$*.*

PROOF: By induction. (Base step.) Since p x_0-delivers m, x_1 is a child of x_0, and $reach(x_1) \cap m.dst \neq \emptyset$, p x_1-broadcasts m. The claim follows from the validity of atomic broadcast and the fact that there are $2f+1$ correct processes in x_0. (Inductive step.) Assume each correct process r in x_k x_k-deliver m at least $(f+1)$ times. From Algorithm 3, and the fact that x_{k+1} is a child of x_k and $reach(x_{k+1}) \cap m.dst \neq \emptyset$, r x_{k+1}-broadcasts m. From the validity of atomic broadcast and the fact that there are $2f+1$ correct processes in x_k, every correct process in x_{k+1} x_{k+1}-delivers m. □

Lemma 8 *For any atomically multicast message m, let group* x_0 *be the lowest common ancestor of* $m.dst$*. For all* $x_d \in m.dst$*, if correct process p in* x_d x_d*-delivers m, then all correct processes in the path* $x_0, ..., x_d$*,* x_k*-deliver m, where* $0 \leq k \leq d$*.*

PROOF: By backwards induction. (*Base step.*) The case for $k = d$ follows directly from agreement of atomic broadcast in group x_d. (*Inductive step.*) Assume that every correct process $r \in x_k$ x_k-delivers m. It is shown that correct processes in x_{k-1} x_{k-1}-deliver m. From Algorithm 3, r x_k-delivered m $(f+1)$ times. From integrity of atomic broadcast in x_k, at least one correct process s in x_{k-1} x_k-broadcasts m. Therefore, s x_{k-1}-delivers m, and from agreement of atomic broadcast in x_{k-1} all correct processes x_{k-1}-deliver m. □

Proposition 13 *(Validity) If a correct process p a-multicasts a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, a-deliver m.*

PROOF: Let group x_0 be the lowest common ancestor of $m.dst$ and x_d a group in $m.dst$. From Algorithm 3, p x_0-broadcasts m and from validity of atomic broadcast, all correct processes in x_0 x_0-deliver m. From Lemma 7, all correct processes in x_d, x_d-deliver m $(f+1)$ times. Hence, every correct process in x_d a-delivers m. ☐

Proposition 14 *(Agreement) If a correct process p in group x_d a-delivers a message m, then eventually all correct processes $q \in g$, where $g \in m.dst$, a-deliver m.*

PROOF: From Lemma 8, all correct processes in x_0, x_0-deliver m. Thus, from Lemma 7, all $x_d \in m.dst$ x_d-deliver m $(f+1)$ times. It follows from Algorithm 3 that all $q \in x_d$ a-deliver m. ☐

Proposition 15 *(Integrity) For any correct process p and any message m, p a-delivers m at most once, and only if $p \in g$, $g \in m.dst$, and m was previously a-multicast.*

PROOF: From Algorithm 3, it follows immediately that a correct process $p \in g$ a-delivers m at most once, only if $g \in m.dst$ and m is a-multicast. ☐

Lemma 9 *If m and m′ are two messages atomically multicast to one or more destination groups in common, then $lca(m) \in subtree(m')$ or $lca(m') \in subtree(m)$.*

PROOF: Assume group x is a common destination in m and m' (i.e., $x \in m.dst \cap m'.dst$). Let $path(x)$ be the sequence of groups in the overlay tree $\mathcal{T}$ from the root until x. From Algorithm 3, in order to reach x, $lca(m)$ (resp., $lca(m')$) must be a group in $path(x)$. Without loss of generality, assume that $lca(m)$ is higher than $lca(m')$ or at the same height as $lca(m')$. Then, $lca(m') \in subtree(m)$, which concludes the lemma. ☐

Lemma 10 *If a correct process in group x_0 x_0-delivers m before m′, then for every ancestor group x_d of x_0, where $x_d \in m.dst \cap m'.dst$, every correct process in x_d x_d-delivers m before m′.*

PROOF: By induction on the path $x_0, ..., x_k, ..., x_d$. (*Base step.*) Trivially from the properties of atomic broadcast in group x_0. (*Inductive step.*) Let $p \in x_k$ x_k-deliver m before m'. Thus, p x_{k+1}-broadcasts m before m' and from the FIFO guarantee of atomic broadcast in x_{k+1}, every correct process $q \in x_{k+1}$ x_{k+1}-delivers m before m'. □

Proposition 16 *(Prefix order) For any two messages m and m' and any two correct processes p and q such that $p \in g$, $q \in h$ and $\{g, h\} \subseteq m.dst \cap m'.dst$, 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: The proposition holds trivially if p and q are in the same group, so assume that $g \neq h$. From Lemma 9, and without loss of generality, assume that $lca(m') \in subtree(m)$. Thus, $lca(m')$ will order m and m'. From Lemma 10, both p and q a-deliver m and m' in the same order as $lca(m')$. □

Proposition 17 *(Acyclic order) The relation $<$ is acyclic.*

PROOF: For a contradiction, assume there is an execution of ByzCast that results in a cycle $m_0 < ... < m_d < m_0$. Since all correct processes in the same group a-deliver messages in the same order, the cycle must involve messages a-multicast to multiple groups. Let x be the highest lowest common ancestor of all messages in the cycle. Consider $subtree(x, 1), subtree(x, 2), ...$ as the subtrees of group x in $\mathcal{T}$. Since the cycle involves groups in the subtree of x, there must exist messages m and m' such that (a) m is a-delivered before m' in groups in $subtree(x, i)$ and (b) m' is a-delivered before m in groups in $subtree(x,j), i \neq j$. From Lemma 10, item (a) implies that processes in x x-deliver m and then m', and item (b) implies that processes in x x-deliver m' and then m, a contradiction. □

3.3 Implementation

ByzCast was implemented on top of BFT-SMaRt, a well-known library for BFT replication [12]. This library has been used in many academic projects and a few recent blockchain systems (e.g., [19, 82]).

BFT-SMaRt message ordering is implemented through the Mod-SMaRt algorithm [80], which uses the Byzantine-variant of Paxos described in [18] to establish consensus on the i-th (batch of) operation(s) to be processed by the replicated state machine. The leader starts a consensus instance every time there are pending client requests to be processed and there are no consensus being executed. Consensus follows a message pattern similar to PBFT [21]: the leader *proposes* a batch of messages to be processed, the replicas validate this proposal by *writing* the proposal in the other replicas; the replicas *accept* the proposal if a Byzantine quorum of $n-f$ replicas perform the write. When a replica learns that $n-f$ replicas accepted the proposal, it executes the operation and sends replies to the clients. In case of leader failure or network asynchrony, a new leader is elected. BFT-SMaRt also implements protocols for replica recovering (i.e., state transfer), and group reconfiguration [12].

In ByzCast, each group (either target or auxiliary) corresponds to a BFT replicated state machine. Each replica in auxiliary groups connects to all the replicas in the next level. The implementation considers two overlay trees. A three-level tree, as the one presented in Figure 3.1 and a two-level tree that uses a single auxiliary group to order global messages.

Replicas only process messages from a higher-level group when they (FIFO) a-deliver them $f+1$ times. Target groups execute a-delivered messages and reply either to clients or to auxiliary groups whether the message is local or global. Both clients and auxiliary groups wait for $f+1$ correct replies. Figure 3.2 depicts the described logic in executions of a request from a client in a local and a global message. Except for client requests, which are single messages, all messages exchanges between groups need $f+1$ equal responses before they can be processed. Even though multiple processes in group invoke the broadcast of a message in another group, thanks to BFT-SMaRt's batching optimization, it is likely that all such invocations are ordered in a single instance of consensus.

Clients run in a closed loop (i.e., only send a new message after the previous message reply) and forward messages to every replica in the lowest common ancestor group of the message. ByzCast was implemented in Java and the source code is publicly available.[1]

3.4 Performance evaluation

This section describes the main motivations that guided the design of experiments (§3.4.1), details the environments in which experiments were conducted (§3.4.2), and then presents and discusses the results (§3.4.3–3.4.8).

---

[1] `https://github.com/tarcisiocjr/byzcast`.

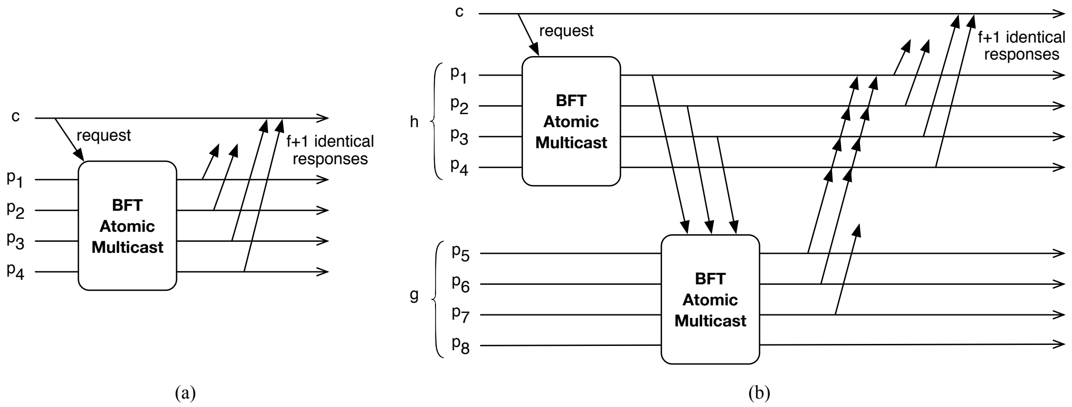


Figure 3.2. Executions of ByzCast with (a) a local message and (b) a global message. Each group has four processes, one of which may be Byzantine. For clarity, the execution of ByzCast shows a single target group only.

3.4.1 Evaluation rationale

In the following, the choices for environments, benchmarks, and protocols are detailed.

Environments

Experiments run on a local-area network (LAN) and a wide-area network (WAN). The LAN provides a controlled environment, where experiments run in isolation; the WAN represents a more challenging setting.

Benchmarks

A microbenchmark with 64-byte messages is useful to evaluate particular scenarios in isolation. The number of groups varies (up to 8 groups, the largest configuration accommodated in our local infrastructure) as well as the number of message destinations. Two layouts are considered for the ByzCast tree: a 2-level and a 3-level tree. Executions with a single client allow to understand the performance of ByzCast without queueing effects, and with multiple clients to evaluate the solution under stress. Finally, experiments consider workloads with and without locality (i.e., skewed access).

Protocols

ByzCast is compared to BFT-SMaRt and to a non-genuine 2-level atomic multicast protocol, called Baseline. BFT-SMaRt uses a single group and provides a reference to the performance of ByzCast with local messages. The Baseline protocol has one auxiliary group that orders all messages regardless of the message destination. After the message is ordered, it is forwarded to its destinations. Each process in the target group waits until it receives the message from $f + 1$ processes in the auxiliary group. Although the non-genuine protocol does not scale, it provides a performance reference for global messages.

3.4.2 Environments and configuration

Details about the environments for the evaluation of the three protocols are presented next.

Local-area network (LAN)

This environment consisted of a set replica nodes with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8GB of memory, SATA SSD disks, and 1Gbps ethernet card; and clients nodes with a four-core AMD Opteron 2212 processor at 2.0GHz, 4GB of memory, and 1Gbps ethernet card. Each node runs CentOS 7.1 64 bits. The RTT (round-trip time) between nodes in the cluster is around 0.1ms.

Wide-area network (WAN)

Amazon EC2, a public wide-area network, is used. All nodes are c4.xlarge instances, with 4 vCPUs and 7.5GB of memory. Nodes are allocated in four regions: California (R1), North Virginia (R2), Frankfurt (R3) and Tokyo (R4). Table 3.1 summarizes the latency between pairs of regions in milliseconds.

	EU	**CA**	**VA**	**JP**
CA	165	–	70	112
VA	88	70	–	175
JP	239	112	175	–

Table 3.1. Latencies within Amazon EC2 infrastructure.

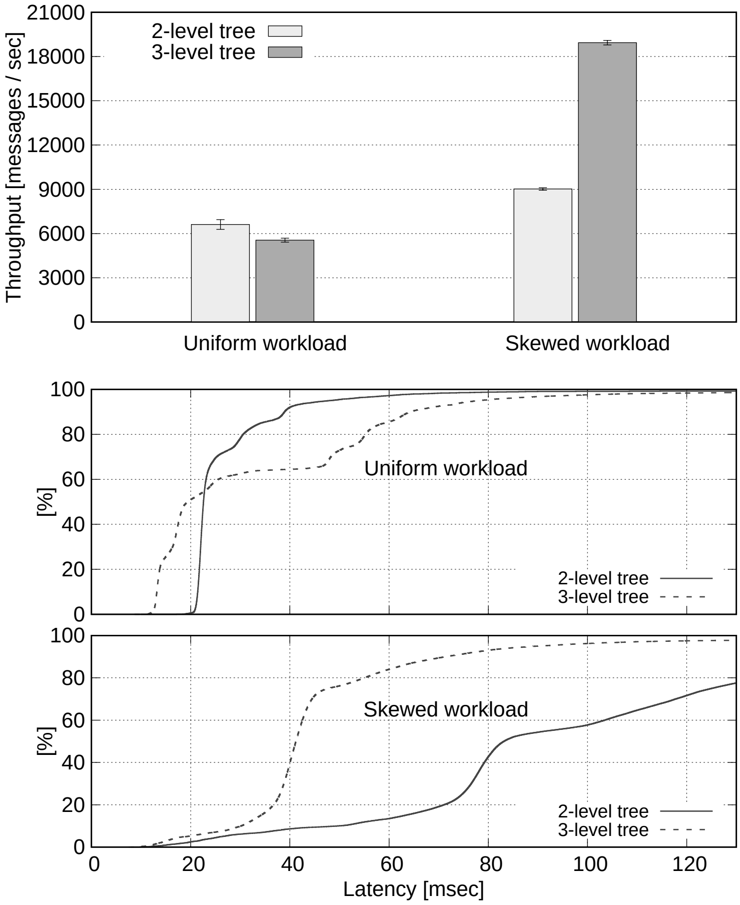


Figure 3.3. ByzCast global messages throughput and latency CDF with 2-level and 3-level trees. Whiskers show 95% confidence interval.

Configuration

In all experiments, groups contain four processes, each process running in a different node. The number of groups depends on the tree layout. In the 2-level tree the number of target groups varies from 2 to 8 with 1 auxiliary for global messages. In the 3-level tree the number of target groups is fixed to 4 and the number of auxiliary groups to 3, as depicted in Figure 3.1. In the WAN setup, clients are distributed along all the regions and deploy each process of a group in a different region. Consequently, the system can tolerate the failure of a whole region.

Uniform workload	
$D_u = \{\{g_i, g_j\} \mid 1 \leq i, j \leq 4 \land i \neq j\}$	$\forall d \in D_u : F_u(d) = 1200$ m/s
Skewed workload	
$D_s = \{\{g_1, g_2\}, \{g_3, g_4\}\}$	$\forall d \in D_s : F_s(d) = 9000$ m/s

Table 3.2. Uniform and skewed workloads.

Uniform workload			
$T_u(\mathscr{T}_2, h_1) = D_u$	$L_u(\mathscr{T}_2, h_1) = 7200$ m/s	$\sum_{d \in D_u} H(\mathscr{T}_2, d) = 12$	Best choice (lowest heights)
$T_u(\mathscr{T}_3, h_1) = D_u \setminus \{\{g_1, g_2\}, \{g_3, g_4\}\}$ $T_u(\mathscr{T}_3, h_2) = D_u \setminus \{\{g_3, g_4\}\}$ $T_u(\mathscr{T}_3, h_3) = D_u \setminus \{\{g_1, g_2\}\}$	$L_u(\mathscr{T}_3, h_1) = 4800$ m/s $L_u(\mathscr{T}_3, h_2) = 6000$ m/s $L_u(\mathscr{T}_3, h_3) = 6000$ m/s	$\sum_{d \in D_u} H(\mathscr{T}_3, d) = 16$	Poor choice
Skewed workload			
$T_s(\mathscr{T}_2, h_1) = D_s$	$L_s(\mathscr{T}_2, h_1) = 18000$ m/s	$\sum_{d \in D_s} H(\mathscr{T}_2, d) = 4$	Not viable (exceeds capacity)
$T_s(\mathscr{T}_3, h_1) = \emptyset$ $T_s(\mathscr{T}_3, h_2) = \{\{g_1, g_2\}\}$ $T_s(\mathscr{T}_3, h_3) = \{\{g_3, g_4\}\}$	$L_s(\mathscr{T}_3, h_1) = 0$ m/s $L_s(\mathscr{T}_3, h_2) = 9000$ m/s $L_s(\mathscr{T}_3, h_3) = 9000$ m/s	$\sum_{d \in D_s} H(\mathscr{T}_3, d) = 4$	Best choice

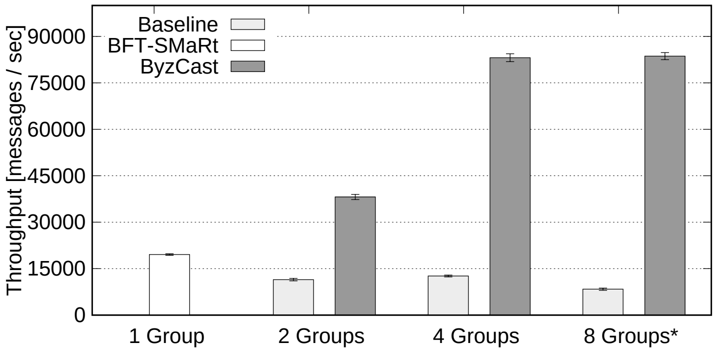
Table 3.3. Optimization model outcomes for uniform and skewed workloads.

3.4.3 Overlay tree versus workload

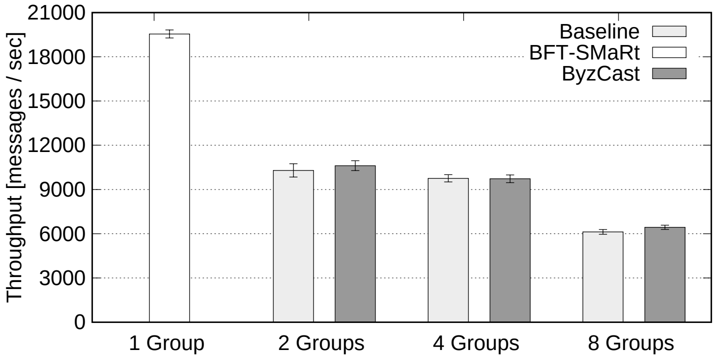
This experiment assesses how the workload and the performance of groups affect the overlay tree. The system has four target groups and up to three auxiliary groups subject to two workloads. In both workloads only global messages are considered since local messages are multicast directly to target groups and do not affect the tree layout. In the *uniform workload,* clients multicast messages to two groups and all combinations of destinations have an equal probability of being chosen. In the *skewed workload,* clients multicast messages to either groups $\{g_1, g_2\}$ or to $\{g_3, g_4\}$. Moreover, the load in the skewed workload is higher. Table 3.2 details the two workloads. Based on the experiments reported in §3.4.4, an auxiliary group can sustain approximately 9500 messages/sec (i.e., $K(h_i) = 9500$ m/s).

Table 3.3 shows outcomes for the two workloads with two-level ($\mathscr{T}_2$) and three-level ($\mathscr{T}_3$) trees (for the three-level tree depicted in Figure 3.1). For the uniform workload, a two-level tree is the best option since the root can sustain the load (i.e., $L_u(\mathscr{T}_2, h_1) < K(h_1)$) and the sum of heights is lower than in the three-level tree (12 instead of 16). For the skewed workload, a two-level tree would impose too high a load on the root (i.e., $L_s(\mathscr{T}_2, h_1) > K(h_1)$) and therefore it is not a viable solution. In this case, in a three-level tree the traffic is divided among the two branches of the tree (h_2 and h_3).

Figure 3.3 exhibits the experimental results in terms of throughput and latency Cumulative Distribution Function (CDF) for each scenario. For the uniform workload, the average latency with a two-level tree is lower than with a three-level tree, although about 55% of messages have lower latency. This happens because the three-level tree distributes the load more uniformly among inner groups. In the skewed workload, the high load on the root of the two-level tree leads to much higher latencies than the three-level tree. The experiments presented next (both LAN and WAN) use the 2-level tree.



(a) Local messages.

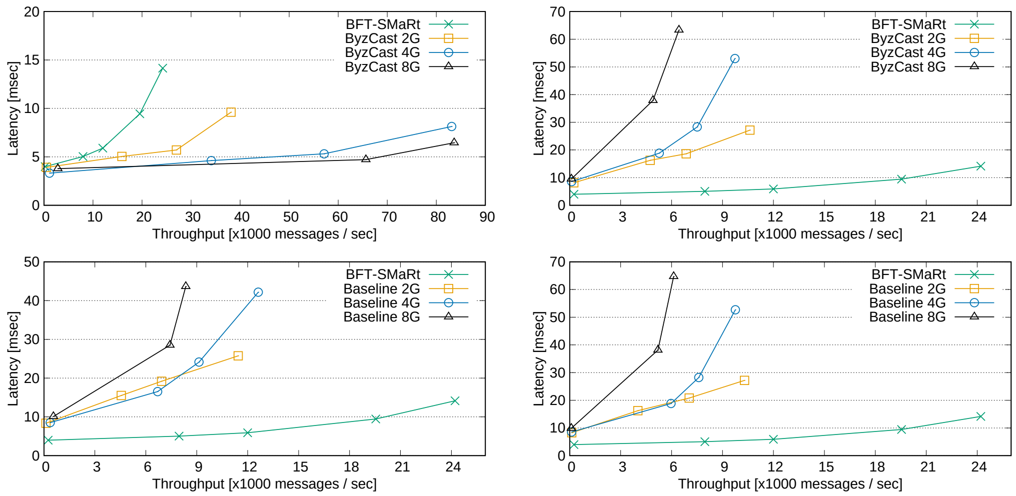


(b) Global messages.

Figure 3.4. Throughput in a LAN. Whiskers show 95% confidence interval.

3.4.4 Scalability of ByzCast in LAN

This experiment assesses the performance of ByzCast and compares it to BFT-SMaRt (using a single group) and to Baseline, a non-genuine atomic multicast approach. Figure 3.4(a) shows the throughput in messages per second versus the number of groups, when 200 clients per group multicast local messages only (except for the 8-group setup where there are 100 clients per group since there are not enough client nodes to deploy 200 clients per group without saturating those nodes). The results show that the genuineness of ByzCast with respect to local messages pays off. The throughput scales linearly with the number of groups with respect to BFT-SMaRt (single group), delivering more than 83000 messages/sec with 4 groups and 200 clients per group and the same with 8 groups and 100 clients per group. This happens because, for single-group messages, ByzCast only involves the sender of a message and the destination target group. Since a single group must order all the messages with the Baseline protocol, it becomes nearly saturated with 400 clients. Thus, the performance with four groups is only slightly higher than with two groups, from 11000 to 12000 messages/sec), and even smaller with eight groups. Figure 3.4(b) shows that ByzCast's throughput when all the clients multicast global messages only is at most half the throughput of BFT-SMaRt: 9700 messages/sec against 19500 messages/sec in the best case. Differently from BFT-SMaRt, a global message in ByzCast has to be ordered by both the auxiliary group and the target groups, impacting the message latency and the overall throughput. The same observation holds for the Baseline protocol, which behaves similarly to ByzCast.



(a) Local messages: ByzCast (top) and Baseline (bottom).

(b) Global messages: ByzCast (top) and Baseline (bottom).

Figure 3.5. Throughput vs. latency in a LAN.

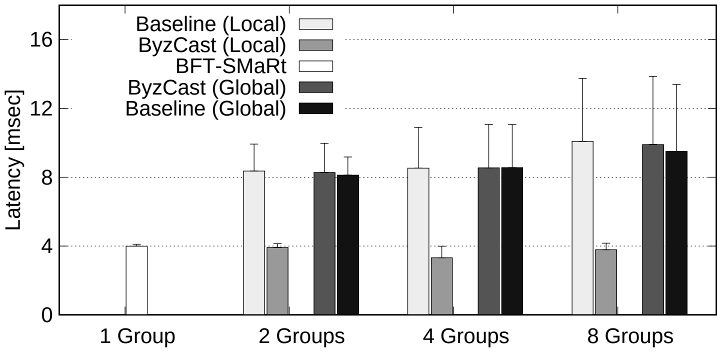


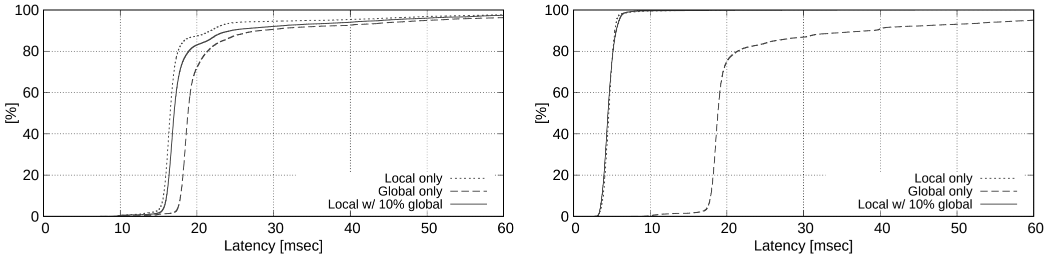
Figure 3.6. Single-client latency in a LAN. Bars show median latency and whiskers show 95-th percentile.

3.4.5 Throughput versus latency in LAN

Figure 3.5(a) shows how the mean latency behaves as the number of clients increases. ByzCast (top) is at least twice as fast and has half the Baseline's latency even with only 2 groups. In executions where all request are global messages, even for small number of clients, BFT-SMaRt has always the best performance, as depicted in Figure 3.5(b). This results reinforces the observation that an atomic broadcast (BFT-SMaRt) is preferable over an atomic multicast when most messages are global [74]. ByzCast and Baseline for 2, 4 and 8 groups perform very alike and the latency saturates with less than half BFT-SMaRt's throughput.

3.4.6 Latency without contention in LAN

The next experiments assess latency with a single client. This setup aims to check how the protocols perform in the absence of contention or queuing effects. The configurations have an increasing number of groups with both local and global messages. Figure 3.6 shows that regarding local messages ByzCast performs as well as BFT-SMaRt no matter the number of groups, with latency around 4 msec. The fact that groups do not interact with each other when ordering local messages guarantees this expected latency. Global messages have twice the latency of local messages in ByzCast because they go through the auxiliary group before reaching the target groups. Besides, global messages latency increases slightly as more target groups are added because replicas in the auxiliary group need to perform multiple broadcasts to all the groups in message destination.



(a) Baseline *(b)* ByzCast

Figure 3.7. Latency CDF with 10% of global messages.

3.4.7 Performance with mixed workload in LAN

The last experiment in LAN assesses the performance of ByzCast with both local and global messages. In a 2-level overlay tree with 4 target groups, 160 equally distributed clients multicast local and global messages in a proportion of 10:1. Figure 3.7 shows the latency CDF for both Baseline and ByzCast. Since in the Baseline protocol (Figure 3.7(a)) all messages are ordered in the same auxiliary group before reaching the target group(s), the latency for both local and global messages are similar. ByzCast, on the contrary, is genuine for local messages, which have a considerably smaller latency up to the 99.5-th percentile, as exhibited in Figure 3.7(b). For global messages, ByzCast and Baseline have similar performance. It is worth noticing that in ByzCast local messages do not suffer from the "convoy effect", a phenomenon in which the slower ordering of global messages can impact the latency of local ones [76]. In fact, the local-message latency CDF for ByzCast with 10% of global messages is very similar to the latency for 100% local messages.

3.4.8 Latency without contention in WAN

The first experiment in WAN measures the latency of ByzCast without any queuing effect or resources overload. A single client from each region multicasts local and global messages in a closed loop. The conclusions, shown in Figure 3.8, are similar to those presented for a LAN. ByzCast has latency as good as a single group (BFT-SMaRt) for local messages and twice the value for global ones. In ByzCast, clients multicast global messages via an auxiliary group that totally orders all messages before broadcasting them to target groups, what explains the doubled latency. The Baseline protocol pays this double ordering for every message.

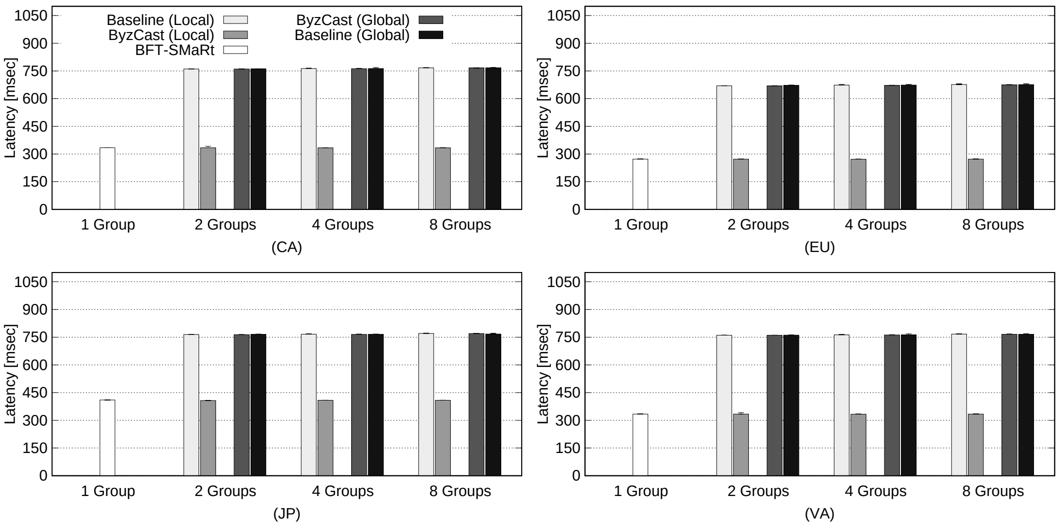


Figure 3.8. Latency with single client in WAN. Bars show median latency and whiskers represent the 95-th percentile.

3.4.9 Performance with mixed workload in WAN

The last experiment evaluates ByzCast with a mix of local and global messages in a proportion of 10:1, which would represent a more realistic workload. The setup comprehends 4 target groups, 1 auxiliary group to order global messages, and 40 clients per target group equally distributed among the 4 geographical regions. The results presented in Figure 3.9 shows that ByzCast is 2x to 3x faster than the Baseline protocol in terms of throughput. Figure 3.10 shows the latency CDF for global and local messages. As expected, ByzCast has local latency 2x to 4x smaller than the values for the Baseline protocol. Regarding global messages, both protocols behave similarly as exposed by previous experiments in LAN and WAN. The latency CDF also confirms that ByzCast does not suffer from the convoy effect, as the local latency is stable even in the presence of global messages.

3.5 Related work

ByzCast is at the intersection of two topics: atomic multicast (§3.5.1) and BFT protocols (§3.5.2).

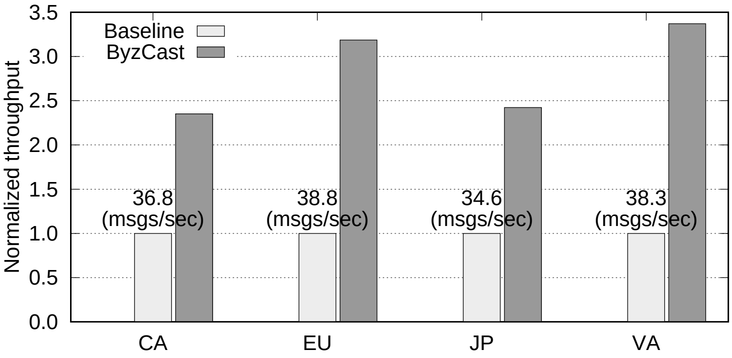
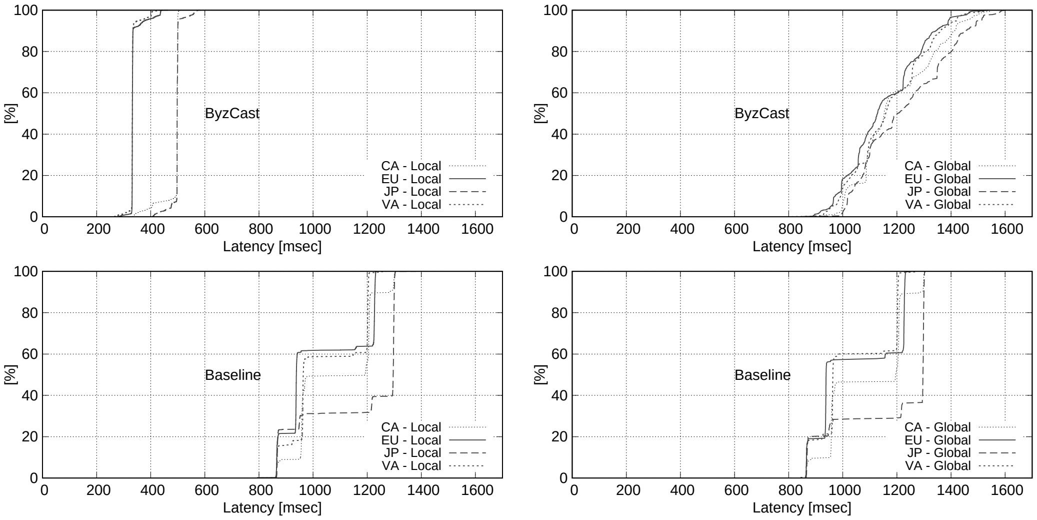


Figure 3.9. Normalized throughput with mixed workload in a WAN.



(a) Local messages. *(b)* Global messages.

Figure 3.10. Latency CDF with 40 clients per group and 10% of global messages.

3.5.1 Atomic Multicast

Several multicast and broadcast algorithms have been proposed [31]. Moreover, many systems use "ad hoc" ordering protocols that do not implement all the properties of atomic multicast (e.g., [28, 44, 79]). Since no atomic multicast algorithm exists for Byzantine failures, the discussion about atomic multicast for benign failures presented in §2.5 also relates to ByzCast.

3.5.2 Scalable BFT

Despite the large amount of work on BFT replication in the last two decades (e.g., [1, 5, 12, 21, 36, 45, 53, 58, 81, 85]), the scalability of BFT protocols is still a relatively unexplored topic, which is discussed in this section.

A common observation of BFT protocols is that their performance degrades significantly as the number of faults tolerated increase [1]. This lack of *fault-scalability* comes mostly from the all-to-all communication used in these protocols, which implies in a quadratic amount of messages. This limitation can be mitigated either by using protocols with linear message pattern [1, 36, 45], by using protocols with a smaller ratio between n and f [53, 85], or by exploring erasure codes and large message batches [58]. Independently on the trade-offs explored by these protocols, all of them lose performance as the number of replicas increase, contrary to ByzCast.

There are few BFT protocols that target wide-area networks [5, 81]. These protocols tend to use more replicas to decrease the relative quorum size or the distance between replicas in the quorums. Similarly to the scalable protocols described before, the performance of these protocols tends to decrease with the number of replicas.

The natural way of scaling replicated systems is sharding the state in multiple replica groups and running ordering protocols only in these groups. To the best of our knowledge, there are only three works that consider partitionable replication for BFT systems. Augustus [63] and Callinicos [62] introduces protocols for executing transactions in multiple shards of a key-value store implemented on top of multiple BFT groups. A recent work by Nogueira et al. [60] introduces protocols for splitting and merging replica groups in BFT-SMaRt, without discussing ways to disseminate messages to more than one of these groups with Byzantine failures. ByzCast complements these works by providing a protocol for disseminating requests on multiple partitions, enabling thus the efficient support for services that require multi-partition operations.

3.6 Conclusion

Atomic multicast is a fundamental communication abstraction in the design of scalable and highly available strongly consistent distributed systems. This chapter presents ByzCast, the first Byzantine Fault-Tolerant atomic multicast, designed to build on top of existing BFT abstractions. ByzCast is partially genuine, i.e., it scales linearly with the number of groups, for messages addressed to a single group. In addition to introducing a novel atomic multicast algorithm, its performance is also assessed in two different environments. The results show that ByzCast outperforms BFT-SMaRt in most cases, as well as a non-genuine BFT atomic multicast protocol.

Chapter 4

Speeding up state machine replication in wide-area networks

4.1 Introduction

Many current online services must serve clients distributed across geographic areas. In order to improve service availability and performance, servers are typically replicated and deployed over geographically distributed sites (i.e., datacenters). By replicating the servers, the service can be configured to tolerate the crash of nodes within a single datacenter and the disruption of an entire datacenter. Geographic replication can improve performance by placing the data close to the clients, which reduces service latency.

Designing systems that coordinate geographically distributed replicas is challenging. Some replicated systems resort to weak consistency to avoid the overhead of wide-area communication. Strong consistency provides more intuitive service behavior than weak consistency, at the cost of increased latency. Due to the importance of providing services that clients can intuitively understand, several approaches have been proposed to improve the performance of geo-distributed strongly consistent systems (e.g., [44, 59, 78, 83]). This chapter presents GeoPaxos, a protocol that combines three insights to implement efficient state machine replication in geographically distributed environments.

First, *GeoPaxos decouples ordering from execution* [87]. Although Paxos introduces different roles for the ordering and execution of operations (acceptors and learners, respectively [48]), Paxos-based systems typically combine the two roles in a replica (e.g., [44, 59, 67]). Coupling order and execution in a geographically distributed setting, however, leads to a performance dilemma. On the one hand, replicas must be deployed near clients to reduce latency (e.g., clients can quickly read from a nearby replica). On the other hand, distributing replicas across geographic areas to serve remote clients slows down ordering, since replicas must coordinate to order operations. By decoupling order from execution, GeoPaxos can quickly order operations using servers in different datacenters within the same region [44] and deploy geographically distributed replicas without penalizing the ordering of operations.

Second, instead of totally ordering operations before executing them, as traditionally done in state machine replication [47], *GeoPaxos partially orders operations*. It is well-known that state machine replication does not need a total order of operations [77] and a few systems have exploited this fact (e.g., [46, 59]). GeoPaxos differs from existing systems in the way it implements partial order. GeoPaxos uses multiple independent instances of Multi-Paxos [23] to order operations—hereafter, an instance of Multi-Paxos is called a Paxos group or simply a group. Operations are ordered by one or more groups, depending on the objects they access. Operations ordered by a single group are the most efficient ones since they involve servers in datacenters in the same region. Operations that involve multiple groups require coordination among servers in datacenters that may be far apart and thus perform worse than single-group operations. GeoPaxos' approach to partial order can take advantage of public cloud computing infrastructures such as Amazon EC2 [4]: fault tolerance is provided by nodes in datacenters in different *availability zones*, within the same region; performance is provided by replicas in different regions. Although intra-region redundancy does not tolerate catastrophic failures in which all datacenters of a region are wiped out, most applications do not require this level of reliability [44].

Third, to maximize the number of single-group operations, *GeoPaxos exploits geographic locality*. Geographic locality presumes that objects have a preferred site, that is, a site where objects are most likely accessed. Geographic locality is common in many online services. For example, operations on a user's data often originate in the region where the user is. Some distributed systems exploit locality by sharding the data and placing shards near the users of the data (e.g., [44, 83]). GeoPaxos does not shard the service state; instead, it distributes the responsibility for ordering operations to Paxos groups deployed in different regions. Operations are ordered by the groups in the preferred sites of the objects accessed by the operation.

The rest of the chapter is structured as follows. Section 4.2 details the system model and recalls fundamental notions. Section 4.3 overviews the main contributions. Section 4.4 details GeoPaxos. Section 4.5 discuss the correctness of the provided algorithm. Section 4.6 describes the prototype. Section 4.7 presents performance evaluation. Section 4.8 reviews related work and Section 4.9 concludes the chapter.

4.2 System model specifics

Besides the system model defined in §1.2, additional assumptions related to GeoPaxos are provided next. The considered system ia a message-passing geographically distributed system. Client and server processes are grouped within *datacenters* (also known as *sites* or *availability zones*) distributed over different *regions*. The system is asynchronous in that there is no bound on message delays and on relative process speeds, but communication between processes within the same region experience much shorter delays than communication between processes in different regions.

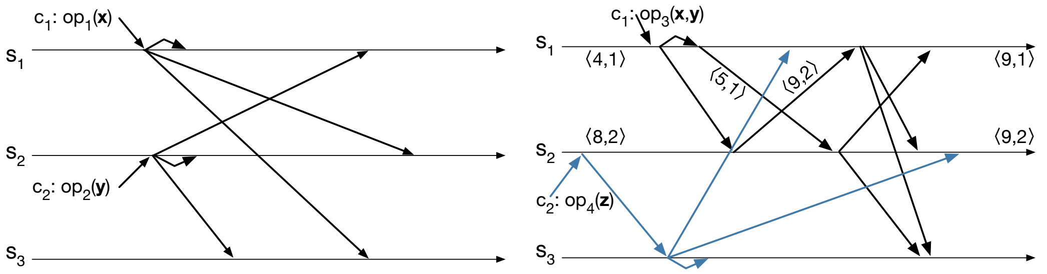
Processes are subject to crash failures and do not behave maliciously (e.g., no Byzantine failures). Service state can be replicated in servers in datacenters within the same region and across regions. The approach taken to replication is aligned with cloud environments [4, 44]: Replication in different datacenters in the same region is used to tolerate failures. Replication across regions is mostly used to explore locality. The system also accounts for client-data proximity by assuming that objects have a *preferred site* [83]. For this purpose, each object has an attribute that indicates its preferred site. The preferred site of an object may be initially unknown by the system and change over time (e.g., as users change their physical location).

4.3 Overview

This section explains how GeoPaxos implements partial order in the absence of failures (§4.3.1), optimizes performance (§4.3.2), and tolerates failures (§4.3.3). It concludes discussing GeoPaxos execution model (§4.3.4).

4.3.1 Partial ordering of operations

GeoPaxos explores the fact that operations that access disjoint sets of objects (i.e., commutative operations) do not need to be executed in the same order by the replicas [77]. Consider the example in Fig. 4.1(a) where two clients invoke operations op_1 and op_2 that access objects x and y, respectively. In replica s_1, the operation on x is executed before the operation on y, while in s_2 and s_3 the



(a) Commutative operations. *(b)* Operations ordered by multiple replicas.

Figure 4.1. Simple GeoPaxos executions. Replicas s_1, s_2 and s_3 are *preferred sites* of objects x, y and z respectively.

opposite happens. Despite the different execution order, all replicas produce the same output for the operations.

In typical implementations of state machine replication (e.g., [48, 59]), a quorum of replicas must order operations before the operations can be executed. The responsibility for ordering operations is assigned to replicas at the preferred sites of the objects accessed in the operation. For example, in Fig. 4.1, s_1, s_2 and s_3 are the preferred sites for objects x, y and z, respectively. So, when s_1 receives operation $op_1(x)$ (see Fig. 4.1(a)), it can order and propagate the operation to the other replicas, but when s_2 receives $op_4(z)$ from c_2 (see Fig. 4.1(b)), s_2 forwards the operation to s_3 to be ordered.

For operations that access multiple objects, there are two possibilities: (i) all the objects accessed by the operation share the same preferred site or (ii) the objects' preferred sites are different. In case (i), the operation is ordered by the replica in the only preferred site, as described above. In case (ii), the involved replicas need to coordinate to decide the final order of the operation, as explained next.

Replicas assign unique timestamps to operations and execute operations in timestamp order.[1] In the case of an operation that involves multiple replicas, the involved replicas must agree on the operation timestamp. In Fig. 4.1(b), client c_1 sends operation op_3 to s_1. Since op_3 accesses objects x and y, whose preferred sites are s_1 and s_2, s_1 forwards op_3 to s_1 and s_2, which assign their next available timestamp to op_3. After exchanging assigned timestamps, op_3's final timestamp is computed as the maximum between $\langle 5,1\rangle$ and $\langle 9,2\rangle$. Replicas s_1, s_2 and s_3 can execute op_3 as soon as (1) they have received op_3's final timestamp from the replicas involved in the ordering procedure, and (2) they have executed all non-commuting operations with smaller timestamp than op_3's final timestamp. Since op_3 and op_4 commute, they can be ordered by disjoint sets of replicas. Further details about GeoPaxos ordering algorithm are provided in §4.4.

---

[1] A timestamp is a tuple $\langle i,j\rangle$, where i is an integer and j a unique id associated with each replica. For any two timestamps $\langle i,j\rangle$ and $\langle k,l\rangle$, $\langle i,j\rangle < \langle k,l\rangle$ if (a) $i<k$ or (b) $i=k$ and $j<l$.

4.3.2 Optimizing performance

The preferred site of objects plays a central role in GeoPaxos performance. First, it is important to maximize *locality*, i.e., assign the object preferred site to a replica close to clients that access the object more often. Besides, single-replica operations are the most efficient ones in GeoPaxos: (a) they involve no inter-replica communication in the ordering process and scale with the number of replicas; (b) they can be ordered and executed independently and concurrently by different replicas. By taking into account the *affinity* between objects (i.e., how often objects are accessed in the same operation), single-replica operations can be optimized. Finally, when defining object preferred site, one should strive to keep the *load* on the replicas equally distributed. Ideally, the system should seek to achieve a balance among locality, affinity and load, as discussed in §4.4.2.

4.3.3 Fault tolerance

The protocol presented in §4.3.1 does not tolerate replica failures. If replica s_1 in Fig. 4.1(b) fails, for instance, then all clients that depend on objects ordered by s_1 will block until s_1 recovers. The solution to avoid blocking in the event of failures is inspired by Spanner [16]. In Google's distributed database, transactions that span multiple shards are committed using two-phase commit (2PC), a blocking protocol. Spanner avoids blocking by replicating each shard within a Paxos group. Such an approach tolerates the failure of a minority of replicas in each Paxos group.

GeoPaxos relies on Paxos groups to replicate the logic responsible for ordering operations. This essentially splits the roles of ordering from execution at replicas. Moreover, GeoPaxos explores the different Paxos roles to split ordering from execution, performed by acceptors and learners, respectively. Fig. 4.2 exhibits a GeoPaxos deployment with three Paxos groups in regions A, B and C. Different Paxos groups can order operations concurrently and replicas (learners) execute all operations in the order established by the acceptors of each group. Such deployment can tolerate the failure of one datacenter, a failure model similar to Spanner's.

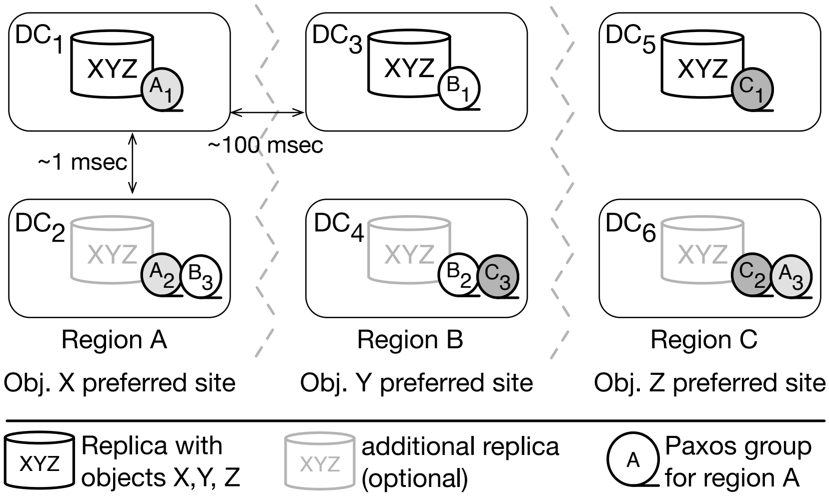

DC1
XYZ
A1
DC3
XYZ
B1
DC5
XYZ
C1
~1 msec
~100 msec
DC2
XYZ
A2
B3
DC4
XYZ
B2
C3
DC6
XYZ
C2
A3
Region A
Obj. X preferred site
Region B
Obj. Y preferred site
Region C
Obj. Z preferred site
XYZ
Replica with objects X,Y, Z
XYZ
additional replica (optional)
A
Paxos group for region A

Figure 4.2. A deployment of GeoPaxos within six datacenters (DC_i) in three regions.

4.3.4 Execution Model

While GeoPaxos shares Spanner's failure model, the two systems differ in a fundamental manner: Spanner supports *read-write objects* (i.e., storage) and GeoPaxos supports *read-modify-write* objects (i.e., state machines). In Spanner, clients can build complex applications by issuing multiple read and write requests to servers and encapsulating multiple requests into a transaction, which can also contain client-side arbitrary computation on the data. GeoPaxos supports state machines: the application logic runs entirely at the replicas; clients simply invoke the operations (similarly to database stored procedures).

From a broader perspective, Spanner executes transactions, possibly spanning multiple shards, and then coordinates the involved shards by means of 2PC; in GeoPaxos, operations are first ordered and then executed. These two approaches have important differences. First, transactions in Spanner's execute-coordinate model may abort, which never happens in GeoPaxos' order-execute model. The order-execute model requires deterministic execution, so that replicas reach the same state after executing the same sequence of operations.

Additionally, GeoPaxos allows commutative operations to execute concurrently, in any order. Even though EPaxos [59], M2Paxos [67], and Mencius [55] can also exploit commutative operations to improve performance, they have to contact at least a majority quorum of distant replicas for each operation. In other words, quorum-based systems where replicas participate in both the ordering and execution are vulnerable to the performance dilemma described in

§4.1. GeoPaxos is empirically compared to this class of quorum-based protocols in §4.7.

4.4 Design

In this section, we detail the GeoPaxos protocol (§4.4.1), present some extensions and improvements to the basic protocol (§4.4.2), and discuss practical aspects (§4.4.3).

4.4.1 The ordering protocol

In GeoPaxos, clients can send operations to any replica. An operation op received by a replica has two main attributes: (a) the destination $op.dst$, which contains the preferred sites of the objects accessed in *op*; and (b) the timestamp $op.ts$, which defines the order in which op must be executed by the replicas. A preferred site corresponds to a Paxos group. Note that only the groups representing the preferred sites of the objects accessed in *op* coordinate to order *op*, but all replicas execute *op*.

Algorithm 4 presents GeoPaxos, divided in atomic tasks. Whenever a client submits an operation op for execution to a replica r, the latter invokes $psite(op)$, which defines $op.dst$ as the set of preferred sites from every object accessed by op, and proposes op to each group in $op.dst$ in a STEP1 message, as detailed in Task 0. The proposing of an operation triggers an execution of Paxos in each group in $op.dst$.

In Task 1, when a replica in $op.dst$ learns about a new STEP1 message addressed to a group h (line 13), it increments h's local clock, updates $op.ts$, and appends op to the set of unordered messages. When r learns STEP1 messages from each group in $op.dst$, it proposes op final timestamp to its local group g (the group it belongs to) within a STEP2 message (line 18).

When r learns a STEP2 message from a group h (the decide event in line 19), it updates h's clock to the maximum between the previous value and the one just decided. As soon as replicas learn STEP2 messages from all groups in $op.dst$, they consider the operation ordered and move it from the *ToOrder* set to the *Ordered* set.

An operation in the *Ordered* set can be executed by the replica in Task 2 when it has a timestamp smaller than any other operations with common destination in both *ToOrder* and *Ordered* sets.

Algorithm 4 replica r in group g.

```
 1: Initialization
 2:    $clock[N] \leftarrow \{\langle 0,1\rangle,\cdots,\langle 0,g\rangle,\cdots,\langle 0,N\rangle\}$
 3:    $\mathit{ToOrder} \leftarrow \emptyset$; $\mathit{Ordered} \leftarrow \emptyset$

 4: To submit operation $op$:                                        {Task 0}
 5:    $op.dst \leftarrow$ psite($op$)
 6:    $op.ts \leftarrow \langle 0,0\rangle$
 7:    for all $h \in op.dst$ do
 8:       propose$_h$(STEP1, $op$, $-$)

 9: when decide$_h$(S, $op$, $clock'$)                                 {Task 1}
10:    if S = STEP1 then
11:       if $|op.dst| = 1$ then
12:          execute operation $op$
13:       else
14:          inc($clock[h]$)
15:          $op.ts \leftarrow max(op.ts, clock[h])$
16:          $\mathit{ToOrder} \leftarrow \mathit{ToOrder} \cup \{op\}$
17:          if decided$_j$(STEP1, $op$, _) from all $j \in op.dst$ then
18:             propose$_g$(STEP2, $op$, $op.ts$)
19:    else
20:       $clock[h] \leftarrow max(clock[h], clock')$
21:       if decided$_j$(STEP2, $op$, _) from all $j \in op.dst$ then
22:          $\mathit{Ordered} \leftarrow \mathit{Ordered} \cup \{op\}$
23:          $\mathit{ToOrder} \leftarrow \mathit{ToOrder} \setminus \{op\}$

24: while $\exists op \in \mathit{Ordered}$ :                                  {Task 2}
       ($\forall op' \in \mathit{Ordered} \cup \mathit{ToOrder}$ :
          $op \neq op' \wedge op.dst \cap op'.dst \neq \emptyset \Rightarrow op.ts < op'.ts$) do
25:    execute operation $op$
26:    $\mathit{Ordered} \leftarrow \mathit{Ordered} \setminus \{op\}$
```

Operations ordered by a single Paxos group (single-group operations) do not receive a timestamp. Instead, they are immediately forwarded to all replicas to be executed, bypassing any ongoing multi-group operations (line 11). We detail this optimization in §4.4.2.

4.4.2 Extensions and optimizations

In this section, we describe a few optimizations that improve the performance of the basic ordering protocol.

Speeding up single-group operations

In Algorithm 4, replicas deliver single-group operations without assigning them timestamps. This optimization ensures that operations involving a single Paxos group do not have to wait for slower multi-group messages. To understand why, assume that both single- and multi-group messages are assigned timestamps. Let op_s and op_m be two operations that access a common object x (i.e., op_s and op_m do not commute), where op_s is single-group and op_m is multi-group. Thus, op_s and op_m must be executed in the same order by all replicas.

Consider an execution in which op_s's communication round happens between op_m's first and second communication rounds at the replicas in the preferred site for object x. Thus, op_s receives a timestamp greater than op_m's. This means that even if op_s is ready to be executed, it must wait for op_m to be ordered. We call the phenomenon by which the execution of an ordered operation is delayed by the ordering of another operation the "convoy effect". Since there is a significant difference between the response times of single-group and multi-group operations, even a small percentage of multi-group operations in the workload can add substantial delays in the execution of single-group operations. To cope with the convoy effect, we let single-group operations be executed as soon as they are ordered in STEP1 (line 11 of Algorithm 4). Consequently, STEP2 of a multi-group operation does not delay the execution of a single-group operation. Intuitively, this does not violate correctness because op_s and op_m are handled in the same total order within a group, and so, all replicas agree that op_s should be executed before op_m is ordered.

Setting an object's preferred site

Since the preferred site of objects may be unknown until the first accesses to the objects and it may also change over time (e.g., as users change their physical location), reassigning an object's preferred site is inevitable. A practical way to maintain preferred sites is to monitor access to objects, client location, and replica load, and periodically evaluate and reassign groups to object accordingly.

Differently from Spanner, where data needs to be copied in the background to the destination zone by the *placement driver*, in GeoPaxos there is no bulk data transfer involved in reassigning an object preferred site. The task is accomplished by a $set\_psite(object\_id, new\_preferred\_site)$ operation that involves the Paxos groups at the object's current and new preferred sites. After ordered, the operation updates the object's preferred site in all replicas. We exercise this feature of GeoPaxos in §4.7.2 using simple heuristics to assign preferred sites to objects.

4.4.3 Practical considerations

Paxos groups are available as long as there is a majority of non-faulty nodes (acceptors) in the group. Clients connected to a replica that fails can reconnect to any operational replica, possibly in the client's closest region. We experimentally evaluate the effects on performance when clients reconnect to a remote replica in §4.7.2.

To recover from failures, the in-memory state of acceptors must be saved on stable storage (i.e., disk). In GeoPaxos, acceptors can persist their state in both asynchronous or synchronous mode. These modes represent a performance and reliability trade-off: the asynchronous mode is more efficient but can cause information loss if an acceptor crashes before flushing its state to disk. We use asynchronous mode in our evaluation.

As an optimization, multi-group operations (with associated parameters) do not need to be sent to all groups involved in the operation. It is sufficient that one group receives the full operation while the other groups receive only the unique id of the operation, so that the operation can be ordered in all involved groups.

4.5 Proof of correctness

Proposition 18 *If operations op_i and op_j do not commute, then replicas execute them in the same order.*

Proof: Since op_i and op_j do not commute, they access at least one common object. Let PS_i and PS_j be the preferred sites for op_i and op_j (i.e., these are the Paxos groups that will order each operation). Thus, either (i) $PS_i \cap PS_j \neq \emptyset$, that is, the operations are ordered by at least one group in common; or (ii) $PS_i \cap PS_j \neq \emptyset$. Case (ii) is only possible if the objects accessed by op_i and op_j changed their preferred site after the first operation is executed and before the second operation is executed. Without lack of generality, let x be an object accessed by the two operations, and $PS(x)_i$ and $PS(x)_j$ be the preferred sites for x when op_i and op_j are executed, respectively. From the mechanism to reassign the preferred site of an object, both the current and the next preferred sites must be involved in the operation. It is possible to conclude that op_i and op_j are ordered in at least one group, directly, as in case (a), or indirectly, as in case (b).

The claim follows from two facts: (a) for ordered operations (operations in the $Ordered$ set) op_i and op_j either $op_i.tp < op_j.tp$ or $op_j.tp < op_i.tp$; and (b) replicas execute ordered operations in timestamp order. Fact (a) holds since timestamp values are unique and the timestamp of an ordered operation *op* is the maximum among the timestamp values proposed by each one of the destination groups in $op.dst$ (line 20 of the Algorithm 4). Fact (b) holds because when an operation *op* is executed by a replica, there is no operation op' at the replica with a smaller timestamp (Task 2 of Algorithm 4). Moreover, no future operation can have a smaller timestamp than *op*'s timestamp since timestamps are monotonically increasing and the timestamp of each group in $op.dst$ is at least equal to *op*'s (line 19 of Algorithm 4). ☐

Proposition 19 *GeoPaxos is linearizable.*

PROOF: From the definition of linearizability [40], there must exist a permutation π of the operations in any execution of GeoPaxos that respects (i) the real-time ordering of operations as seen by the clients, and (ii) the semantics of the operations. Let op_i and op_j be two operations submitted by clients C_i and C_j, respectively.

There are two cases to consider.

Case (a): op_i and op_j are commutative. Thus, op_i and op_j access different objects and the sets of groups representing preferred sites of the objects involved in each operation are disjoint. Consequently, the execution of one operation does not affect the execution of the other and they can be placed in any relative order in π. Operations op_i and op_j are arranged in π so that their relative order respects their real-time dependencies, if any.

Case (b): op_i and op_j do not commute. It follows from GeoPaxos order property above that replicas execute the operations in the same order. Since the two operations execute in sequence, the execution of the operations satisfies their semantics. It is now shown that the execution order satisfies any real-time constraints among op_i and op_j. Without lack of generality, assume op_i finishes before op_j starts (i.e., op_i precedes op_j in real time). Thus, before op_j is submitted by C_j, op_i has completed (i.e., C_i has received op_i's response). Since op_j is ordered and then executed, the conclusion is that op_i is ordered before op_j.

From the claims above, op_i and op_j can be arranged in π according to their delivery order so that the execution of each operation satisfies its semantics.

The last consideration regards the earlier execution of single-group operations as presented in lines 10 to 12 of Algorithm 4. Because replicas receive both STEP1 and STEP2 messages from a group g by means of consensus (propose$_g$())

and decide$_g$()), they are totally ordered. This means that all replicas will decide$_g$ and execute a single-group operation op_i in some group g in the same order with respect to any multi-group operation op_j which also has g in $op_j.dst$. An operation op_k that does not contain g in $op_k.dst$ is commutative with respect to ip_i; in such case, op_k is not proposed in group g and both operations may appear in any relative order in π. ☐

4.6 Implementation

GeoPaxos was implemented in C.[2] The prototype allows to configure disk access mode, synchronous or asynchronous, by default set to asynchronous. Libpaxos[3] is used as the Paxos library. In GeoPaxos proposers and acceptors are single-threaded processes. To ensure liveness, the system starts with a default distinguished proposer, which exchanges heartbeats with the other proposers to allow progress in the event of a failure. Replicas are multithreaded processes. The learner for each Paxos group is executed as an independent thread and only synchronizes with other learners when an operation addresses multiple Paxos groups. An additional thread handles the requests from the clients and, depending on the operation parameters, sets the destination groups accordingly.

Clients are multithreaded, with each thread usually connected to the closest replica. Operations are submitted in a closed loop, i.e., an operation is only sent after the response for the previous operation is received.

4.7 Evaluation

Experiments are conducted in two environments, a local-area network (LAN) and a public wide-area network (WAN). GeoPaxos is compared to other protocols that also implement state machine replication: Multi-Paxos (implemented with Libpaxos), EPaxos,[4] and M2Paxos.[5] There is no empirical comparison between GeoPaxos and Mencius as Mencius's source code is not publicly available—see §4.8 for an analytical comparison of the protocols.

---

[2] `https://bitbucket.org/paulo_coelho/replica-ssn`

[3] `https://bitbucket.org/sciascid/libpaxos`

[4] The evaluation of EPaxos used the authors' original code, available at `https://github.com/efficient/epaxos` and compiled using Go version 1.6.3.

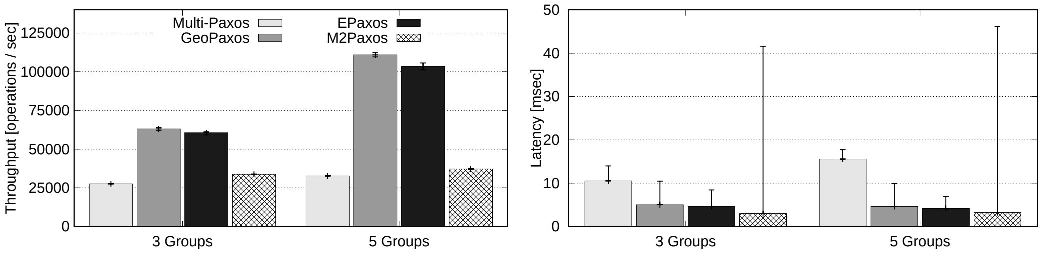
[5] The evaluation of M2Paxos used the authors' original code, available at `https://bitbucket.org/talex/hyflow-go` and compiled using Go version 1.6.3.

4.7.1 Performance in the LAN

The experiments in the LAN allow to assess the protocols in a controlled environment, where it is possible to compare GeoPaxos performance to related protocols (§4.7.1).

Environment

The LAN consists of a cluster of nodes, each one with an 8-core Intel Xeon L5420 processor (2.5GHz), 8GB of memory, SATA SSD disks, and 1Gbps ethernet card. Each node runs CentOS 7.1 64 bits. The RTT (round-trip time) between nodes in the cluster is ∼0.1 msec. GeoPaxos uses three acceptors per Paxos group, with the replica co-located with an acceptor (see Fig. 4.2). For Multi-Paxos, one of the replicas is the coordinator.

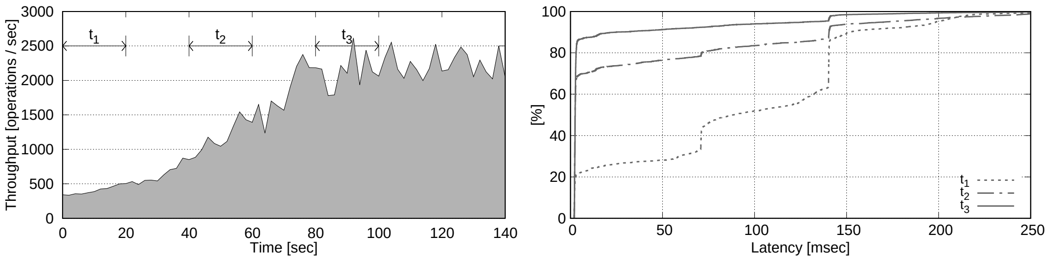


(a) Peak throughput *(b)* Latency

Figure 4.3. Performance in LAN (whiskers: 95% confidence interval for throughput, 99-th percentile for latency).

Benchmark and configuration

The LAN configuration uses a replicated key-value store service. In the workload, all the client requests are 64-byte updates. Clients run in a closed loop and the number of clients is increased until the system is saturated and no increase in throughput is possible. For GeoPaxos, EPaxos and M2Paxos, where the clients are equally distributed among replicas, 100 simultaneous clients per replica are enough to saturate the system. Multi-Paxos saturates sooner, with around 80 clients per replica, which forward the operations to the coordinator. Batching is enabled for Multi-Paxos, EPaxos and GeoPaxos. M2Paxos does not provide batching support.



(a) Throughput *(b)* Latency CDF for highlighted intervals

Figure 4.4. Impact of dynamic preferred site change on throughput and latency.

Throughput and latency

As depicted in Fig. 4.3(a), GeoPaxos and EPaxos have similar performance. This is explained by the absence of leader in EPaxos and the independent ordering in each replica in GeoPaxos. M2Paxos combines clients and replicas in the same process, imposing high CPU usage. As the number of replicas increases and the amount of single-group operations remains high, GeoPaxos throughput is expected to increase linearly, because such operations are ordered by a single Paxos group. Multi-Paxos saturates when the coordinator reaches maximum CPU usage.

Fig. 4.3(b) shows the latency at peak load for all protocols. GeoPaxos and EPaxos have similar results, substantially lower than Multi-Paxos, which suffers the effects of the overloaded coordinator. M2Paxos has the lowest median latency but much larger latency tail.

4.7.2 Performance in the WAN

This section assesses the following aspects: (a) the effectiveness of simple heuristics to set the preferred site of objects (§4.7.2), (b) the latency of the various protocols in a WAN (§4.7.2 and §4.7.2), (c) the convoy effect and mechanism to counter it (§4.7.2), and (d) the performance of GeoPaxos under failures (§4.7.2).

Environment

The WAN configuration uses Amazon EC2, with each replica deployed in a different region. All the nodes are m3.large instances, with 2 vCPUs and 7.5GB of memory. The experiments with 3 replicas uses 2 datacenters in California (CA), 3 datacenters in North Virginia (VA) and 3 datacenters in Ireland (EU). Each region holds three acceptors and one replica (co-located with one acceptor) in different datacenters for fault tolerance in the ordering protocol. The regions of Oregon (OR), with 3 datacenters, and Tokyo (JP), with 2 datacenters, are included to complete the 5 replicas scenario. Table 4.1 summarizes the RTT between these regions. RTT within a datacenter is smaller than 1 msec and between datacenters in the same region below 2.5 msec.

	VA	EU	OR	JP
CA	75	145	22	110
VA	–	75	80	150
EU	–	–	130	215
OR	–	–	–	95

Table 4.1. Average RTT between regions, in milliseconds.

Benchmark and configuration

The WAN configuration uses a social network service. Social networks are notorious for exhibiting locality properties of the sort that GeoPaxos can take advantage of [3, 17, 71]. The social network has 10000 users, each one represented as an object. The friendship relations follow a Zipf distribution with skew 1.5. There are two operations: *getTimeline* and *post*. The *getTimeline* operation returns the last 10 messages posted on a specified user timeline. The *post* appends a message to the timeline of all the followers of the specified user. While *getTimeline* is always a single-group operation, *post* depends on the set of preferred sites of posting user's followers.

The experiments in WAN were executed with 10 clients in each region and a mix of *getTimeline* and *post* operations in a proportion of 4:1. In the WAN experiments, GeoPaxos contains 3 acceptors and 1 replica per region, with the acceptors distributed in different availability zones and the replica co-located with one acceptor in a node (see Fig. 4.2). Multi-Paxos, M2Paxos and EPaxos use one replica per region. Clients (which represent the users) are evenly distributed among the three or five regions, depending on the configuration. When the system starts, the preferred site of each object is set to an arbitrary region (Paxos group).

Defining objects preferred site

This experiment assesses a simple strategy to assign preferred sites to objects in the social network service. The strategy works as follows. To account for locality, the first time an object is accessed (i.e., when a user executes his first *post* or *getTimeline*), the user's preferred site is set to the region of the user so that further accesses will be "local" to the user. Affinity is taken into account by monitoring the frequency of operations involving an object. A user A that follows a user B will have its preferred site set to B's preferred site if B posts more often than A reads his timeline.

Fig. 4.4 shows the throughput in the first 140 seconds of an execution and the latency CDF of three time intervals ($t_1 = (0, 20), t_2 = (40, 60), t_3 = (80, 100)$). Until the system has optimized the preferred site of objects performance is low. Once objects preferred site have been efficiently assigned, something that happens after 75 seconds into the execution, most operations will be local single-group. The following distribution is obtained when the assignments become stable:

- Three regions ordering 3404, 3170 and 3426 users; where 80% of users have followers with the same preferred site, 18% of users have followers ordered in two regions, and 2% in all regions.
- Five regions ordering 1998, 1942, 2057, 1943 and 2060 users; 74% of users have followers ordered in the same preferred site , 22% in two regions, 2.6% in three regions, 1% in four regions, and 0.4% in five regions.

In the experiments presented next, we report results after assignments are stable.

Although conceptually M2Paxos supports the notion of object ownership (equivalent to object preferred site), it was not possible to compare it experimentally to GeoPaxos since the available implementation does not include this feature. Besides, M2Paxos does not support multi-group operations.

Latency in WAN

Fig. 4.5 shows the median latency and 99-th percentile of the protocols in scenarios with a single client. In the "remote client" configurations, the client is in Ireland (EU) and connects to the replica in CA; in all other executions the client is in California (CA). The deployment with a single client assesses the protocols in the absence of queueing effects.

Multi-Paxos, EPaxos and M2Paxos have their latency strictly related to the proximity to other replicas. Multi-Paxos also depends on the location of the clients, because all operations must reach the coordinator in CA. GeoPaxos depends on the number of groups that an operation is addressed to, while proximity to other replicas only impacts operations that access objects ordered by multiple replicas. For single-group operations, the latency of GeoPaxos is around 2 msec,

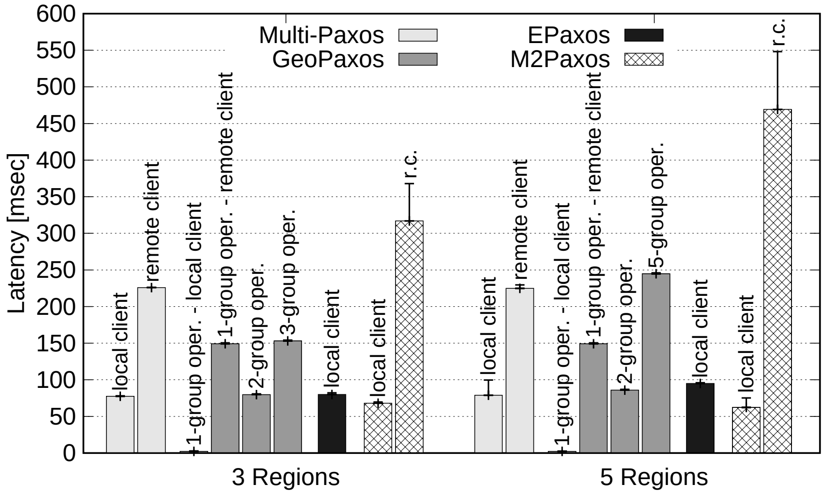


Figure 4.5. Latency in WAN (whiskers: 99-th percentiles).

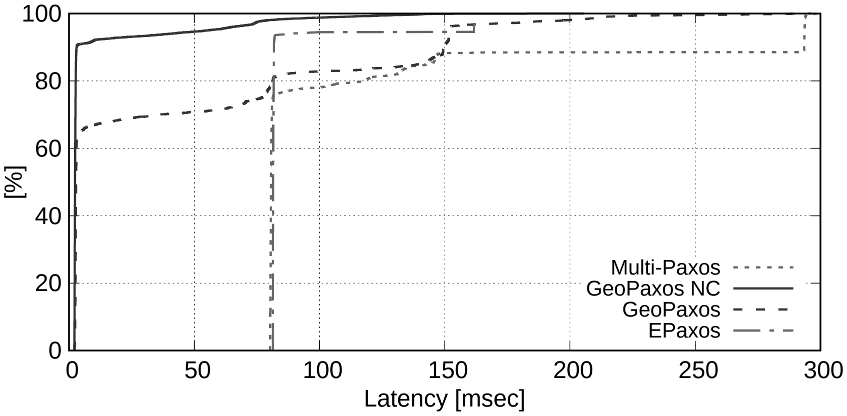
while the best case for EPaxos and Multi-Paxos is around 80–90 msec for both 3 and 5 regions. Such latency is similar to Spanner read-only operations with all datacenters in a single region [44]. No state machine replication system achieves latency as low as GeoPaxos because at least a majority of replicas needs to be contacted. M2Paxos takes around 63 msec to order a message. Even with a single client, M2Paxos has high latency for remote clients, almost twice the value of GeoPaxos's latency in the 5-region scenario. Operations that involve two groups in GeoPaxos have latency between 80 msec and 90 msec. It is worth mentioning that in the 5-region scenario, even though a remote client does not make sense for EPaxos, a client in EU connected to a local replica would have a latency similar to a GeoPaxos remote client invoking single-group operations.

GeoPaxos only has higher latency than other techniques in particular multi-group operations like the situation with all regions, something that is expected to happen very scarcely, depending directly on the latency of the farthest group.

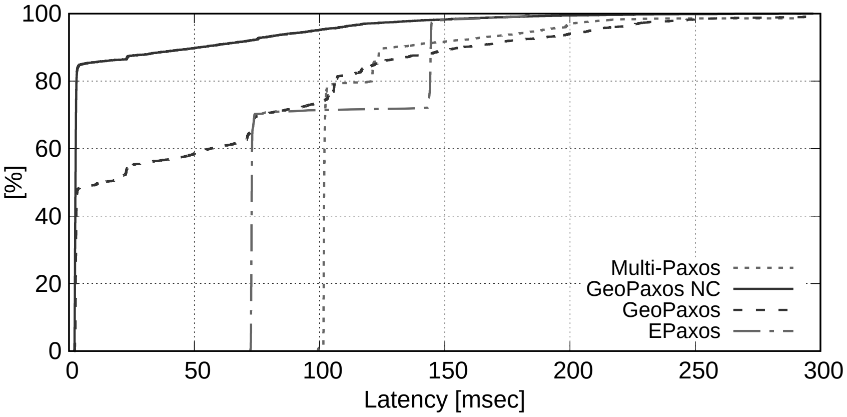
Convoy effect

The following experiment compares GeoPaxos without the optimizations to cope with the convoy effect ("GeoPaxos") and with the optimization described in §4.4.2 to mitigate the convoy effect ("GeoPaxos NC"). EPaxos and Multi-Paxos are also included in the evaluation. M2Paxos is not evaluated in this setup since the available implementation cannot handle multi-group operations.

Fig. 4.6 shows the cumulative distribution functions (CDF) for 3- and 5-region



(a) Latency in WAN and convoy effect with 3 regions.



(b) Latency in WAN and convoy effect with 5 regions.

Figure 4.6. Impact of the convoy effect on latency.

setups. With 3 regions, GeoPaxos NC brings the percentage of low-latency single-group operations from around 65% to more than 90%. With 5 regions, half the single-group operations experience the convoy effect originally and less than 15% are penalized with GeoPaxos NC. Part of the single-group operations that display high latency with GeoPaxos NC are due to queuing effects and CPU scheduling (2 vCPUs in configurations with as many threads as the number of replicas) and a minor fraction of users with followers exclusively external to their own region (single-group operations ordered by a remote client, resulting in a remote client).

Fig. 4.7 exhibits the impact on throughput. GeoPaxos is 2x faster than EPaxos with 3 regions (from 374 to 745 operations/sec) and GeoPaxos NC is 6x faster (2245 operations/second). With 5 regions, the speedup over EPaxos is 3.8x and 7.6x for GeoPaxos and GeoPaxos NC, respectively (366, 1357 and 2767 operations/sec). Multi-Paxos experienced the lowest throughput due to the ordering of operations done by the single coordinator.

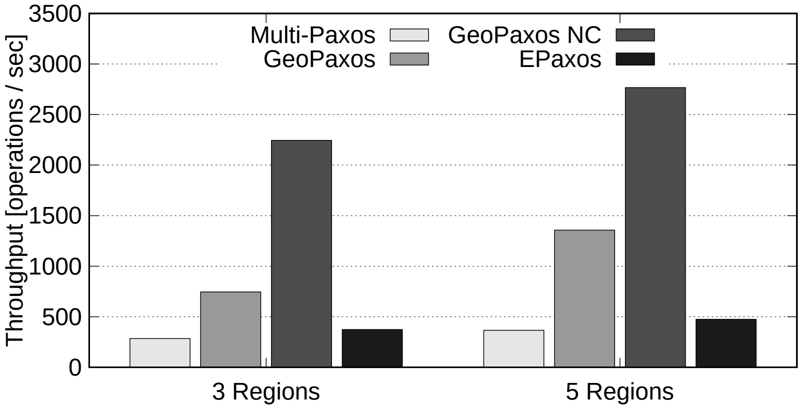


Figure 4.7. Impact of the convoy effect on throughput.

Performance under failures

The last set of experiments assess GeoPaxos in the presence of replica failures. Initially, the system is configured with clients running in closed loop and equally distributed across groups in order to keep the overall throughput between 10000 and 12000 operations per second, without saturating the replicas (see Fig. 4.8).

Clients from 3 EC2 regions (CA, VA and EU) connect to their local replicas. Clients in CA have a backup replica in another datacenter in the same region, while clients in VA are configured to connect to the EU replica in case of failures. After 30 seconds into the execution, a replica in CA is killed. Clients in this region immediately connect to the second replica of the region, which has a slightly greater latency (from 0.4 to 1.2 msec), resulting in some throughput decrease. A second replica is killed 30 seconds later, now in VA. Clients reconnect to the EU replica, but are subject to increased latency (around 140 msec).

The replica in EU keeps a constant throughput despite two failures in two distinct regions. This is possible due to the partial ordering implemented by GeoPaxos and the separation of Paxos roles, disassociating the acceptor and proposer from the replicas (learners).

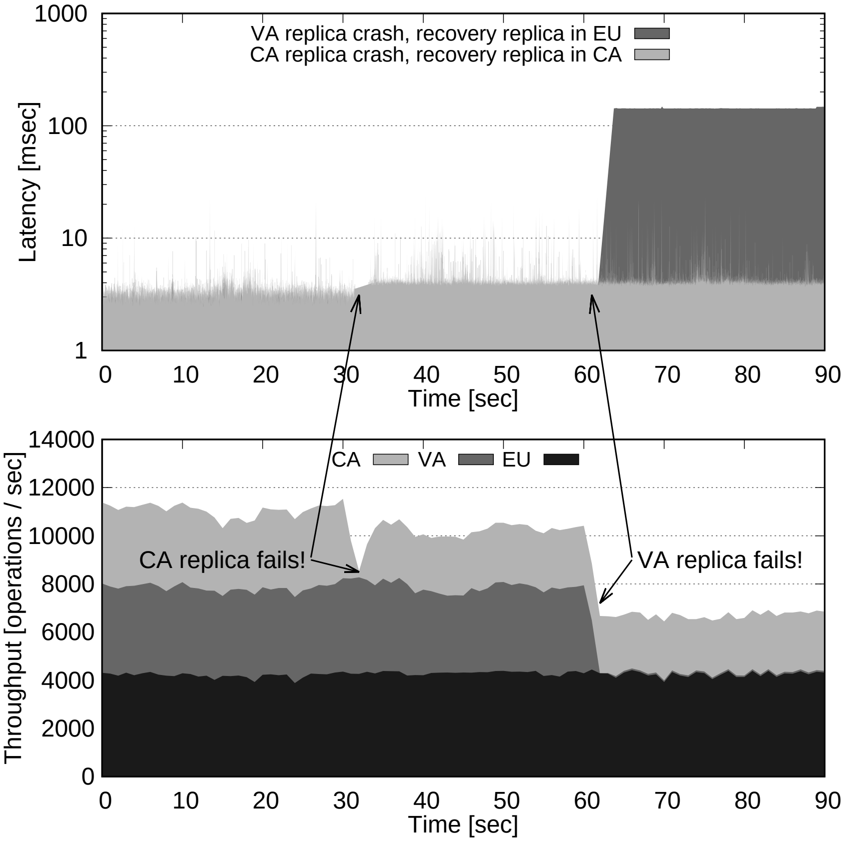


Figure 4.8. Latency and throughput in WAN (3 regions, clients in all regions).

4.8 Related work

If on the one hand state machine replication is widely used to increase service availability (e.g., [20, 42, 44]), on the other hand it is notably criticized for its overhead. From single-leader algorithms, like Paxos [48], to leaderless algorithms (e.g., [59, 84]) and variations that take the semantics of operations into account (e.g., [49, 66]), all efforts have been directed to finding faster ways to order operations. None of the solutions, however, can avoid the large latency imposed by geographically distributed applications since at least a simple majority quorum of replicas is needed to order operations [50]. Furthermore, existing solutions experience reduced performance as the number of replicas increases.

GeoPaxos improves the performance of state machine replication by (a) exploiting the fact that operations do not need a total order; (b) distinguishing ordering from execution; and (c) judiciously choosing the Paxos group that will order operations. Partially ordering operations with the goal of improving performance has been previously implemented by EPaxos, Alvin, Caesar and M2Paxos.

EPaxos [59] improved on traditional Paxos [48] by reducing the overload on the coordinator and allowing any replica to order operations. As long as replicas observe common dependencies set, operations can be ordered in one round-trip (fast decision).

Alvin [84] is a system for managing concurrent transactions that relies on Partial Order Broadcast (POB) to order conflicting transactions. While POB is very similar to EPaxos, its main contribution lies in the substitution of EPaxos complex dependency graph analysis by a set of cycle-free dependencies based on timestamps.

Caesar [8] extends POB and reduces the numbers of scenarios that would impose one additional round-trip (slow decisions). Differently from Alvin and EPaxos, where nodes must agree on operations dependency set, Caesar seeks agreement on a common final timestamp for each operation. This strategy allows fast decisions even in specific cases where dependencies do not match, resulting in better performance as contention increases.

Depending on the interference between operations, however, the rate of slow decisions in EPaxos, Alvin and Caesar increases considerably, a high price to be paid by geographically distributed applications.

M2Paxos [67] is another implementation of Generalized Consensus [49] that does not establish operation dependencies based on conflicts, but, similar to GeoPaxos, maps replicas to accessed objects. M2Paxos guarantees that operations that access the same objects are ordered by the same replica. It needs at least two communications steps for local operations and one additional step for remote operations. M2Paxos's mechanism to handle operations that access objects mapped to multiple replicas requires remapping the involved objects to a single replica. Replicas executing different operations may dispute the same objects indefinitely.

Mencius [55] extends traditional Multi-Paxos with a multi-leader solution that partitions the sequence of consensus instances among geographically distributed replicas to avoid the additional round-trip for clients far from the single-leader. Besides, it provides a mechanism to deal with unbalanced load, allowing one replica to propose a `SKIP` message when there is no operation to be executed in that instance. Mencius also allows out-of-order execution of commutative operations.

Differently from GeoPaxos, none of the existing solutions takes locality into account. Instead, they require a quorum of replicas, and thus cannot avoid inter-region latencies in geo-distributed scenarios. GeoPaxos takes advantage of locality to deliver single-replica operations with intra-region latency. Table 4.2

Protocol	Δ	*Operation type*
Alvin [84]	2	RMW
Caesar [8]	2	RMW
EPaxos [59]	2	RMW
Mencius [55]	2	RMW
M2Paxos [67]	2	RMW
Multi-Paxos [48]	2	RMW
Spanner [44]		
single-shard operation	0	RW
multi-shard operation	2	RW
GeoPaxos		
single-group operation	**0**	**RMW**
multi-group operation	**2**	**RMW**

Table 4.2. Minimum inter-region delays (Δ) and operation type: *read-modify-write* (RMW) vs. *read-write* (RW).

compares the latency of GeoPaxos to other state machine replication-based systems. GeoPaxos optimizes for single-group operations ordered in a region. Other solutions rely either on a classic quorum or a fast quorum of replicas in different regions.

Several solutions that partition (i.e., shard) the data have appeared in the literature. Systems in this category are sometimes referred to as partially replicated systems, as opposed to designs in which each replica has a full copy of the service state, like in GeoPaxos. Spanner [44] is a partitioned distributed database for WANs. It uses a combination of two-phase commit and a TrueTime API to achieve consistent multi-partition transactions. TrueTime uses hardware clocks to derive bounds on clock uncertainty, and is used to assign globally valid timestamps and for consistent reads across partitions. It requires elaborate synchronization mechanisms to keep the clock skew among nodes within an acceptable limit. Furthermore, Spanner provides more restrictive type of operations (read-write objects) than state machine replication (read-modify-write objects).

Spinnaker [69] is similar to the approach presented here. It also uses several instances of Multi-Paxos to achieve scalability. However, Spinnaker does not support operations across multiple Multi-Paxos instances.

Differently from existing sharded systems, where replicas contain only part of the service state, in GeoPaxos each replica contains the entire state. In doing this, GeoPaxos can improve performance without sacrificing the simplicity of the state machine replication approach. Moreover, there is no need to reshard data across nodes and to migrate data across nodes for load balance and in response to failures.

Some systems seek to boost performance by exploiting transactional semantics. The most related to GeoPaxos are MDCC, Granola, Geo-DUR and P-Store. MDCC [46] is a replicated transactional data store that also uses several instances of Paxos. MDCC optimizes for commutative transactions, and uses Generalized Paxos to relax the order of commuting transactions. Granola [28] is a distributed transaction coordination system that relies on clock-based timestamps to provide strong consistency among transactions. It depends on loosely synchronized clocks and needs three communication delays to order multi-repository transactions, in the absence of aborts. P-Store [76] and Geo-DUR [78] are optimized for wide-area networks and have also looked into techniques to reduce the convoy effect. Differently from these systems, GeoPaxos does not require transaction support. For example, GeoPaxos does not need to handle rollbacks in case partitions do not agree on the order of operations [78].

Some solutions have faced the “high latency” of state machine replication by weakening consistency guarantees. One example is eventual consistency [30], which allows replicas to diverge in case of network partitions, with the advantage that the system is always available. However, clients are exposed to conflicts and reconciliation must be handled at the application level. Walter [83] offers Parallel Snapshot Isolation (PSI) for databases replicated across multiple datacenters. PSI guarantees snapshot isolation and total order of updates within a site, but only causal ordering across datacenters. COPS [54] ensures a stronger version of causal consistency: in addition to ordering causally related write operations, it also orders writes to the same data items.

4.9 Conclusion

Online services with clients all over the globe are becoming usual, imposing tough requirements on replicated protocols. Coordinating replicas geographically distributed, while keeping performance at acceptable level is challenging. This thesis proposes GeoPaxos, a protocol that confronts this challenge. First, GeoPaxos decouples order from execution in state machine replication. Second, it imposes a partial order on operations. Finally, GeoPaxos exploits geographic locality and places replicas close to clients. The results reveal that GeoPaxos outperforms state-of-the-art approaches by more than an order of magnitude in some cases.

Chapter 5

Speeding up Paxos

At the core of replicated systems that provide strong consistency, like FastCast and GeoPaxos, servers must solve consensus, a distributed problem in which servers must agree on a value (e.g., the i-th operation to be executed). Many consensus algorithms have been developed in the literature and Paxos [48] is probably the most prominent consensus algorithm proposed to date. In part this is attributed to the fact that Paxos is resilience-optimum (i.e., it ensures progress with a majority-quorum of non-faulty replicas) and delay-optimum (i.e., it requires a minimum number of network delays to reach a decision) [50].

While relying on an algorithm that optimizes resilience and communication is important, quickly ordering operations also requires an efficient implementation of Paxos. Therefore, much effort has been placed on implementing Paxos efficiently. Existing proposals fall in three categories: protocols that take advantage of special network topologies (i.e., overlay networks) [38, 57]; protocols that resort to specialized hardware (e.g., part of Paxos is deployed in the network) [29, 52]; and protocols that exploit the semantics of applications (e.g., if the order of two messages does not matter to the application, they do not have to be ordered) [49, 59, 67]. This chapter presents Kernel Paxos [1], an alternative approach, which does not depend on special network topologies, specialized hardware, or application semantics. The main idea behind Kernel Paxos is simply to reduce the main sources of overhead in traditional Paxos implementations. Since Paxos does not involve complex computation, most of the overhead stems from communication. Kernel Paxos reduces communication overhead in two ways: (a) by eliminating context switches needed to send and receive messages and (b) by avoiding the TCP/IP stack. More concretely, Kernel Paxos eliminates context switches by placing the protocol in the kernel; instead of TCP/IP, Kernel Paxos uses raw Ethernet frames.

---

[1] This work is the result of a collaboration with Emanuele Giuseppe Esposito and it is partially described in his bachelor project.

Kernel Paxos has been fully implemented and its performance carefully assessed. Kernel Paxos is derived from LibPaxos, a popular Paxos library. The performance of Kernel Paxos is compared to LibPaxos and an improved version of LibPaxos. The performed experiments used different message sizes, system load, and communication hardware. The experiments show that Kernel Paxos largely outperforms these libraries.

This chapter describes the following contributions.

- It introduces a number of optimizations to LibPaxos, a popular Paxos library. These optimizations have consistently improved the performance of LibPaxos.
- It describes Kernel Paxos, a kernel-based library that outperforms both LibPaxos and its optimized version.
- It assesses the performance of Kernel Paxos under different conditions and experimentally compares it to LibPaxos and its improved version.

The rest of the chapter is organized as follows. Section 5.1 overviews the Paxos algorithm. Section 5.2 introduces Kernel Paxos. Section 5.3 details the prototype. Section 5.4 describes the experimental evaluation. Section 5.5 surveys related work and Section 5.6 concludes the chapter.

5.1 Background on Paxos

Paxos is a fault-tolerant consensus protocol with important characteristics: it has been proven safe under asynchronous assumptions (i.e., when there are no timing bounds on message propagation and process execution), live under weak synchronous assumptions, and resilience-optimum [48].

Paxos distinguishes the following roles that a process can play: *proposers*, *acceptors* and *learners*. An execution of Paxos proceeds in two phases. During the first phase, a proposer that wants to submit a value selects a unique round number and sends a prepare request to a group of acceptors (at least a quorum). Upon receiving a prepare request with a round number bigger than any previously received round number, the acceptor responds to the proposer promising that it will reject any future prepare requests with smaller round numbers. If the acceptor already accepted a request for the current instance (explained next), it will return the accepted value to the proposer, together with the round number received when the request was accepted. When the proposer receives answers from a quorum of acceptors, it proceeds to the second phase of the protocol.

The proposer selects a value according to the following rule. If no acceptor in the quorum of responses accepted a value, the proposer can select a new value for the instance; however, if any of the acceptors returned a value in the first phase, the proposer chooses the value with the highest round number. The proposer then sends an accept request with the round number used in the first phase and the value chosen to at least a quorum of acceptors. When receiving such a request, the acceptors acknowledge it by sending a message to the learners, unless the acceptors have already acknowledged another request with a higher round number. Some implementations extend the protocol such that the acceptor also sends an acknowledge to the proposer. Consensus is reached when a quorum of acceptors accepts a value.

Paxos ensures consistency despite concurrent leaders and progress in the presence of a single leader. Paxos is resilience-optimum in that it tolerates the failure of up to f acceptors (or replicas) from a total of $2f+1$ acceptors to ensure progress (i.e., a quorum of $f+1$ acceptors must be non-faulty) [50].

5.1.1 Paxos and state-machine replication

In practice, replicated services run multiple executions of the Paxos protocol to achieve consensus on a sequence of values. Multiple executions, or instances, of Paxos chained together are referred as Multi-Paxos [23].

Clients of a replicated service are typically proposers, and propose operations that need to be ordered by Paxos before they are learned and executed by the replicated state machines. These replicas typically play the roles of acceptors and learners. If multiple proposers simultaneously execute the described procedure for the same instance, then no proposer may be able to execute the two phases of the protocol and reach consensus. To avoid scenarios in which proposers compete indefinitely in the same instance, a *leader* process can be chosen. In this case, proposers submit values to the leader, which executes the first and second phases of the protocol. If the leader fails, another process takes over its role. In typical state machine replication implementations one of the replicas assumes the role of leader and proposes operations forwarded by clients.

5.1.2 Optimizations

If the leader identity does not change between instances, then the protocol can be optimized by pre-initializing acceptor state with previously agreed upon instance and round numbers, avoiding the need to send first phase messages [48]. This is possible because only the leader sends values in the second phase of the protocol. With this optimization, consensus can be reached in three communication steps: the message from the proposer to the leader, the accept request from the leader to the acceptors, and the response to this request from the acceptors to the leader and learners. Moreover, the leader pre-initialization (first phase of the protocol) for a further instance could be piggybacked in the second phase message for the current instance. Fig. 5.1 exhibits such optimizations.

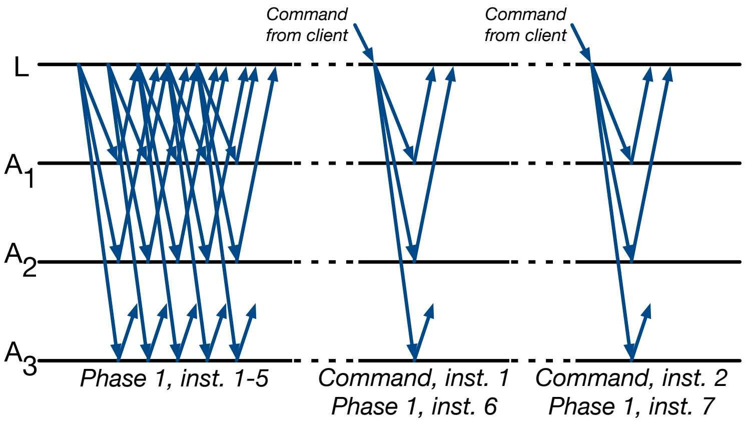


Figure 5.1. Multi-Paxos optimization (example with five pre-initialized instances followed by two client operations).

5.2 Paxos in the kernel

This section motivates the use of in-kernel Paxos (§5.2.1) and describes Kernel Paxos architecture (§5.2.2) and message flow (§5.2.3).

5.2.1 Linux kernel and TCP/IP stack

The Linux kernel is an open-source software that implements the basic functionalities provided by an operating system (e.g., interface to the hardware devices, memory and process management). Due to its openness, the Linux kernel is easily extensible by means of loadable kernel modules (LKM). A module can add support to a new device or introduce a new operating system feature. Besides, it allows separating the minimum core functionalities from on-demand loadable complementary utilities.

Linux implements the traditional user and kernel separation of concerns: user applications run in user-space while kernel-related tasks run in the kernel-space. This separation protects the operating system and the hardware against intentional and unintentional misbehave of applications. Every time a user application wants to send a message through the network, for instance, it uses a system call to context switch from user-space to kernel-space. The system call is necessary to access the network interface card (NIC) and execute the actual sending of data.

A clear trade-off comes from such an approach: if on the one hand the system calls protect the operating system and control the execution of programs, on the other hand every system call involves two context switches, one from user-space to kernel-space so that the call can be executed and another context switch to return from kernel-space to user-space.

Applications that heavily rely on system calls (e.g., I/O-bound applications) suffer the most from the overhead of context switches. In an attempt to measure the impact of context switches in network applications, a simple echo application that runs either in user-space or entirely in kernel-space as a loadable kernel module has been developed. Clients send outstanding UDP messages to a server, which simply replies with the same message it has received from the client. The experiments are executed with increasing number of clients until the point when the throughput stopped increasing. The first line of Table 5.1 shows the peak performance for both the user application and the loadable kernel module.

	User-space application	*Loadable kernel module*
UDP echo	120000 msgs/sec	135000 msgs/sec
Ethernet echo	318000 msgs/sec	**520000 msgs/sec**

Table 5.1. Performance of user-space and kernel-space echo with UDP and Ethernet.

In controlled environments such as local-area networks (LAN), network application performance may also benefit from bypassing the TCP/IP stack, i.e., instead of having IP and UDP headers inside a Ethernet frame to differentiate between application addresses and ports, one can use custom Ethernet types and send application data directly as the Ethernet frame payload. To assess the impact of this modification the initial UDP echo application has been extended to send application data directly on top of Ethernet frames. The second line of Table 5.1 shows the improvement with respect to the UDP version for both user and kernel module versions.[2]

---

[2] The hardware setup used in this evaluation is detailed in §5.4. The source code used in these

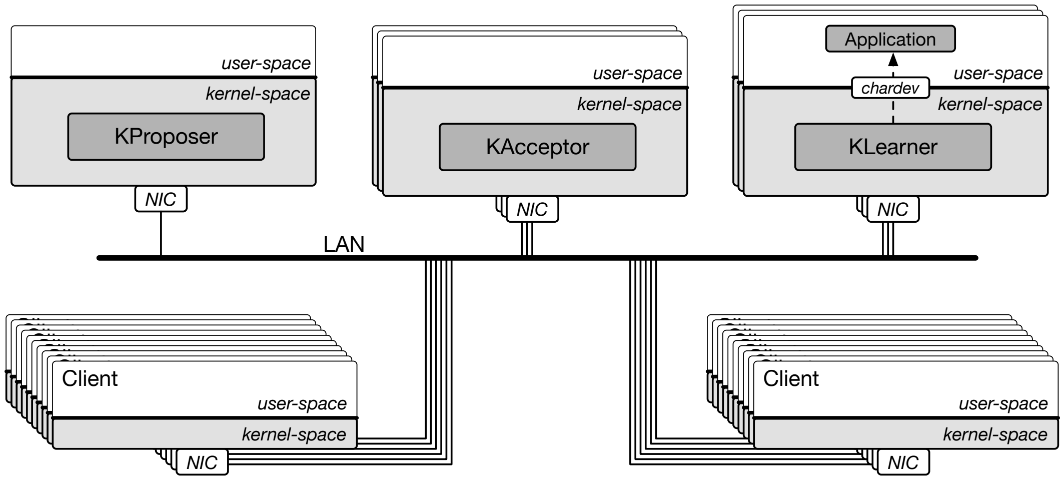

user-space
kernel-space
KProposer
NIC
user-space
kernel-space
KAcceptor
NIC
Application
chardev
user-space
kernel-space
KLearner
NIC
LAN
Client
user-space
kernel-space
NIC
Client
user-space
kernel-space
NIC

Figure 5.2. Kernel Paxos architecture.

While a typical user-space UDP-based echo application can reach peak throughput at 120K messages per second, a kernel-space Ethernet-based system attains 520K messages per second, a 4x performance improvement. These results motivate Kernel Paxos. Moreover, as shown later in this chapter, latency also largely benefits from such an approach.

5.2.2 Kernel Paxos architecture

Paxos relies on a leader for liveness. Therefore, typical Paxos implementations have their performance limited by what the leader can achieve. While some proposals extend Paxos with multi-leader or generalized solutions [49, 59, 67], this work intends to reduce the overhead imposed by the communication layer and the operating system to improve the performance of classic Paxos. Fig. 5.2 shows the proposed architecture.

In Kernel Paxos, each one of the Paxos roles, proposer, acceptor and learner, can run either on the same machine (defined as a *replica* in such case) or deployed in independent nodes, as depicted in Fig. 5.2. Each role runs entirely in kernel space and communicates with the other Paxos players by means of message passing. For the reasons discussed earlier, the system bypasses the TCP/IP stack and exchanges messages right on top of Ethernet frames.

The kernel adopts an event-driven model regarding the reception of Ethernet frames, i.e., the LKM must assign a callback function to each Ethernet type, as

---

experiments is publicly available at `https://github.com/paulo-coelho/kether`.

exhibited in Fig. 5.3. As soon as a new frame arrives, the kernel invokes the corresponding handler automatically. The great advantage of such an approach is that there is no necessity for kernel threads that block while waiting for messages. In fact, each module is a single event-driven kernel thread and no synchronization is required with respect to the data accessed by registered handlers. In Kernel Paxos specifically, each Paxos-related message, like a *promise* or a *prepare* message, is statically associated to an Ethernet type and the corresponding handler function. This way, each Paxos player registers itself to receive the types its roles requires. For example, a KProposer registers to receive both *promise* and *accepted* messages representing the replies from acceptors for phases one and two, besides the *client_value* type for clients requests.

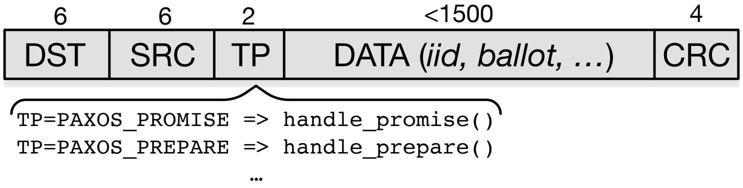

6
6
2
<1500
4
DST
SRC
TP
DATA (*iid, ballot, ...*)
CRC
TP=PAXOS_PROMISE => handle_promise()
TP=PAXOS_PREPARE => handle_prepare()
...

Figure 5.3. Kernel Paxos event-driven approach for Ethernet types.

5.2.3 Message flow in Kernel Paxos

When a KProposer is loaded into the kernel, it contacts a quorum of KAcceptors and pre-executes the first phase for a configurable number of instances. Upon receipt of operations from clients, the KProposer uses such instances to send *accept* messages to the KAcceptors, which, under normal execution, reply with *accepted* messages to both KProposer and KLearners. A KLearner waits until it receives a quorum of such messages before it can deliver the operation to the application in user-space. The application in the user-space receives the operation from a KLearner reading from a character device, as depicted in Fig. 5.2. The application blocks until the operation is ready and the KLearner has written it to the character device. operations are delivered to the application in increasing instance order.

Both KProposer and KLearner take care of possible message losses. KProposer resends first- and second-phase messages for timed-out instances. A KLearner restarts an instance i when it receives a quorum of second-phase messages for an instance $j > i$, but not for i. Restarting an instance i means the KLearner tries to propose a *NULL* value for instance i in order to learn the previous decided value (if any) or skip the instance otherwise. Clients and application are not aware of

Kernel Paxos internals. Clients simply send requests in an Ethernet frame. From the application point of view, receiving an operation is as simple as reading from a file.

5.3 Implementation

Kernel Paxos was implemented in C and the source code is publicly available.[3] Kernel Paxos borrows Paxos algorithm logic implementation from LibPaxos,[4] an open-source user-space Multi-Paxos library written in C as well. Each Paxos role comprises five components responsible for specific tasks. Fig. 5.4 shows the relation among such components.

The *network* component handles the creation of Ethernet frames with application specific payload and type to be sent as well as the unpacking of received frames and invocation of registered handlers.

The *storage* component is a general-purpose in-memory key-value store, used mainly to keep the acceptor state and "live" proposer and acceptor instances.

The *kpaxos* component consists of two sub-components: (a) a network-independent set of objects for each Paxos role and functions to update the state from received messages returning the replies to be sent over the network; (b) the set of handlers for each Ethernet type, which are responsible for (de)serializing messages and invoking the corresponding function in the other sub-component.

To keep statistical information about Kernel Paxos performance, a *statistics* component is also notified by *kpaxos* handlers every time a new message arrives. When the kernel module is unloaded the collected statistical data is persisted to disk.

The last component, *chardev*, manages the creation, reading, writing and destruction of character devices and represents the interface between Kernel Paxos and the actual replicated application. It is only present in a KLearner, which reads from this component to deliver operations to the application in increasing instance order.

5.4 Performance

This section describes the main motivations that guided the experiments design (§5.4.1), details the environment in which experiments were con-

---

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

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

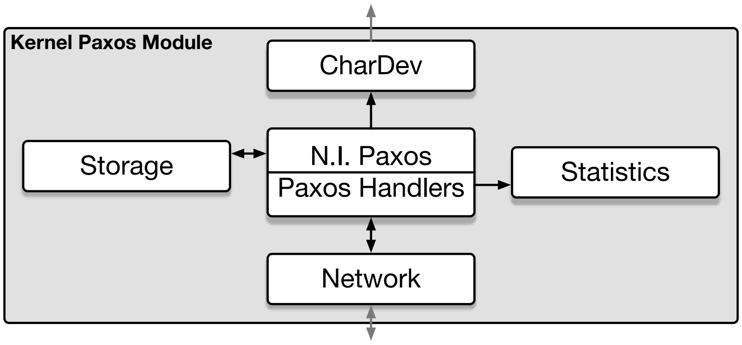


Figure 5.4. Main components of Kernel Paxos module.

ducted (§5.4.2), and then presents and discusses the results (§5.4.3 – §5.4.7). It concludes with a summary of the main findings (§5.4.8).

5.4.1 Evaluation rationale

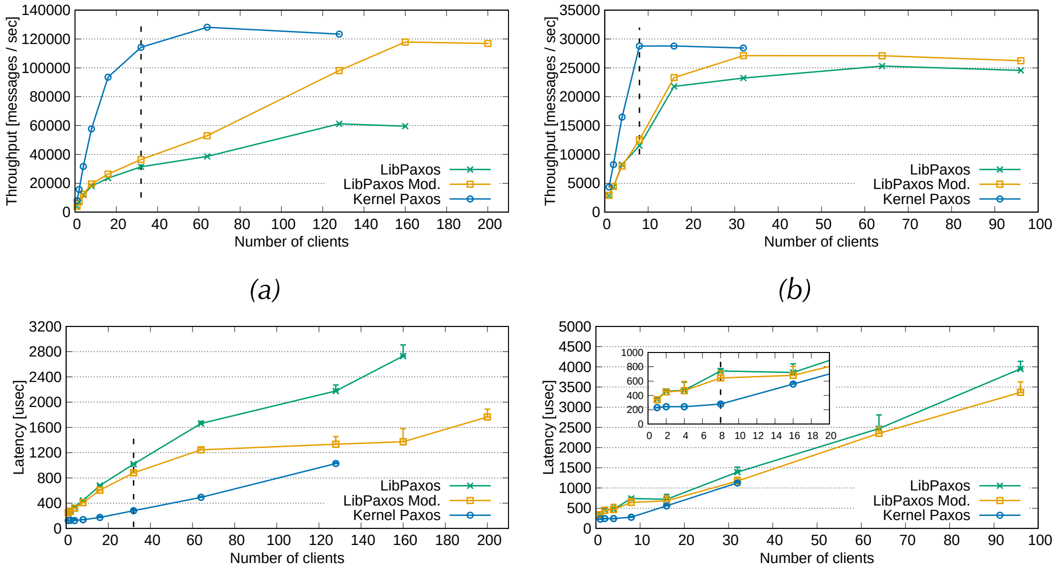
All experiments are conducted in a local-area network (LAN). The LAN provides a controlled environment, where experiments can run in isolation. In all experiments, there is a single proposer representing the leader and three acceptors, deployed in different machines. To measure the performance of the ordering protocol, each client is also a learner. Thus, the latency of a command represents the time between the command has been sent by the client and delivered to the learner in the same process or kernel module.

The experiments use a micro-benchmark with 64-byte and 1000-byte messages to evaluate particular scenarios in isolation. The benchmark considers executions with a single client to understand the performance of Kernel Paxos without queueing effects and executions with multiple clients to evaluate the protocol under stress. Executions with multiple clients are performed by controlling the number of outstanding messages of up to three learner processes (or kernel modules) in different machines.

5.4.2 Environment

The experiments compare Kernel Paxos to LibPaxos, a user-space library that provides the basic Paxos logic present in the “network-independent” part of KPaxos component (see Figure 5.4). Besides comparing with the original LibPaxos library, a modified version of LibPaxos that includes all the optimizations applied to Kernel Paxos and described in §5.1 has been implemented and assessed.

The LAN environment consists of a set of nodes, each one with an eight-core Intel Xeon L5420 processor working at 2.5GHz, 8GB of memory, SATA SSD disks, and 1Gbps ethernet card. Each node runs CentOS 7.1 64 bits. The round-trip time (RTT) between nodes in the cluster is around 0.1ms. Kernel Paxos has also been assessed on nodes with 10Gbps ethernet card, where the RTT is around 0.04ms. The experiments in the 1Gbps nodes compare Kernel Paxos to LibPaxos, while the experiments in the 10Gbps nodes assess the performance of Kernel Paxos in two different configurations.



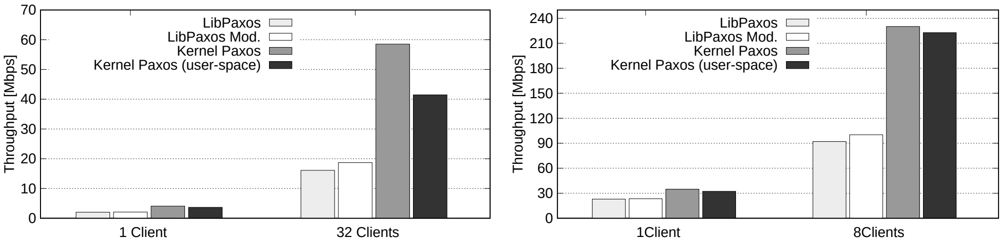
(a) (b)

(c) (d)

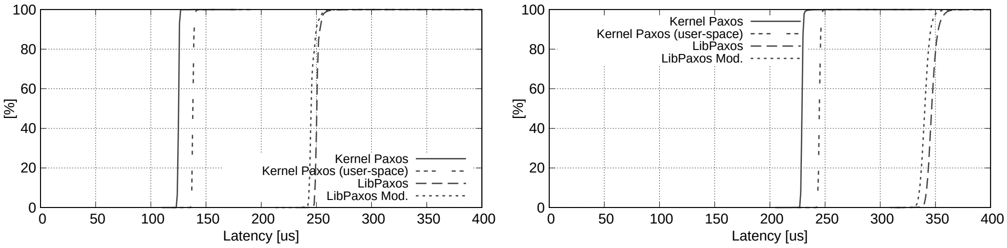
Figure 5.5. Throughput and median latency for increasing number of clients with 64-byte (left) and 1000-byte messages (right). Whiskers represent the 95-th percentile.

5.4.3 Throughput in a LAN

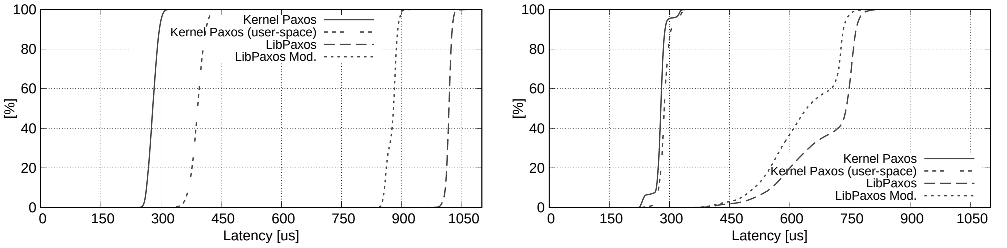
These experiments measure the throughput of standard LibPaxos, improved LibPaxos, and Kernel Paxos as the number of clients increase in scenarios with 64-byte and 1000-byte messages, as depicted in Figure 5.5(a) and Figure 5.5(b), respectively. In summary, Kernel Paxos outperforms both the original LibPaxos and the modified version for any number of clients and message size.



(a) *(b)*



(c) 1 client. *(d)* 1 client.



(e) 32 clients. *(f)* 8 clients.

Figure 5.6. Throughput and latency CDF with the selected number of clients for 64-byte (left) and 1000-byte messages (right).

Performance with small (64-byte) messages

With a single client, Kernel Paxos is twice as fast as LibPaxos with throughput around 8000 msgs/sec. The difference increases even more as the number of clients is augmented, with Kernel Paxos throughput close to 100000 msgs/sec with 16 clients, 4x better than both versions of LibPaxos. The modification to LibPaxos starts paying off with more clients. The performance of the modified version with 128 clients is close to 100000 msgs/sec while the original library already saturates the proposer and cannot go beyond 60000 msgs/sec. The maximum performance of Kernel Paxos, LibPaxos and LibPaxos Modified is 128000, 61000 and 117000 msgs/sec with 64, 128 and 160 clients, respectively. The maximum represents the point where the *proposer* is overloaded, i.e., its CPU reaches 100%.

Perfomance with large (1000-byte) messages

With a single client Kernel Paxos is still faster than LibPaxos with throughput around 4500 msgs/sec against less than 3000 msgs/sec for LibPaxos. In this setup, larger messages saturate the network before *proposers* are overloaded. For such reason, both versions of LibPaxos have similar performance and reach a maximum of 25000 and 27000 msgs/sec for the original and the modified version. Kernel Paxos can still push close to 29000 msgs/sec. For 1000-byte messages, this represents a throughput of around 235 Mbps in the clients, amounting to an aggregated throughput in the proposer of 940 Mbps: 235 Mbps from the clients and 3×235 Mbps from the acceptors. This aggregated throughput was confirmed with *iptraf* on the proposer machine.

5.4.4 Latency in a LAN

The overall better performance of Kernel Paxos in terms of throughput repeats for latency. Figure 5.5(c) and Figure 5.5(d) exhibit the results for small and large client messages.

Latency with small (64-byte) messages

With a single client, the latency of Kernel Paxos is very close to 1 round-trip time (RTT) and as low as 124 microseconds. LibPaxos, on the contrary, has a median latency of 248 and 252 microseconds for the modified and original versions, what represents more than 2 RTTs. The modifications in LibPaxos have direct positive impact on the latency. With 160 clients, the modified version has a median latency of around 1400 microseconds against almost twice the value for the original version with 2700 microseconds. In all implementations the measured latencies are quite stable with very low variance around the median values. With the maximum throughput, Kernel Paxos latency is below 500 microseconds with 64 clients, 3x to 5x less than the values for LibPaxos.

Latency with large (1000-byte) messages

With bigger messages, the latency for all protocols increase. With a single client, messages need close to 350 microseconds to be ordered with LibPaxos and 230 microseconds with Kernel Paxos. At maximum throughput, Kernel Paxos can keep latency as low as 279 microseconds while LibPaxos needs 8x and 9x more time to deliver a message for the modified and original libraries, respectively. Kernel Paxos has smaller latency, higher throughput and needs only 8 clients to saturate the data link, while LibPaxos needs at least 32 clients.

5.4.5 Performance with similar number of clients

As observed in Figure 5.5, depending on the message size clients can saturate either the proposer's CPU or the communication links. In such cases, the maximum throughput of Kernel Paxos is only 10% superior to the modified version of LibPaxos in the same setup.

The next results compare the various protocols in scenarios with the same number of clients, choosing two specific scenarios from the previous experiments and zoomed in on the behavior of the protocols. These scenarios represent (a) the setup with a single client and thus no contention or queuing effects, to isolate the benefits of eliminating context switches and TCP/IP stack; and (b) the performance under stress, specifically with 32 clients for 64-byte massages and 8 clients for 1000-byte messages, since none of the implementations are saturated at these points. The dashed lines in Figure 5.5 highlight the selected points.

Besides, these experiments also evaluate the behavior of Kernel Paxos with client/learner in user-space sending messages to the proposer using raw sockets and learning decided values from the character device created by the kernel-space learner.

Throughput and latency with small (64-byte) messages

Figure 5.6(a) exhibits the throughput advantage introduced by Kernel Paxos. With a single client, Kernel Paxos reduction in communication overhead and context switches is enough to double the throughput. With 32 clients, Kernel Paxos can reach 60 Mbps, almost 4x the performance of the original library and 3x the performance of the modified version.

Besides, the Cumulative Distribution Function (CDF) with 1 client shown in Figure 5.6(c) for Kernel Paxos is quite stable with the 99.9-th percentile equal to 128 microseconds while the median is 126 microseconds. These values represent at least half of the latencies for the user-space protocols.

Figure 5.6(e) exhibits a similar behavior with 32 clients, where the CDF is stable in Kernel Paxos going from 280 to 305 microseconds in the 50-th and 95-th percentile, respectively. In the user-space library, the latency is at least 3x greater.

Another interesting result regards the difference between Kernel Paxos entirely in the kernel and the deployment with client and application in user-space. The throughput and latency distribution show that, although there is a small overhead in the latter deployment, Kernel Paxos is still 2x to 3x times better than LibPaxos.

Throughput and latency with large (1000-byte) messages

Figure 5.6(b) shows the throughput for this scenario. With a single client, Kernel Paxos reaches a throughput 50% greater than LibPaxos, around 35 Mbps. With only 8 clients, Kernel Paxos almost saturated the proposer communication link: with 232 Mbps in the client, the proposer aggregated throughput is around 930 Mbps. LibPaxos, on the other hand, uses only 10% of the communication link capacity, since performance is capped by the processing power of the proposer.

Regarding latency, both LibPaxos and Kernel Paxos are very stable for a single client, with latencies close to 340 and 230 microseconds, respectively, as shown in Figure 5.6(d). With 8 clients, however, LibPaxos latency has a large variation, from 500 microseconds in the 5-th percentile to almost 800 microseconds in the 95-th percentile. Kernel Paxos remains quite stable with values between 220 and 295 microseconds for the same percentiles as exhibited in Figure 5.6(f).

For large messages, the difference of a deployment of Kernel Paxos with client and application in user-space is minimal for both throughput and latency.

5.4.6 Context-switch overhead

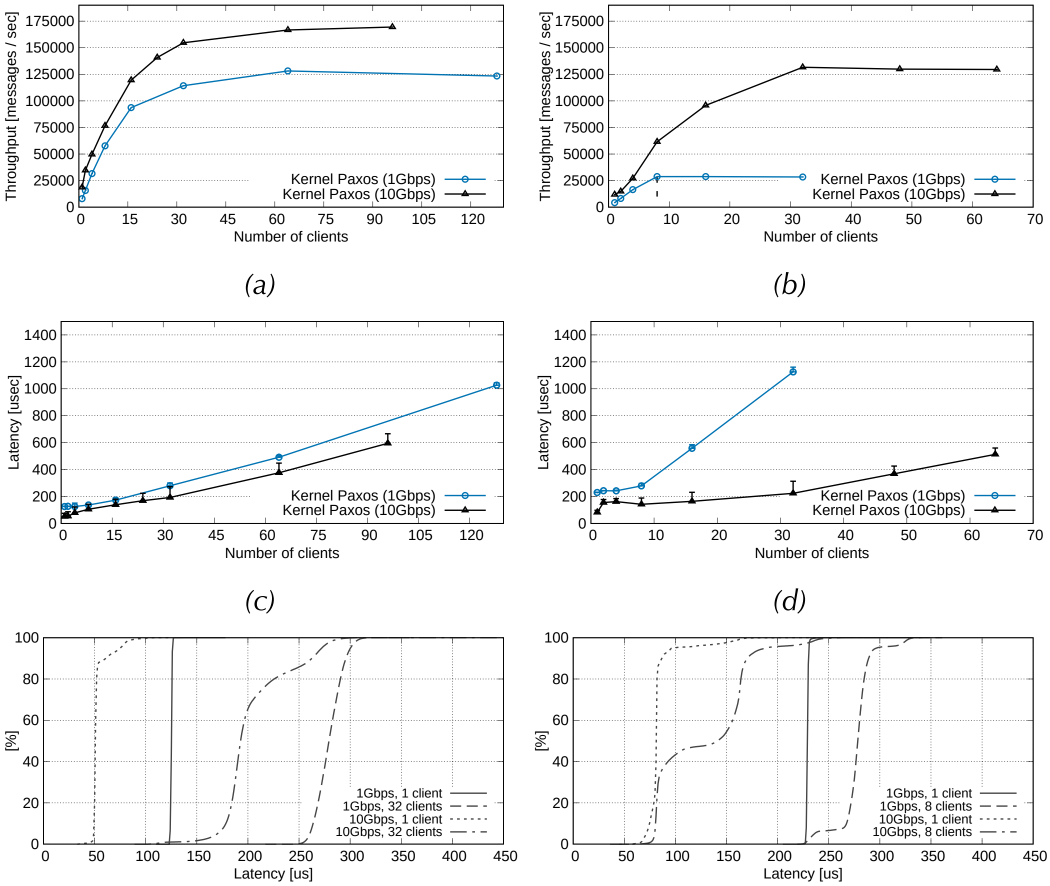
Since LibPaxos is a user-space library, sending and receiving data imply invoking a system call to context-switch to kernel-space and access the network card, followed by another context-switch back to user-space when the system call returns.

Kernel Paxos runs entirely in kernel-space and the context-switch overhead at the proposer and acceptors is completely eliminated, contributing to improve the overall performance.

To quantify the difference, *dstat* is used to monitor the number of context-switches for the experiments with 1000-byte messages. Table 5.2 exhibits the approximate numbers for both libraries as the number of clients increase.

Clients	*LibPaxos*	*Kernel Paxos*
1	7000	150
2	9000	150
4	12000	150
8	12000	150

Table 5.2. Approximate number of context-switches with increasing number of clients



(a) (b)

(c) (d)

(e) (f)

Figure 5.7. Kernel Paxos performance in 1Gbps and 10Gbps setups for increasing number of clients with 64-byte (left) and 1000-byte messages (right). Whiskers represent the 95-th percentile.

5.4.7 Kernel Paxos in a 10Gbps network

The last experiments evaluate Kernel Paxos in a 10Gbps network and compare the results with the 1Gbps network. The 10Gbps setup represents environments typically used by hardware-based consensus solutions, which rely on high-throughput network cards. The experiments assess Kernel Paxos with small and large messages while increasing number of clients.

Throughput and latency with small (64-byte) messages

Figure 5.7(a) and 5.7(c) show throughput and latency in both networks. The throughput for a single client in the faster network is more than 2x greater, close to 19000 msgs/sec or 10 Mbps. The median latencies are 52 and 126 microseconds for the 10Gbps and 1Gbps network, respectively, representing roughly the measure of one RTT.

The maximum throughput is 170000 msgs/sec (87 Mbps) for the 10Gbps network against 129000 msgs/sec (66 Mbps). In both cases the proposer is CPU bound and the difference in this case is mainly due to the better CPU in the nodes with 10Gbps NIC.

The CDF in Figure 5.7(e) confirms that although the values are smaller for the 10Gbps network, the slower network has a much more stable curve.

Throughput and latency with large (1000-byte) messages

With 1000-byte messages, the communication links are expected to saturate before the proposer is CPU bound. As demonstrated in Figure 5.5(b) and now in Figure 5.7(b), the 1Gbps link is saturated in the proposer when the client reaches around 29000 msgs/sec or 240 Mbps. The 10Gbps link, on the other hand, has enough bandwidth to go beyond 1Gbps in the client, and close to 4.5Gbps in the proposer, which differently from the slower network, is saturated at around 130000 msgs/sec.

The median latencies are 83 and 230 microseconds with a single client and 224 and 279 microseconds with 8 and 32 clients for the 10Gbps and 1Gbps networks, as seen in Figure 5.7(d).

The CDF shown in Figure 5.7(f) reinforces the overall stability in the 1Gbps network, most likely due to lower load in the proposer, since the 1Gbps link is not enough to overload it in this configuration.

5.4.8 Summary

This section draws some general conclusions from the experimental evaluation:

- Kernel Paxos outperforms LibPaxos, both original and improved versions, in each and every experiment, with small and large messages, and presents superior throughput and smaller latency for the same number of clients.
- In setups with large messages (1000 bytes), the evaluated libraries are able to saturate the proposer's data link in a 1Gbps network.
- In a 10Gbps network, Kernel Paxos can deliver commands at a rate superior to 1Gbps, representing 130000 1000-byte messages per second.

The last two observations suggest that batching could be quite effective in further boosting Kernel Paxos performance at the cost of small increase in latency. For example, if the KProposer would batch 10 small client requests (i.e., 100 bytes) in 1000-byte messages, Kernel Paxos could potentially reach one million ordered messages per second.

5.5 Related work

Improving the performance of Paxos has been a hot research topic. The following discussion considers related proposals from three perspectives: those that take advantage of special network topologies (i.e., overlays); those that resort to specialized hardware; and those that exploit the semantics of applications.

5.5.1 Protocols that exploit special topologies

Several approaches modify the Paxos [48] protocol to improve performance. In [38], the authors proved that ring topologies allow systems to achieve optimal throughput. Some protocols that benefit from such topologies are LCR [38], Spread [6] and Ring Paxos [57]. LCR arranges processes in a ring and uses vector clocks to ensure total order. Spread, which is based on Totem [7], relies on daemons interconnected as a ring to order messages, while message payloads are disseminated using IP-multicast. Finally, Ring Paxos deploys Paxos processes in a ring to maximize throughput. A problem of all such ring-based protocols is that their latency is proportional to the size of the system times the network point-to-point latency.

5.5.2 Protocols that exploit special hardware

Some solutions provide ordering as a network service. Speculative Paxos [68] and NOPaxos [52] use a combination of techniques to eliminate packet reordering in a data center, including IP multicast, fixed-length network topologies, and a single top-of-rack switch acting as a serializer (although in case of NOPaxos a Linux server can act as a sequencer at the cost of higher latency). NetPaxos [29] deploys acceptors and coordinator in switches. Although this reduces the latency, it makes the recovery more complex: switches have limited memory, making it hard to keep the state of acceptors. Besides, replacing a failed coordinator is not trivial in such configuration. In [43], the authors bring the consensus logic to the hardware, implementing Zookeeper [42] in FPGA. Even though the throughput is quite high, such programmable hardwares are not suitable to store large amount of replicated state [52]. While Kernel Paxos cannot achieve the performance of hardware implementations for state machine replication protocols, it can still provide a considerably high throughput without depending on any network ordering guarantees or specialized hardware. Kernel Paxos can virtually run on any machine with commodity hardware and a Linux operating system.

5.5.3 Protocols that exploit message semantics

Several Paxos variants yield better performance with a multi-leader or generalized approach. EPaxos [59] improves on traditional Paxos [48] and reduces the leader overload by allowing any replica to order commands. As long as replicas observe common dependencies set, commands can be ordered in one round-trip. M2Paxos [67] is a implementation of Generalized Consensus [49] that allows different ordering of non-conflicting commands. It detects conflicts by looking into the objects the commands access and making sure a replica has exclusive access to such objects. M2Paxos guarantees that commands that access the same objects are ordered by the same replica. Any of such solutions could benefit from Kernel Paxos strategies to reduce latency and increase performance, i.e., the protocols could be moved to the Linux kernel as module and use raw Ethernet frames to exchange messages.

5.6 Conclusion

At the core of of most strong consistent replicated systems like state machine replication and atomic multicast, there is the consensus problem. While many protocols have been proposed to implement consensus, Paxos stands out. Therefore, much effort has been placed on implementing Paxos efficiently. This thesis proposes Kernel Paxos, a different approach to improving the performance of Paxos. Differently from previous proposals, Kernel Paxos does not rely on special network topologies, specialized hardware or application semantics. Kernel Paxos addresses the performance challenge by reducing the communication overhead in two directions: (a) eliminating context switches by placing the protocol implementation in the Linux kernel, and (b) bypassing the TCP/IP stack and exchanging protocol messages inside raw Ethernet frames. The performance of Kernel Paxos has been assessed and compared to user-space implementations. The results show that in most configurations of interest Kernel Paxos largely outperforms user-space protocols.

Chapter 6

Conclusions

Many modern online applications require scalable performance and high availability. Designing systems that combine scalability and fault tolerance, however, is challenging. This thesis has provided two broad directions applications can follow to fulfill such requirements. The first approach yields sharded systems a group communication abstraction with reliability and ordering guarantees, presenting efficient atomic multicast algorithms for both crash- and byzantine-failure models. The second approach explores partial order and Paxos different roles to improve SMR scalability. It also improves performance for the core piece of most replicated systems, the consensus service. The thesis provides a Paxos implementation that eliminates common sources of overhead.

6.1 Research assessment

This thesis presents four main contributions: (i) FastCast, the first crash-fault tolerant genuine atomic multicast algorithm to deliver multi-group messages in four communication delays; (ii) ByzCast, the first atomic multicast algorithm that tolerates Byzantine failures; (iii) GeoPaxos, a SMR protocol that provides availability and can scale with the number of replicas in a geographically distributed deployment; and (iv) Kernel Paxos, a high-performance Paxos service infra-structure provided as a Linux loadable kernel module.

FastCast

The algorithm extends Skeen's original algorithm [15] to support server crashes and provides an additional optimistic path that merges with original flow to deliver multi-group messages fast. The assumption for such fast delivery is the existence of a stable leader within each group. If the assumption holds, each leader tries to optimistically guess a message final timestamp. If the guess matches the actual timestamp after consensus, then a message can be delivered fast, in four communication delays. FastCast algorithm has been proved correct, implemented, and largely assessed in both LAN and WAN environments.

ByzCast

ByzCast builds up on existing BFT abstractions to provide the first atomic multicast protocol that can tolerate Byzantine failures. It organizes groups hierarchically in a overlay tree. Multi-group messages are ordered by the common ancestor that includes all the message destinations. Single-group messages go straight to target groups. ByzCast is therefore classified as *partially genuine*, since it is genuine for single-group messages, but potentially non-genuine for multi-group ones. The algorithm has been implemented and assessed in both local- and wide-area networks.

GeoPaxos

GeoPaxos extends the concept of partial order from atomic multicast algorithms to provide a fully replicated system with partial order, allowing for concurrent ordering and execution of operations that access unrelated objects. It does so by assigning objects to groups and turning replicas into multi-learner processes. It also provides mechanisms to manage operations that involve multiple groups. GeoPaxos explores locality and can scale linearly with respect to single-group operations. The protocol has been implemented and evaluated in LAN and WAN deployments.

Kernel Paxos

Consensus stands at the core of all the previous contributions of this thesis. For this reason, a fast and reliable consensus implementation is fundamental for the performance of distributed systems. This thesis has presented Kernel Paxos, a Paxos implementation with high throughput and low latency provided as a loadable Linux kernel module. Such an approach speeds up decisions for two main points: (i) it bypasses the TCP/IP stack and provides messages on top of Ethernet frames, and (ii) it places all the communication and algorithm logic in the kernel to avoid context switches. The implementation has been extensively evaluated and can potentially batch over one million small messages per second.

6.2 Future directions

The contributions of this thesis have raised several questions that are worth further investigation. The following discusses the main open questions.

Partially genuine atomic multicast

ByzCast defines a new class of atomic multicast algorithms called partially genuine. It has been shown in [75] that non-genuine atomic multicast algorithms can deliver commands with lower latency since they typically require fewer communication steps. However, they do not behave as well as genuine atomic multicast algorithms when the number of groups and the load increase as they involve processes other than those in the message's destination. A partially genuine algorithm is genuine with respect to single-group messages, but can potentially involve groups other than the destinations for multi-group messages.

Intuitively, the behavior of multi-group messages for different workloads could lead to situations where a partially genuine algorithm performs better. The implementation of a ByzCast equivalent algorithm in the crash-failure model is quite straightforward and would allow comparison between genuine and partially genuine algorithms. A detailed analysis of such scenarios would bring a valuable contribution and could result in a solution that puts together the main advantages of genuine and non-genuine algorithms.

ByzCast also defines the challenge of choosing the best overlay tree as an optimization problem, showing the strong relation between workload and tree topology for performance. The definition of the best topology based on the on-the-fly observed workload and the adjustment of the tree topology dynamically may also be an interesting ByzCast spin-off.

Dynamic ownership changes

In GeoPaxos, the larger the proportion of single-group operations in relation to multi-group ones, the more the system can scale. Keeping this proportion as large as possible is not a trivial task. Firstly, the workload of the system must have a pattern that allows assigning objects to groups, preferably with some degree of locality. Furthermore, even supposing we have such a workload, the dynamics of the system may change along time, leading to temporary locality characteristics that non long reflect the initial optimal ownership partitioning scheme and thus hurt performance.

To address this problem, object preferred sites must be changed among groups accordingly. GeoPaxos already allows static object ownership transfers among groups using the *move(object_id_list, source, destination)* operation, addressed to the source and the destination groups, where *object_id_list* contains a list of objects whose ownership should be re-assigned.

Even with the advantage that changing objects preferred site does not imply in any transfer of actual objects (since every replica contains a full copy of the application state), GeoPaxos does not provide a smart way of analyzing workload changes and triggering a redistribution of objects among groups.

Understanding and adapting to the dynamic behavior of the system is a “must-have” to any infra-structure that intends to be used in real-world deployments. In order to enhance GeoPaxos with mechanisms to address these issues, a natural follow-up for GeoPaxos is the proposal and implementation of a set of heuristics that would allow to automatically adjust to the workload changes. Moreover, a differentiation between read and write operations would allow the adoption of a ROWA model bringing a sensitive benefit for read intensive workloads [11, 27, 86].

Paxos as a service

Kernel Paxos minimizes the operating system overhead and delivers a low latency Paxos implementation. The current system can, however, be further enhanced. First of all, the acceptors state should be persisted to disk to allow recovery from crashes and increased reliability. Additionally, a useful extension would be the provision of Paxos as an operating system service. This means the current kernel modules should be able to attend multiple applications, presenting itself as an isolated service to each one.

Bibliography

[1] Abd-El-Malek, M., Ganger, G., Goodson, G., Reiter, M. and Wylie, J. [2005]. Fault-scalable Byzantine fault-tolerant services, *SOSP*.

[2] Agarwal, D. A., Moser, L. E., Melliar-Smith, P. M. and Budhia, R. K. [1998]. The totem multiple-ring ordering and topology maintenance protocol, *ACM Trans. Comput. Syst.* **16**(2): 93–132.

[3] Allamanis, M., Scellato, S. and Mascolo, C. [2012]. Evolution of a location-based online social network: Analysis and models, *IMC*.

[4] Amazon [2018]. Elastic Compute Cloud (EC2) – Cloud Server & Hosting – AWS, `https://aws.amazon.com/ec2/`. Online; accessed 2018-12-09.

[5] Amir, Y., Danilov, C., Dolev, D., Kirsch, J., Lane, J., Nita-Rotaru, C. and Josh Olsen, D. Z. [2010]. STEWARD: Scaling Byzantine fault-tolerant replication to wide area networks, *IEEE Trans. on Dependable and Secure Computing* **7**(1).

[6] Amir, Y., Danilov, C., Miskin-Amir, M., Schultz, J. and Stanton, J. [2004]. The spread toolkit: Architecture and performance, *Johns Hopkins University, Tech. Rep. CNDS-2004-1* .

[7] Amir, Y., Moser, L. E., Melliar-Smith, P. M., Agarwal, D. A. and Ciarfella, P. [1995]. The totem single-ring ordering and membership protocol, *ACM Transactions on Computer Systems (TOCS)* **13**(4): 311–342.

[8] Arun, B., Peluso, S., Palmieri, R., Losa, G. and Ravindran, B. [2017]. Speeding up consensus by chasing fast decisions, *DSN*.

[9] Babay, A. and Amir, Y. [2016]. Fast total ordering for modern data centers, *ICDCS*.

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

[11] Bernstein, P. A., Hadzilacos, V. and Goodman, N. [1987]. Concurrency control and recovery in database systems.

[12] Bessani, A., Sousa, J. and Alchieri, E. [2014]. State machine replication for the masses with BFT-SMaRt, *DSN*.

[13] Bezerra, C. E., Pedone, F. and van Renesse, R. [2014]. Scalable state-machine replication, *DSN*.

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

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

[16] Brewer, E. [2017]. Spanner, truetime and the cap theorem, *Technical report*, Google.
URL: *https://research.google.com/pubs/pub45855.html*

[17] Brodersen, A., Scellato, S. and Wattenhofer, M. [2012]. Youtube around the world: Geographic popularity of videos, *WWW*.

[18] Cachin, C. [2009]. Yet another visit to Paxos, *Technical Report RZ3754*, IBM Research, Zurich, Switzerland.

[19] Cachin, C. and Vukolic, M. [2017]. Blockchain consensus protocols in the wild, *Technical report*, Cornell University.

[20] Calder, B., Wang, J., Ogus, A., Nilakantan, N., Skjolsvold, A., McKelvie, S., Xu, Y., Srivastav, S., Wu, J., Simitci, H., Haridas, J., Uddaraju, C., Khatri, H., Edwards, A., Bedekar, V., Mainali, S., Abbasi, R., Agarwal, A., Haq, M. F. u., Haq, M. I. u., Bhardwaj, D., Dayanand, S., Adusumilli, A., McNett, M., Sankaran, S., Manivannan, K. and Rigas, L. [2011]. Windows azure storage: A highly available cloud storage service with strong consistency, *SOSP*.

[21] Castro, M. and Liskov, B. [1999]. Practical Byzantine Fault Tolerance, *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI'99)*, USENIX, New Orleans, LA.

[22] Castro, M. and Liskov, B. [2002]. Practical byzantine fault tolerance and proactive recovery, *ACM Trans. on Computer Systems (TOCS)* **20**(4): 398–461.

[23] Chandra, T. D., Griesemer, R. and Redstone, J. [2007]. Paxos made live: an engineering perspective, *PODC'07*, ACM, Portland, OR, pp. 398–407.

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

[25] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A. and Gruber, R. E. [2008]. Bigtable: A distributed storage system for structured data, *ACM Trans. Comput. Syst.* **26**(2).

[26] Coelho, P., Schiper, N. and Pedone, F. [2017]. Fast atomic multicast, *Proceedings of the 47th Dependable Systems and Networks (DSN)*.

[27] Cook, S. A., Pachl, J. and Pressman, I. S. [2002]. The optimal location of replicas in a network using a read-one-write-all policy, *Distributed Computing* **15**(1): 57–66.

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

[29] Dang, H. T., Sciascia, D., Canini, M., Pedone, F. and Soulé, R. [2015]. Netpaxos: Consensus at network speed, *SOSR*, ACM.

[30] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P. and Vogels, W. [2007]. Dynamo: Amazon's highly available key-value store, *SOSP*.

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

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

[33] Fischer, M. J. [1983]. The consensus problem in unreliable distributed systems (a brief survey), *Technical Report DCS/RR-273*, Department of Computer Science, Yale University.

[34] Fischer, M. J., Lynch, N. A. and Patterson, M. S. [1985]. Impossibility of distributed consensus with one faulty process, *J. ACM* **32**(2): 374–382.

[35] Fritzke, U., Ingels, P., Mostéfaoui, A. and Raynal, M. [1998]. Fault-tolerant total order multicast to asynchronous groups, *SRDS*.

[36] Guerraoui, R., Knezevic, N., Quema, V. and Vukolic, M. [2010]. The next 700 BFT protocols, *Proc. of the 5th ACM European conf. on Computer systems (EUROSYS'10)*, Paris, France.

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

[38] Guerraoui, R. and Vukolić, M. [2010]. Refined quorum systems, *Distributed Computing* **23**(1): 1–42.

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

[40] Herlihy, M. P. and Wing, J. M. [1990]. Linearizability: A correctness condition for concurrent objects, *Trans. on Programming Languages and Systems* **12**(3): 463–492.

[41] Hunt, P., Konar, M., Junqueira, F. P. and Reed, B. [2010a]. Zookeeper: Wait-free coordination for internet-scale systems., *USENIX ATC*.

[42] Hunt, P., Konar, M., Junqueira, F. and Reed, B. [2010b]. ZooKeeper: Wait-free coordination for Internet-scale systems, *USENIX Annual Technology Conference*.

[43] István, Z., Sidler, D., Alonso, G. and Vukolic, M. [2016]. Consensus in a box: Inexpensive coordination in hardware., *NSDI*.

[44] J. C. Corbett, J. D. and et al, M. E. [2012]. Spanner: Google's globally distributed database, *OSDI*.

[45] Kotla, R., Alvisi, L., Dahlin, M., Clement, A. and Wong, E. [2009]. Zyzzyva: Speculative Byzantine fault tolerance, *ACM Trans. on Computer Systems* **27**(4).

[46] Kraska, T., Pang, G., Franklin, M. J. and Madden, S. [2012]. MDCC: Multi-Data Center Consistency, *CoRR* .

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

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

[49] Lamport, L. [2005]. Generalized consensus and paxos, *Technical Report MSR-TR-2005-33*, Microsoft Research (MSR).

[50] Lamport, L. [2006]. Lower bounds for asynchronous consensus, *Distributed Computing* **19**(2): 104–125.

[51] Lamport, L. and Fischer, M. [1982]. Byzantine Generals and transaction commit protocols, *Technical Report 62*, SRI Int.

[52] Li, J., Michael, E., Sharma, N. K., Szekeres, A. and Ports, D. R. [2016]. Just say no to paxos overhead: Replacing consensus with network ordering., *OSDI*.

[53] Liu, S., Viotti, P., Cachin, C., Quéma, V. and Vukolic, M. [2016]. XFT: practical fault tolerance beyond crashes, *OSDI*.

[54] Lloyd, W., Freedman, M. J., Kaminsky, M. and Andersen, D. G. [2011]. Don't settle for eventual: scalable causal consistency for wide-area storage with COPS, *SOSP*.

[55] Mao, Y., Junqueira, F. and Marzullo, K. [2008]. Mencius: Building efficient replicated state machine for WANs, *8th Usenix Symposium on Operating Systems Design and Implementation (OSDI08)*, pp. 369–384.

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

[57] Marandi, P., Primi, M., Schiper, N. and Pedone, F. [2010]. Ring Paxos: A high-throughput atomic broadcast protocol, *Proc. of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'10)*, pp. 527–536.

[58] Miller, A., Xia, Y., Croman, K., Shi, E. and Song, D. [2016]. The honey badger of BFT protocols, *CCS*.

[59] Moraru, I., Andersen, D. G. and Kaminsky, M. [2013]. There is more consensus in egalitarian parliaments, *SOSP*.

[60] Nogueira, A., Casimiro, A. and Bessani, A. [2017]. Elastic state machine replication, *IEEE Trans. on Parallel and Distributed Systems* **28**(9).

[61] Ongaro, D. and Ousterhout, J. K. [2014]. In search of an understandable consensus algorithm., *USENIX ATC*.

[62] Padilha, R., Fynn, E., Soulé, R. and Pedone, F. [2016]. Callinicos: Robust transactional storage for distributed data structures, *USENIX ATC*.

[63] Padilha, R. and Pedone, F. [2013]. Augustus: Scalable and robust storage for cloud applications, *EuroSys*.

[64] Pedone, F. [2001]. Boosting system performance with optimistic distributed protocols, *Computer* **34**(12): 80–86.

[65] Pedone, F. and Schiper, A. [1998]. Optimistic atomic broadcast, *DISC*.

[66] Pedone, F. and Schiper, A. [1999]. Generic broadcast, *DISC*.

[67] Peluso, S., Turcu, A., Palmieri, R., Losa, G. and Ravindran, B. [2016]. Making fast consensus generally faster, *DSN*.

[68] Ports, D. R., Li, J., Liu, V., Sharma, N. K. and Krishnamurthy, A. [2015]. Designing distributed systems using approximate synchrony in data center networks., *NSDI*.

[69] Rao, J., Shekita, E. and Tata, S. [2011]. Using Paxos to build a scalable, consistent, and highly available datastore, *Proceedings of the VLDB Endowment* **4**(4): 243–254.

[70] Rodrigues, L., Guerraoui, R. and Schiper, A. [1998]. Scalable atomic multicast, *IC3N*.

[71] Scellato, S. [2012]. *Spatial properties of online social services: measurement, analysis and applications*, PhD thesis, University of Cambridge, UK.

[72] Schiper, N. [2009]. *On multicast primitives in large networks and partial replication protocols*, PhD thesis, Università della Svizzera italiana (USI).

[73] Schiper, N. and Pedone, F. [2007]. Optimal atomic broadcast and multicast algorithms for wide area networks, *PODC*.

[74] Schiper, N. and Pedone, F. [2008]. On the inherent cost of atomic broadcast and multicast in wide area networks, *ICDCN*.

[75] Schiper, N., Sutra, P. and Pedone, F. [2009]. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study, *SRDS*.

[76] Schiper, N., Sutra, P. and Pedone, F. [2010]. P-Store: Genuine partial replication in wide area networks, *SRDS*.

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

[78] Sciascia, D. and Pedone, F. [2013]. Geo-replicated storage with scalable deferred update replication, *DSN*.

[79] Sciascia, D., Pedone, F. and Junqueira, F. [2012]. Scalable deferred update replication, *DSN*.

[80] Sousa, J. and Bessani, A. [2012]. From Byzantine consensus to BFT state machine replication: A latency-optimal transformation, *EDCC*.

[81] Sousa, J. and Bessani, A. [2015]. Separating the WHEAT from the chaff: An empirical design for geo-replicated state machines, *SRDS*.

[82] Sousa, J., Bessani, A. and Vukolic, M. [2018]. A Byzantine fault-tolerant ordering service for the hyperledger fabric blockchain platform, *DSN*.

[83] Sovran, Y., Power, R., Aguilera, M. K. and Li, J. [2011]. Transactional storage for geo-replicated systems, *SOSP*.

[84] Turcu, A., Peluso, S., Palmieri, R. and Ravindran, B. [2014]. Be general and don't give up consistency in geo-replicated transactional systems, *OPODIS*.

[85] Veronese, G. S., Correia, M., Bessani, A., Lung, L. C. and Verissimo, P. [2013]. Efficient Byzantine fault-tolerance, *IEEE Trans. on Computers* **62**(1).

[86] Wolfson, O., Jajodia, S. and Huang, Y. [1997]. An adaptive data replication algorithm, *ACM Transactions on Database Systems (TODS)* **22**(2): 255–314.

[87] Yin, J., Martin, J.-P., Venkataramani, A., Alvisi, L. and Dahlin, M. [2003]. Separating agreement from execution for byzantine fault tolerant services, *SOSP*.

[88] Zieliński, P. [2005]. Optimistic generic broadcast, *DISC*.