Egalitarian Distributed Consensus

Iulian Moraru

CMU-CS-14-133

August 2014

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David Andersen, Chair
Miguel Castro, Microsoft Research
Greg Ganger
Garth Gibson
Michael Kaminsky

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was funded in part by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by the National Science Foundation under award CCF-0964474, and by a gift from Google. We thank Amazon for donating part of the EC2 time used for the experiments reported in this thesis.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: State machine replication, Paxos, fault tolerance, geo-distributed replication, leases

To my family.

Abstract

This thesis describes the design and implementation of state machine replication (SMR) that achieves near-perfect load balancing and availability, near-optimal request processing latency (especially in the wide area), and performance robustness when confronted with failures and slow replicas.

Traditionally, practical replicated state machines have used leader-based implementations of consensus algorithms, because it has been believed that they provide the best performance—highest throughput and lowest latency. At the same time, however, a leader-based approach has many drawbacks: the failure of the leader halts the entire replicated state machine temporarily, the speed of the entire set is determined by the speed of the leader, and, in geo-replicated scenarios, the distance to the leader causes remote clients to experience high latency.

This work shows that leaderless approaches can not only solve these problems and provide the flexibility of a completely decentralized system, but they can also achieve substantially higher performance than leader-based protocols. We introduce a new variant of the Paxos protocol that we call *Egalitarian Paxos.* In Egalitarian Paxos all replicas perform the same functions simultaneously to ensure better load balancing and availability, lower commit latency and higher performance robustness when compared to previous Paxos variants. The benefits of Egalitarian Paxos are most apparent in the wide area, where its latency is optimal in many practical scenarios. We show—both theoretically and empirically—that Egalitarian Paxos has the aforementioned benefits when updating the state of a replicated state machine. We then apply the same leaderless design principle to improve the SMR read performance: *quorum read leases* generalize previously proposed time lease-based approaches to allow arbitrary sets of replicas to perform strongly consistent local reads for parts of the replicated state.

Acknowledgments

I am very grateful to Dave, my advisor, and to Michael, essentially my co-advisor, for making my time as a PhD student a thoroughly exciting and enjoyable experience. Every time I will look back, the first thing I will remember will be our discussions and exchanges of ideas—challenging and great fun.

I am also grateful to the other members of my thesis committee. Garth and Greg's technical nous, drive and humor have been inspiring. Miguel's expertise was invaluable in refining the ideas of this thesis.

I was lucky to have been helped and kept honest by amazingly smart colleagues: the FAWN group—Amar Phanishayee, Bin Fan, Vijay Vasudevan, Hyeontaek Lim, Dong Zhou, Anuj Kalia, Huanchen Zhang, Charlie Garrod; and the PDL group—Lin Xiao, Jiří Šimša, Swapnil Patil, Raja Sambasivan, Alexey Tumanov, Elie Krevat, Michelle Mazurek, Kai Ren, Ilari Shafer, Jim Cipar.

The administrative aspects of the program have been made infinitely easier by Deb Cavlovich. Many thanks to Kathy McNiff and Angela Miller for all their help throughout my time at CMU. Karen Lindenfelser has been instrumental in the organization of wonderful PDL events. Without Joan Digney's help I would have spent many hours trying and failing to design posters.

This research was funded in part by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC), by the National Science Foundation under award CCF-0964474, and by a gift from Google. We thank Amazon for donating part of the EC2 time used for the experiments reported in this thesis.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet services today rely on massive and continuously expanding infrastructure: networks of data centers that span across continents and comprise hundreds of thousands and even millions of computers [56]. In this context, where machine failures are increasingly common and entire data centers can be made unavailable by human error [55] or extreme weather events [57], fault tolerance through state replication is critical for the availability and integrity of these applications.

The goal of this thesis is to advance the state of the art in strongly-consistent replicated computer systems design. It does so by improvements to the Paxos [35] protocol for state machine replication. The primary benefits of these improvements are that systems using it can retain strongly-consistent behavior with substantially lower latency for clients.

1.1 Problem Statement and Scope

Redundancy through state replication is the primary mechanism for fault tolerance in distributed systems. *State machine replication* is used extensively both within data centers—where machine failures are common and must be tolerated—and in the wide-area, to guard against data loss and service unavailability caused by data center outages [55, 57], and to ensure that the data is close to all of the geographically-distributed clients that access it [7, 16].

The design space for replication protocols is vast, reflecting the complex (and fundamental [19]) trade-offs between the desirable traits of a replicated system: availability, consistency and high performance—especially low latency. Availability is important, particularly for user-facing applications, because the delays caused by failed nodes or unreachable data centers can produce costly service disruptions. Strong consistency gives programmers an intuitive model for the effects of concurrent updates (i.e., akin to concurrent updates to a single copy of the system state), making it less likely that complex system behavior will result in bugs. Finally, low latency is increasingly important in the context of geo-replicated databases, where small inefficiencies add up to significant increases in latency—tens or hundreds of milliseconds. On top of these considerations, developers have naturally preferred simpler protocols that are easier to understand and debug.

On one side of the design spectrum, the systems that guarantee strong consistency rely on a variety of protocols ranging from simple primary-backup protocols, to more complex protocols such as Paxos and Byzantine fault tolerance. Over the past decade, the protocol of choice in systems where availability is critical and failures are benign has been Paxos [34, 35]. Unlike primary-backup protocols, Paxos does not depend on external failure detectors or reconfiguration services to recover from failures, and therefore, systems using Paxos have high availability. Examples of such systems include reliable lock services such as Chubby [10] and Boxwood [44], or geo-replicated table stores such as Spanner [16] and Megastore [7].

On the other hand, many recent proposals have focused on obtaining low latency at the expense of strong consistency [40, 42, 43, 52]. In practical terms, weaker consistency translates into increased complexity and implementation effort in the upper layers of the applications that depend on these protocols.

Deciding which of these two main approaches to use for a particular application hinges on the answer to the following dilemma: how much performance does our application need, and how much complexity are we willing to tolerate to get it? While the answer is application-specific, there is a related, more general question that makes this dilemma simpler: *how much can we decrease the cost of strong consistency* in the first place? This was the question that motivated this work.

As such, this thesis focuses exclusively on strongly-consistent systems: it improves the performance of strongly-consistent state machine replication on multiple dimensions, achieving optimality for many practical scenarios. When we talk about optimality in this thesis, we do so strictly in the confines of strongly-consistent systems (i.e., systems that provide linearizability or strict serializability [23]) that can continue to process commands[1] despite a specified number of concurrent crash failures or partitioned nodes.

Although we also consider various design decisions introduced by Byzantine fault-tolerant protocols, the improvements we describe target only non-Byzantine failures—this is by far the most common failure assumption used in practice today.

1.2 Thesis Hypothesis

There exists a generalization of the Paxos protocol that can simultaneously achieve (i) wide-area latency that is optimal for a strongly-consistent system, (ii) perfect load balancing across replicas, and (iii) constant availability. Furthermore, in many realistic scenarios, reading the replicated state can be performed with minimal latency—i.e., as low or lower than in weakly-consistent systems.

1.3 Results Overview

There are essentially two types of operations that replicated state machines must support: updating the state, and reading the state. Because reads are fundamentally cheaper to perform, it is beneficial to implement different mechanisms for updates and reads. The techniques that we propose in this thesis for improving each of these two aspects are largely orthogonal.

1.3.1 Efficiently Updating the State

We introduce a new state machine replication protocol called *Egalitarian Paxos*. Egalitarian Paxos (abridged EPaxos) retains the strong consistency of Paxos while improving performance in multiple respects. These improvements increase the complexity of the protocol when compared to Paxos, but unlike systems with weaker forms of consistency, this burden is almost entirely the responsibility of the protocol developer, not the clients of the system or the upper application layers.

[1] This refers to general commands that can update and read state. As we explain later, strongly-consistent read-only commands can sometimes be performed locally, thus achieving the minimum latency possible in any system (even weakly-consistent ones).

An important limitation in practical Paxos-based systems is that during efficient, failure-free operation, all clients communicate with a single master (or leader) server at all times. This optimization, sometimes termed “Multi-Paxos”, is important to achieving high throughput in practical systems [13]. Changing the leader requires invoking additional consensus mechanisms that substantially reduce throughput.

This algorithmic limitation has several important consequences. First, it can impair scalability by placing a disproportionately high load on the master, which must process more messages than the other replicas [45]. Second, when performing geo-replication, clients will incur additional latency for communicating with a remote master. Third, as we show in this thesis, traditional Paxos variants are sensitive to both long-term and transient load spikes and network delays that increase latency at the master. Finally, this single-master optimization can harm availability: if the master fails, the system cannot service requests until a new master is elected. Previously proposed solutions such as partitioning or using proxy servers are undesirable because they restrict the type of operations the cluster can perform. For example, a partitioned cluster cannot perform atomic operations across partitions without using additional techniques.

Egalitarian Paxos has no designated leader process. Instead, clients can choose, at every step, which replica to submit a command to, and in most cases the command will be committed without interfering with other concurrent commands. This allows the system to distribute the load evenly to all replicas, eliminating the first bottleneck identified above (having one server that must be on the critical path for all communication). The system can provide higher availability because there is no transient interruption because of leader election: there is no leader, and hence, no need for leader election, as long as more than half of the replicas are available. Finally, EPaxos’s flexible load distribution is better able to handle permanently or transiently slow nodes, as well as the latency heterogeneity caused by geographical distribution of replicas, substantially reducing both the median and tail commit latency—e.g., unlike any previous state machine replication protocol, EPaxos has optimal median commit latency in the wide-area for three and five replica setups (i.e., it attains the lowest update latency possible in a strongly-consistent system that can tolerate up to two concurrent failures). In our experiments, EPaxos achieved an up to 3× improvement in throughput and an up to 65% decrease in wide-area commit latency when compared to Multi-Paxos.

1.3.2 Reading the State with Very Low Latency

We introduce the concept of *quorum read leases* which allow Paxos replicas to perform local reads with minimal latency, while at the same time preserving high write performance.

Many Paxos-based systems improve read performance by using some form of time leases, in which one or several replicas can satisfy read requests locally without having to commit an operation using the relatively more expensive Paxos protocol. This optimization obviously improves read throughput and latency, but also improves write performance by reducing the total number of operations that must be handled by the Paxos replicas.

A variety of approaches to read leases have been used in practical systems. Most Paxos-based systems use the Multi-Paxos optimization, in which one node is temporarily selected as the "stable leader" and is responsible for orchestrating all operations. Absent failures, Multi-Paxos enables operations to commit in a single round trip from the leader to a majority of replicas. In a Multi-Paxos environment, the most common read lease optimization is to grant the stable leader a read lease on all objects comprising the state—files, key-value pairs or relational database records, depending on the application. As a result, reads can be handled with a single round-trip to the leader, and writes incur no additional slowdown, compared to a Paxos system that does not use leases.

Other systems, such as Google's Megastore [7], instead grant a read lease to *all* nodes. This decision aggressively optimizes for reads at the expense of write latency: all reads can be handled locally at a replica with no inter-replica traffic whatsoever, but writes now involve at least one round-trip to every replica (instead of just the nearest majority).

We argue that there is an overlooked alternative that is a more natural fit to the structure of Paxos: *quorum leases*. In this model, a lease for each object in the system could be granted to different subsets of nodes. The size and composition of these subsets is selected either based upon how frequently each replica reads the objects in question (for best read performance) or based upon their proximity to the leader (to improve read performance without slowing write performance). A particularly suitable size for this subset is that of a Paxos quorum: we claim that quorum leases are a "natural" fit to Paxos because writes in Paxos must, by definition, synchronously contact at least a simple majority of nodes anyway. Thus, the lease revocation and the Paxos messages can be combined, often resulting in no additional overhead or delays for handling a leased object.

1.4 Thesis Contributions

This thesis makes the following contributions:

- The description and evaluation of a new state machine replication protocol, Egalitarian Paxos, which is the first SMR protocol to achieve optimal commit latency in the wide area. Simultaneously, Egalitarian Paxos achieves perfect load balancing across all replicas and constant availability as long as more than half the replicas are reachable (both in the local and wide area deployments).

- A generalization of the concept of read leases for quorum consensus-based state machine replication. Quorum leases allow all replicas to service most read requests locally—thus significantly increasing read performance—with only modest write performance costs.

Chapter 2

Background and Related Work

To put this work in context, we begin with a discussion about strongly-consistent versus weakly-consistent systems. We then describe the Paxos protocol, which is the most widely used mechanism to achieve strong consistency in distributed systems. Finally, we discuss several of the most important Paxos variants and other protocols related to state machine replication, and examine how EPaxos relates to this prior work.

2.1 Strong Versus Weak Consistency

As geo-distributed databases have become widespread, there has been increased attention to the consistency versus latency trade-off in designing distributed systems. This trade-off, as shown by the well-known CAP theorem [9, 19], is a fundamental one. The intuition behind it is straightforward: if we want our replicas to be strongly consistent, then either updates or reads must synchronously go across the wide-area to one or more remote data centers, thus incurring high latency.

Informally, systems that provide strong consistency are easier to interact with, because they behave in an intuitive way: they behave as if the system had only one copy of the data, and all operations modified and read that copy atomically.

Strong consistency commonly refers to the formal concepts of either *strict serializability* or *linearizability* [23]. A system is strictly serializable if the outcome of any sequence of operations, as observed by its clients, is equivalent to a serialization of those operations in which the temporal ordering of non-overlapping operations is respected (i.e., if an operation A is acknowledged by the system before another operation B is proposed by some client, then A comes before B in the equivalent serialization). Linearizability is a sub-case of strict serializability in which every operation reads or updates a single object.[1] The advantage of linearizability is its "local" property: it is sufficient for a system to linearize operations for each individual object to achieve global linearizability. Paxos can be used to achieve strict serializability and linearizability.

Recently, there have been several proposals for reducing wide-area latency by relaxing the consistency model [40, 42, 43, 52]. With these approaches, updates can incur minimal latency by being applied synchronously to only the closest site—this can happen to all updates [42], or to only a subset [40, 52]. The downside is that the state of the replicas can diverge. This divergence is usually temporary, but application developers must carefully consider which divergent mutations are safe and which must be applied consistently (e.g., by falling back to a Paxos-like protocol), or they may have to provide application-specific ways of reconciling divergent state. Irrespective, the users of the application may themselves have to tolerate interactions with the application that are less intuitive.

Even with weak consistency, disaster tolerance (i.e., tolerating the outages of entire data centers with no data loss) still requires synchronous geo-replication to at least one remote site. Understanding the fundamental limitations of synchronous replication is therefore important irrespective of which side of the consistency-versus-latency argument one is situated.

In this context, the primary goal of this work is to make it possible for more applications to have the ease of use and intuitive behavior of a strongly-consistent system even though they may have stringent latency requirements.

2.2 Paxos

State machine replication aims to make a set of possibly faulty distributed processors (the replicas) execute the same commands in the same order. Because each processor is a state machine with no other inputs, all non-faulty processors will transition through the same sequence of states. Given a particular position in the command sequence, running the Paxos algorithm [35] guarantees that, if and when termination is reached, all non-faulty replicas agree on a single command to be assigned that position. To be able to make progress, at most a minority of the replicas can be faulty—if N is the total number

[1] After we define the notion of interference between operations in the context of Egalitarian Paxos, we will be able to extend the notion of linearizability to apply to situations where operation interference is a transitive relation.

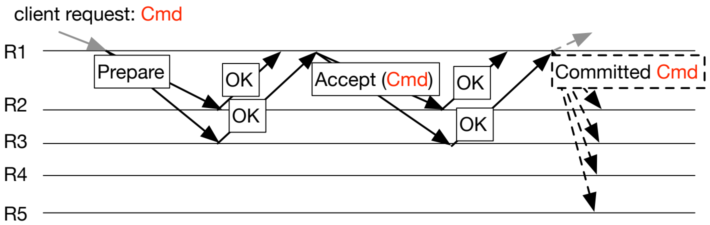


Figure 2.1: The shortest commit path in canonical Paxos (i.e., when there is no contention for a slot) requires two rounds of communication. The dashed arrows in the diagram signify asynchronous messages.

of replicas, at least $\lfloor N/2 \rfloor + 1$ must be non-faulty for Paxos to make progress. Paxos, EPaxos, and other common Paxos variants handle only non-Byzantine failures: a replica may crash, or it may fail to respond to messages from other replicas indefinitely; it cannot, however, respond in a way that does not conform to the protocol.

The execution of a replicated state machine that uses Paxos proceeds as a series of pre-ordered *instances*, where the outcome of each instance is the agreement on a single command. The voting process for one instance may happen concurrently with voting processes for other instances, but does not interfere with them.

Upon receiving a command request from a client, a replica will try to become the *leader* of a not-yet-used instance by sending *Prepare* messages to at least a majority of replicas (possibly including itself). A reply to a *Prepare* contains the command that the replying replica believes may have already been chosen in this instance (in which case the new leader will have to use that command instead of the newly proposed one), and also constitutes a promise not to acknowledge older messages from previous leaders. If the aspiring leader receives at least $\lfloor N/2 \rfloor + 1$ acknowledgements in this prepare phase, it will proceed to *propose* its command by sending it to a majority of peers in the form of *Accept* messages; if these messages are also acknowledged by a majority, the leader commits the command locally, and then asynchronously notifies all its peers and the client. Figure 2.1 depicts these message exchanges.

Because this canonical mode of operation requires at least two rounds of communication (two round trips) to commit a command—and more rounds in the case of dueling leaders—the widely used "Multi-Paxos" optimization designates a replica to be the *stable leader* (or *distinguished proposer*). A replica becomes a stable leader by running the prepare phase for a large (possibly infinite) number of instances at the same time, thus taking ownership of all of them. In steady state, clients send commands only to the stable leader, which directly proposes them in the instances it already owns (i.e., without running the prepare phase). This is depicted in Figure 2.2. When a non-leader replica

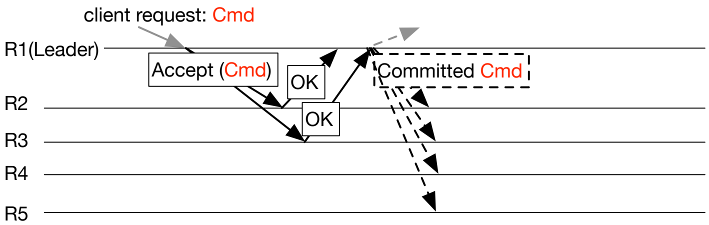


Figure 2.2: The commit path in Multi-Paxos is only one round trip from the leader to a majority.

suspects the leader has failed, it tries to become the new leader by taking ownership of the instances for which it believes commands have not yet been chosen.

2.3 Paxos Variants

There exist many versions of the Paxos protocol, but in practice, Multi-Paxos is by far the most common, being implemented in systems such as Chubby [10], Boxwood [44] and Spanner [16]. The other versions that we describe in this section represent interesting design points for comparing with Egalitarian Paxos.

Multi-Paxos [34, 35] makes efficient forward progress by relying on a stable leader replica that brokers communication with clients and other replicas. With N replicas, for each command, the leader handles $\Theta(N)$ messages, and non-leader replicas handle only $O(1)$. Thus, the leader can become a bottleneck, as practical implementations of Paxos have observed [10]. When the leader fails, the state machine becomes temporarily unavailable until a new leader is elected. This problem is not easily solved: aggressive leader re-election can cause stalls if multiple replicas believe they are the leader.

Mencius [45] distributes load evenly across replicas by rotating the Paxos leader for every command. The instance space is pre-partitioned among all replicas: replica R_{id} owns every instance i where $(i \bmod N) = R_{id}$. The drawback of this approach is that every replica must hear from all other replicas before committing a command A, because otherwise another command B that depends on A may be committed in an instance ordered before the current instance (the other replicas reply either that they are also committing commands for their instances, or that they are skipping their turn). This has two consequences: (1) the replicated state machine runs at the speed of the slowest replica, and (2) Mencius can exhibit worse availability than Multi-Paxos, because if any replica fails to respond, no other replica can make progress until a failure is suspected and another replica commits no-ops on behalf of the possibly failed replica.

Fast Paxos [37] reduces the number of message delays until commands are committed by having clients send commands directly to all replicas. However, some replicas must still act as coordinator and learner nodes, and handle $\Theta(N)$ messages for every command. Like Multi-Paxos, Fast Paxos relies on a stable leader to start voting rounds and arbitrate conflicts (i.e., situations when acceptors order client commands differently, as a consequence of receiving those commands in different orders).

Generalized Paxos [36] commits commands faster by committing them out of order when they do not interfere. Replicas learn commands after just two message delays—which is optimal—as long as they do not interfere. Generalized Paxos requires a stable leader to order commands that interfere, and learners handle $\Theta(N)$ messages for every command.[2] Furthermore, messages become larger as new commands are proposed, so the leader must frequently stop the voting process until it can commit a checkpoint. Multicoordinated Paxos [11] extends Generalized Paxos by using multiple coordinators to increase availability when commands do not conflict, at the expense of using more messages for each command: each client sends its commands to a quorum of coordinators instead of just one. It too relies on a stable leader to ensure consistent ordering if interfering client commands arrive at coordinators in different orders.

In the wide-area, EPaxos has three important advantages over Generalized Paxos: (1) First and foremost, the EPaxos fast-path quorum size is smaller than the fast-path quorum size for Generalized Paxos by exactly one replica, for any total number of replicas—this reduces latency and the overall number of messages exchanged, because a replica must contact fewer of its closest peers to commit a command. (2) Resolving a conflict (two interfering commands arriving at different acceptors in different orders) requires only one additional message delay (i.e., half a round trip) in EPaxos, but will take at least two additional round trips in Generalized Paxos. (3) For three-site replication, EPaxos can commit commands after one round trip to the replica closest to the proposer's site even if all commands conflict. We present the empirical results of this comparison in the next chapter. These advantages make EPaxos a good fit for MDCC [30], which uses Generalized Paxos for wide-area commits.

An important distinction between the fast path in EPaxos and that of Fast and Generalized Paxos is that EPaxos incurs three message delays to commit, whereas Fast and Generalized Paxos require only two. However, in the wide area, the first message delay in EPaxos is usually negligibly short because the client and its closest replica are co-located within the same data center. This distinction allows EPaxos to have smaller fast-path

[2] Based on our experience with EPaxos, we believe it may be possible to modify Generalized Paxos to rotate learners between commands, in the same ballot, to balance load if there are no conflicts. Even so, Generalized Paxos would still depend on the leader for availability.

quorums and has the added benefit of not requiring clients to broadcast their proposals to a super-majority of nodes.

In *S-Paxos* [8], the client-server communication load is shared by all replicas, which batch commands and send them to the leader. The stable leader still handles ordering, so S-Paxos suffers Multi-Paxos's problems in wide-area replication and with slow or faulty leaders.

2.4 Other Protocols Related to Paxos

The original Paxos paper, "*The Part-Time Parliament*", was initially released as a DEC Systems Research Center tech report in 1989 [33] and it built on Lamport's prior work on state machine replication under synchronous communication assumptions [31, 32]. One year prior, Oki and Liskov had published Viewstamped Replication [41, 48], which solved the problem of state machine replication in the presence of failures for an asynchronous setting (i.e., the same problem that Paxos solves). Although submitted for publication in 1990, Paxos was eventually disseminated as a peer-reviewed publication only in 1998 [34]. In the meantime, Chandra and Toueg published a similar protocol called Consensus [14] in 1996. More recently, Zab [27] has gained popularity in practical settings because it is used by ZooKeeper [25], while Raft [49] presented a variant of Viewstamped Replication intended to be more easily understandable. All these protocols have the same communication patterns in normal operation. They are all leader-based, and they differ mainly in the way they handle the change of the leader.

Consistently ordering broadcast messages is equivalent to state machine replication. EPaxos has similarities to generic broadcast algorithms [6, 50, 59], that require a consistent message delivery order only for conflicting messages. Thrifty generic broadcast [6] has the same liveness condition as (E)Paxos, but requires $\Theta(N^2)$ messages for every broadcast message. It relies on atomic broadcast [51] to deliver conflicting messages, which has a latency of four message delays. $\mathcal{GB}$, $\mathcal{GB}+$ [50], and optimistic generic broadcast [59] handle fewer machine failures than (E)Paxos, requiring that more than two thirds of the nodes remain live. They also handle conflicts less efficiently: $\mathcal{GB}$ and $\mathcal{GB}+$ may see conflicts even if messages arrive in the same order at every replica and they use Consensus [14] to solve conflicts; optimistic generic broadcast uses both atomic broadcast and one Consensus instance for every pair of conflicting messages. In contrast, EPaxos requires only one additional one-way message delay to commit commands that interfere; the communication is performed in parallel for all interfering commands; and EPaxos does not need a stable leader to decide the ordering.

While the protocols described so far focus on committing commands efficiently, other systems, such as Eve [28], tackle the orthogonal problem of parallelizing command execution on multi-core systems.

2.5 Protocol Comparison Summary

Table 2.1 compares Egalitarian Paxos and its competitors on a number of relevant dimensions.

The number of critical replicas for a timely response counts those special replicas whose responsiveness is critical for the system to commit commands on the shortest common path. For example, in a leader-based protocol this number is 1 (i.e., the leader itself). For Mencius, all replicas must be responsive for a command to be committed—otherwise the live replicas must time-out and then commit no-ops. EPaxos has no such special replicas.

The quorum for liveness, on the other hand, counts how many replicas in total must remain alive for the system to continue operating without a reconfiguration where crashed replicas are replaced. For some protocols, the commit path is shorter in “good” runs (assumed to be the common case)—this is usually referred to as the *fast path*.

We compare the length of the commit path only for the wide area, because that is where this difference in latency is most relevant. For Egalitarian Paxos, Multi-Paxos and Mencius there is one extra message delay between the client and its closest replica, but this is usually negligible because clients are co-located with their closest replicas in the same data centers.

Protocol	Quorum for liveness	Fast-path quorum	Critical replicas for "timely" response[a]	WAN Fast-path msg. delays to commit	WAN Slow-path msg. delays to commit
Multi-Paxos	$\lfloor N/2 \rfloor + 1$	$\lfloor N/2 \rfloor + 1$	1	3 (2 at leader)	3 (2 at leader)
Fast Paxos [b]	$\lfloor N/2 \rfloor + 1$	$\lceil 3N/4 \rceil$	1	2	3
Generalized Paxos	$\lfloor N/2 \rfloor + 1$	$\lceil 3N/4 \rceil$	1	2	6
Mencius	$\lfloor N/2 \rfloor + 1$	$\lfloor N/2 \rfloor + 1$ to N [c]	N	2	2
Egalitarian Paxos	$\lfloor N/2 \rfloor + 1$	$\lfloor 3N/4 \rfloor$ [d]	0	2	3 [e]
Thrifty generic broadcast	$\lfloor N/2 \rfloor + 1$	$\lfloor N/2 \rfloor + 1$	1	3	6
$\mathcal{GB}, \mathcal{GB}+$	$\lfloor 2N/3 \rfloor + 1$	$\lfloor 2N/3 \rfloor + 1$	1	2	4
Optimistic generic broadcast	$\lfloor 2N/3 \rfloor + 1$	$\lfloor 2N/3 \rfloor + 1$	1	2	3

[a] This counts those special replicas that must reply for commands to be committed on the common shortest path.

[b] Fast Paxos is ill-suited for the wide-area because conflicts occur when different replicas receive commands in different order, so they will be very frequent.

[c] At worst, a Mencius replica may have to wait for all other replicas to reply before it can commit.

[d] $\lfloor 3N/4 \rfloor$ is smaller or equal to $\lfloor 2N/3 \rfloor + 1$ for practical values of N ($N \leq 15$).

[e] Of all the Egalitarian Paxos versions that we present in this thesis, the one that achieves all the properties stated here is EPaxos 3-WAN-Delays. The version that we focus on most in the evaluation, Optimized EPaxos, achieves all but the last property (i.e., it requires 4 message delays on the slow path instead of 3).

Table 2.1: Protocol comparison in the wide area (local area message delays between replicas and co-located clients are ignored). N **is the total number of replicas.**

Chapter 3

Egalitarian Paxos

We begin our presentation of Egalitarian Paxos with the intuition behind the protocol. We then describe the protocol in detail, state its properties and prove them. Finally, we present the results of an empirical comparison between Egalitarian Paxos and previous state machine replication protocols.

In describing EPaxos, we present multiple optimizations, some of which are incompatible. For this reason, we structure the presentation around multiple EPaxos *variants*, starting with a basic variant whose role is mainly to improve understandability.

3.1 Contributions and Intuition

The main goals when designing EPaxos were: (1) optimal commit latency in the wide area, (2) optimal load balancing across all replicas, to achieve high throughput, and (3) graceful performance degradation when some replicas become slow or crash. To achieve these goals, EPaxos must allow all replicas to act as proposers (or *command leaders*) simultaneously, for clients not to waste round trips to remote sites, and functionality to be well balanced across replicas. Furthermore, each proposer must be able to commit a command after communicating with the smallest possible number of remote replicas (i.e., quorums must be as small as possible). Finally, the quorum composition must be flexible, so that command leaders can easily avoid slow or unresponsive replicas.

EPaxos achieves all this due to the novel way in which it orders commands. Previous algorithms ordered commands either by having a single stable leader choose the order (as in Multi-Paxos and Generalized Paxos), or by assigning them to instances (i.e., command

Variant	Described	Evaluated
Basic EPaxos	Section 3.5	No
Optimized EPaxos	Section 3.6	Section 3.11
EPaxos Minimum-Quorum	Section 3.7	No
EPaxos 3-WAN-Delays	Section 3.8	Section 3.11.3

Table 3.1: Summary of Egalitarian Paxos variants described in this thesis.

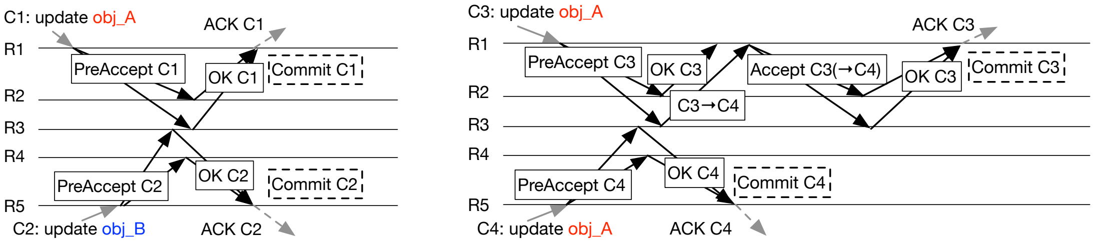


Figure 3.1: EPaxos message flow. R1, R2, ... R5 are the five replicas. Commands C1 and C2 (left) do not interfere, so both can commit on the fast path. C3 and C4 (right) interfere, so one (C3) will be committed on the slow path. C3 → C4 signifies that C3 acquired a dependency on C4. For clarity, we omit the async commit messages.

slots) in a pre-ordered instance space (as in canonical Paxos and Mencius) whereby the order of the commands is the pre-established order of their respective slots. In contrast, EPaxos orders the instances dynamically and in a decentralized fashion: in the process of choosing (i.e., voting) a command in an instance, each participant attaches ordering constraints to that command. EPaxos guarantees that all non-faulty replicas will commit the same command with the same constraints, so every replica can use these constraints to independently reach the same ordering.

This ordering approach is the source of the benefits EPaxos has over previous algorithms. First, committing a command is contingent upon the input of any majority of replicas, unlike in Multi-Paxos where the stable leader must be part of every decision, or in Mencius, where information from all replicas is required. This benefits wide-area commit latency, availability, and also improves performance robustness, because it decouples the performance of the fastest replicas from that of the slowest. Second, any replica can propose a command, not just a distinguished proposer, or leader—this allows for load balancing, which increases throughput.

In taking this ordering approach, EPaxos must maintain safety and provide a linearizable ordering of commands, while minimizing both the number of replicas that must participate in voting for each command and the number of messages exchanged between them. One observation that makes this task easier—by substantially reducing the number of ordering constraints in the common case—was made by generic broadcast algorithms and Generalized Paxos before us: it is not necessary to enforce a consistent ordering for the common case of commands that do not interfere with each other.

Figure 3.1 presents a simplified example of how Egalitarian Paxos works. Commands can be sent by clients to any replica—we call this replica the *command leader* for that command, not to be confused with the stable leader in Multi-Paxos. In practical workloads, concurrent proposals interfere only rarely (for now, think of this common case as concurrent commands that update different objects). EPaxos can commit these commands after only one round of communication between the command leader and a *fast-path quorum* of peers—$F + \left\lfloor \frac{F+1}{2} \right\rfloor$ replicas in total, including the command leader, where F is the number of tolerated failures ($F = 2$ in the example from Figure 3.1).

When commands interfere, they acquire *dependencies* on each other—attributes that commands are committed with, used to determine the correct order in which to execute the commands (the commit and the execution orders are not necessarily the same, but this does not affect correctness). To ensure that every replica commits the same attributes even if there are failures, a second round of communication between the command leader and a classic quorum of peers—$F + 1$ replicas including the command leader—may be required (as in Figure 3.1 for command C3). We call this the *slow path*. As an optimization, we can overlap the two rounds of communication to commit commands after only three message delays (one and a half round trips) on the slow path.

3.2 Preliminaries

We present here our assumptions about the setting where Egalitarian Paxos is used and we introduce our notation.

Messages exchanged by processes (clients and replicas) are asynchronous. Failures are non-Byzantine—a machine can fail by stopping to respond for an indefinite amount of time. The replicated state machine comprises $N = 2F + 1$ replicas, where F is the maximum number of tolerated failures. For every replica R there is an unbounded sequence of numbered command slots $R.1$, $R.2$, $R.3$, ... that replica R is said to *own*. As is customary in the Paxos literature, we call these command slots *protocol instances* [35] (or simply instances). The complete state of each replica comprises all the instances owned by every replica in the system (i.e., for N replicas, the state of each replica can be regarded as a two-dimensional array with N rows and an unbounded number of columns). At most one command will be chosen in an instance. The ordering of the instances is *not pre-determined*—it is determined dynamically by the protocol, as commands are chosen.

It is important to understand that *committing* and *executing* commands are different actions, and that the commit and execution orders are not necessarily the same.

To modify the underlying application state, a client sends *Request(command)* to a replica of its choice. A *RequestReply* from that replica will notify the client that the command has been committed. However, the client has no information about whether the command has been executed or not. Only when the client reads the application state updated by previously committed commands is it necessary for the system to execute those commands.

To read (part of) the state, clients send *Read(objectIDs)* messages and wait for *ReadReplies*. *Read* is a no-op command that *interferes* with updates to the objects it is reading. Clients can also use *RequestAndRead(γ, objectIDs)* to propose command γ and atomically read the machine state immediately after γ is executed—*Read(objectIDs)* is equivalent to *RequestAndRead(no-op, objectIDs)*.

3.3 Command Interference

Before we can describe EPaxos in detail, we must define **command interference**.

Informally, two commands that interfere must be executed in the same order by all replicas—otherwise the states of the replicas will not converge. Commands that do not interfere are those that can commute.

Definition 1 (Interference). Two commands γ and δ *interfere* if there exists a sequence of commands Σ such that the serial execution Σ, γ, δ is not compatible with the serial execution Σ, δ, γ (i.e., it results in a different underlaying application state and/or different read results).

As we explain in the following section, EPaxos guarantees that any two interfering commands will be executed in the same order with respect to each other on every replica. This is sufficient to guarantee that the executions on all replicas are compatible: the serial ordering of commands on a replica can be obtained from that of any other replica by commuting commutative commands.

Note that the interference relation is symmetric, but not necessarily transitive.

3.4 Protocol Guarantees

The formal guarantees that EPaxos offers clients are similar to those provided by other Paxos variants:

Nontriviality: Any command committed by any replica must have been proposed by a client.

Stability: For any replica, the set of committed commands at any time is a subset of the committed commands at any later time. Furthermore, if at time t_1 a replica R has command γ committed at some instance $Q.i$, then R will have γ committed in $Q.i$ at any later time $t_2 > t_1$.

Consistency: Two replicas can never have different commands committed for the same instance.

Execution consistency: If two interfering commands γ and δ are successfully committed (by any replicas) they will be executed in the same order by every replica.

Execution linearizability: If two interfering commands γ and δ are serialized by clients (i.e., δ is proposed only after γ is committed by any replica), then every replica will execute γ before δ.

Execution consistency and execution linearizability imply the classic notion of linearizability [23]: consistency and the definition of command interference imply serializability, which along with execution linearizability imply classic linearizability.

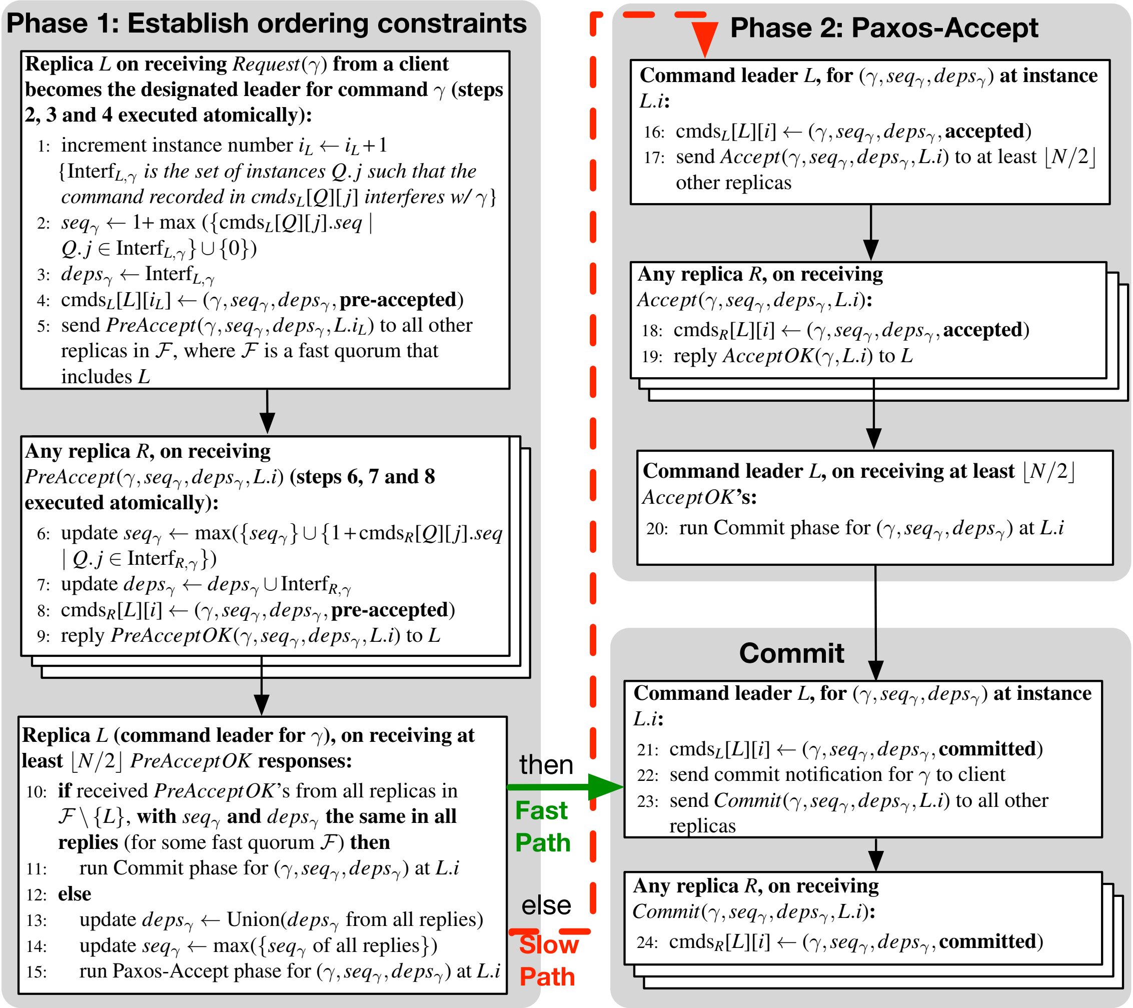
Liveness (w/ high probability): Commands will eventually be committed by every non-faulty replica, as long as fewer than half the replicas are faulty and messages eventually go through before recipients time out.[1]

3.5 The Basic Protocol

For clarity, we first describe the basic Egalitarian Paxos, and improve on it in the next section. This basic EPaxos uses a simplified procedure to recover from failures, and as a consequence, its *fast-path quorum*[2] is $2F$ (out of the total of $N = 2F + 1$ replicas). The

[1] Paxos provides the same liveness guarantees. By FLP [18], it is impossible to provide stronger guarantees for distributed consensus with a practical (i.e., deterministic) protocol.

[2] We use *quorum* to denote both a set of replicas with a particular cardinality, and the cardinality of that set.



Replica L on receiving $Request(\gamma)$ from a client becomes the designated leader for command γ (steps 2, 3 and 4 executed atomically):

```
1:  increment instance number $i_L \leftarrow i_L + 1$
    $\{\text{Interf}_{L,\gamma}$ is the set of instances $Q.j$ such that the command recorded in $cmds_L[Q][j]$ interferes w/ $\gamma\}$
2:  $seq_\gamma \leftarrow 1 + \max(\{\text{cmds}_L[Q][j].seq \mid Q.j \in \text{Interf}_{L,\gamma}\} \cup \{0\})$
3:  $deps_\gamma \leftarrow \text{Interf}_{L,\gamma}$
4:  $\text{cmds}_L[L][i_L] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{pre-accepted})$
5:  send $PreAccept(\gamma, seq_\gamma, deps_\gamma, L.i_L)$ to all other replicas in $\mathcal{F}$, where $\mathcal{F}$ is a fast quorum that includes $L$
```

Any replica R, on receiving $PreAccept(\gamma, seq_\gamma, deps_\gamma, L.i)$ (steps 6, 7 and 8 executed atomically):

```
6:  update $seq_\gamma \leftarrow \max(\{seq_\gamma\} \cup \{1 + \text{cmds}_R[Q][j].seq \mid Q.j \in \text{Interf}_{R,\gamma}\})$
7:  update $deps_\gamma \leftarrow deps_\gamma \cup \text{Interf}_{R,\gamma}$
8:  $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{pre-accepted})$
9:  reply $PreAcceptOK(\gamma, seq_\gamma, deps_\gamma, L.i)$ to $L$
```

Replica L (command leader for γ), on receiving at least $\lfloor N/2 \rfloor$ $PreAcceptOK$ responses:

```
10: if received $PreAcceptOK$'s from all replicas in $\mathcal{F} \setminus \{L\}$, with $seq_\gamma$ and $deps_\gamma$ the same in all replies (for some fast quorum $\mathcal{F}$) then
11:     run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
12: else
13:     update $deps_\gamma \leftarrow \text{Union}(deps_\gamma$ from all replies)
14:     update $seq_\gamma \leftarrow \max(\{seq_\gamma$ of all replies$\})$
15:     run Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
```

Command leader L, for $(\gamma, seq_\gamma, deps_\gamma)$ at instance $L.i$:

```
16: $\text{cmds}_L[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{accepted})$
17: send $Accept(\gamma, seq_\gamma, deps_\gamma, L.i)$ to at least $\lfloor N/2 \rfloor$ other replicas
```

Any replica R, on receiving $Accept(\gamma, seq_\gamma, deps_\gamma, L.i)$:

```
18: $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{accepted})$
19: reply $AcceptOK(\gamma, L.i)$ to $L$
```

Command leader L, on receiving at least $\lfloor N/2 \rfloor$ $AcceptOK$'s:

```
20: run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
```

Command leader L, for $(\gamma, seq_\gamma, deps_\gamma)$ at instance $L.i$:

```
21: $\text{cmds}_L[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{committed})$
22: send commit notification for $\gamma$ to client
23: send $Commit(\gamma, seq_\gamma, deps_\gamma, L.i)$ to all other replicas
```

Any replica R, on receiving $Commit(\gamma, seq_\gamma, deps_\gamma, L.i)$:

```
24: $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{committed})$
```

Figure 3.2: The basic Egalitarian Paxos protocol for choosing commands.

Explicit Prepare

```
Replica $Q$ for instance $L.i$ of potentially failed replica $L$
25: increment ballot number to $epoch.(b+1).Q$, (where $epoch.b.R$ is the highest ballot number $Q$ is aware of in instance $L.i$)
26: send $Prepare(epoch.(b+1).Q, L.i)$ to all replicas (including self) and wait for at least $\lfloor N/2 \rfloor + 1$ replies
27: let $\mathcal{R}$ be the set of replies w/ the highest ballot number
28: if $\mathcal{R}$ contains a $(\gamma, seq_\gamma, deps_\gamma, \text{committed})$ then
29:     run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
30: else if $\mathcal{R}$ contains an $(\gamma, seq_\gamma, deps_\gamma, \text{accepted})$ then
31:     run Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
32: else if $\mathcal{R}$ contains at least $\lfloor N/2 \rfloor$ identical replies $(\gamma, seq_\gamma, deps_\gamma, \text{pre-accepted})$ for the default ballot $epoch.0.L$ of instance $L.i$, and none of those replies is from $L$ then
33:     run Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
34: else if $\mathcal{R}$ contains at least one $(\gamma, seq_\gamma, deps_\gamma, \text{pre-accepted})$ then
35:     start Phase 1 (at line 2) for $\gamma$ at $L.i$, avoid fast path
36: else
37:     start Phase 1 (at line 2) for no-op at $L.i$, avoid fast path

Replica $R$, on receiving $Prepare(epoch.b.Q, L.i)$ from $Q$
38: if $epoch.b.Q$ is larger than the most recent ballot number $epoch.x.Y$ accepted for instance $L.i$ then
39:     reply $PrepareOK(\text{cmds}_R[L][i], epoch.x.Y, L.i)$
40: else
41:     reply NACK
```

Figure 3.3: The EPaxos simplified recovery procedure.

fully optimized EPaxos reduces this quorum to only $F + \left\lfloor \frac{F+1}{2} \right\rfloor$ replicas. The slow-path quorum size is always $F + 1$.

3.5.1 The Commit Protocol

As mentioned earlier, committing and executing commands are separate. Accordingly, EPaxos comprises (1) the protocol for choosing (committing) commands and determining their ordering attributes; and (2) the algorithm for executing commands based on these attributes.

Figure 3.2 shows the pseudocode of the basic protocol for choosing commands. Each replica's state is represented by its private *cmds* log that records all commands seen (but not necessarily committed) by the replica.

We split the description of the commit protocol into multiple phases. Not all phases are executed for every command: a command committed after Phase 1 and Commit was committed on the *fast path*. The *slow path* involves the additional Phase 2 (the Paxos-Accept phase). Explicit Prepare (Figure 3.3) is run only on failure recovery.

Phase 1 starts when a replica L receives a request for a command γ from a client and becomes a command leader. L begins the process of choosing γ in the next available instance of its instance sub-space. It also attaches what it believes are the correct attributes for that command:

deps is the list of all instances that contain commands (not necessarily committed) that interfere with γ; we say that γ *depends* on those instances and their corresponding commands;

seq is a sequence number used to break dependency cycles during the execution algorithm; *seq* is updated to be larger than the *seq* of all interfering commands in *deps*.

The command leader forwards the command and the initial attributes to at least a fast-path quorum of replicas as a *PreAccept* message. Each replica, upon receiving the *PreAccept*, updates γ's *deps* and *seq* attributes according to the contents of its *cmds* log, records γ and the new attributes in the log, and replies to the command leader.

If the command leader receives replies from enough replicas to constitute a fast-path quorum, and all the updated attributes are the same, it commits the command. If it does not receive enough replies, or the attributes in some replies have been updated differently than in others, then the command leader updates the attributes based upon a simple majority ($\lfloor N/2 \rfloor + 1 = F + 1$) of replies (taking the union of all *deps*, and the highest *seq*), and tells at least a majority of replicas to accept these attributes. This can be seen as running classic Paxos to choose the triplet (γ, *deps*$_\gamma$, *seq*$_\gamma$) in γ's instance. At the end of this extra round, after replies from a majority (including itself), the command leader will reply to the client and send *Commit* messages asynchronously to the other replicas.

As in classic Paxos, every message contains a ballot number (for simplicity, we represent it explicitly in our pseudocode only when describing the Explicit Prepare phase in Figure 3.3). The ballot number ensures message freshness: replicas disregard messages with a ballot that is smaller than the largest they have seen for a certain instance. For correctness, ballot numbers used by different replicas must be distinct, so they include a replica ID. Furthermore, a newer configuration of the replica set must have strict precedence over an older one, so we also prepend an *epoch* number (epochs are explained in Section 3.10). The resulting ballot number format is *epoch*$.b.R$, where a replica R increments only the natural number b when trying to initiate a new ballot in Explicit Prepare. Each replica is the default (i.e., initial) leader of its own instances, so the ballot *epoch*$.0.R$ is implicit at the beginning of every instance $R.i$.

3.5.2 The Execution Algorithm

To execute command γ committed in instance $R.i$, a replica will follow these steps:

1. Wait for $R.i$ to be committed (or run Explicit Prepare to force it);

2. Build γ's dependency graph by adding γ and all commands in instances from γ's dependency list as nodes, with directed edges from γ to these nodes, repeating this process recursively for all of γ's dependencies (starting with step 1);

3. Find the strongly connected components, sort them topologically;

4. In inverse topological order, for each strongly connected component, do:

    4.1 Sort all commands in the strongly connected component by their sequence number;

    4.2 Execute every un-executed command in increasing sequence number order, marking them executed.

L, R, Q	Replicas (the command leader is usually denoted by L)
γ, δ	Commands
$R.i$, with $i = 1, 2, ...$	Instances belonging to replica R
epoch$.i.R$, with *epoch*$, i = 0, 1, 2, ...$	Ballot numbers generated by replica R (*epoch*$.0.R$ is the initial ballot for any instance $R.i$)
deps$_\gamma$	The list of dependencies for command γ
seq$_\gamma$	Approximate sequence number for command γ
cmds$_Q[R][i]$	The state of replica Q at instance $R.i$

Table 3.2: Summary of notation.

3.5.3 Informal Proof of Properties

Together, the commit protocol and execution algorithm guarantee the properties stated in Section 3.4. We prove this formally in a later section, but give informal proofs here to convey the intuition of our design choices.

Nontriviality is straightforward: Phase 1 is only executed for commands proposed by clients.

To prove stability and consistency, we first prove:

Proposition 1. *If replica R commits command γ at instance $Q.i$ (with R and Q not necessarily distinct), then for any replica R' that commits command γ' at $Q.i$ it must hold that γ and γ' are the same command.*

Proof sketch. Command γ is committed at instance $Q.i$ only if replica Q has started Phase 1 for γ at instance $Q.i$. Q cannot start Phase 1 for different commands at the same instance, because (1) Q increments its instance number for every new command, and (2) if Q fails and restarts, it will be given a new, unused identifier (Section 3.10). □

The proposition implies **consistency**. Furthermore, because commands can be forgotten only if a replica crashes, it also implies **stability** if the *cmds* log is maintained on persistent storage. Execution consistency also requires stability and consistency for the command attributes.

Definition. If γ is a command with attributes *seq*$_\gamma$ and *deps*$_\gamma$, we say that the tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ is *safe* at instance $Q.i$ if $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ is the only tuple that is or will be committed at $Q.i$ by any replica.

Proposition 2. *Replicas commit only safe tuples.*

Proof sketch. A tuple $(\gamma, \textit{seq}_{\gamma}, \textit{deps}_{\gamma})$ can only be committed at a certain instance $Q.i$ (1) after the Paxos-Accept phase, or (2) directly after Phase 1.

Case 1: A tuple is committed after the Paxos-Accept phase if more than half of the replicas have logged the tuple as accepted (line 20 in Figure 3.2). The tuple is safe via the classic Paxos algorithm guarantees.

Case 2: A tuple is committed directly after Phase 1 only if its command leader receives identical responses from $N-2$ other replicas (line 11). The tuple is now safe: If another replica tries to take over the instance (because it suspects the initial leader has failed), it must execute the *Prepare* phase and it will see at least $\lfloor N/2 \rfloor$ identical replies containing $(\gamma, \textit{seq}_{\gamma}, \textit{deps}_{\gamma})$, so the new leader will identify this tuple as potentially committed and will use it in the Paxos-Accept phase.

So far, we have shown that tuples, including their attributes, are committed consistently across replicas. They are also stable, if recorded on persistent storage. □

We next show that these consistent, stable committed attributes guarantee that all interfering commands are executed in the same order on every replica:

Execution consistency If interfering commands γ and δ are successfully committed (not necessarily by the same replica), they will be executed in the same order by every replica.

Proof sketch. If two commands interfere, at least one will have the other in its dependency set by the time they are committed: Phase 1 ends after the command has been pre-accepted by at least a simple majority of the replicas, and its final set of dependencies is the union of at least the set of dependencies updated at a majority of replicas. This also holds for recovery (line 32 in the pseudocode) because all dependencies are based on those set initially by the possibly failed leader. Thus, at least one replica pre-accepts both γ and δ, and its *PreAcceptReplies* are taken into account when establishing the final dependencies sets for both commands.

By the execution algorithm, a command is executed only after all the commands in its dependency graph have been committed. There are three possible scenarios:

Case 1: Both commands are in each other's dependency graph. By the way the graphs are constructed, this implies: (1) the dependency graphs are identical; and (2) γ and δ are in the same strongly connected component. Therefore, when executing one command, the other is also executed, and they are executed in the order of their sequence numbers (with arbitrary criteria to break ties). By Proposition 2 the attributes of all committed commands are stable and consistent across replicas, so all replicas build the same dependency graph and execute γ and δ in the same order.

Case 2: γ is in δ's dependency graph but δ is not in γ's. There is a path from δ to γ in δ's dependency graph, but there is no path from γ to δ. Therefore, γ and δ are in different strongly connected components, and γ's component will come before δ's in inverse topological order. By the execution algorithm, γ will be executed before δ. This is consistent with the situation when γ had been executed on some replicas before δ was committed (which is possible, because γ does not depend on δ).

Case 3: Just like case 2, with γ and δ reversed. □

Execution linearizability If two interfering commands γ and δ are serialized by clients (i.e., δ is proposed only after γ is committed by any replica), then every replica will execute γ before δ.

Proof sketch. Because δ is proposed after γ was committed, γ's sequence number is stable and consistent by the time any replica receives *PreAccept* messages for δ. Because a tuple containing γ and its final sequence number is logged by at least a majority of replicas, δ's sequence number will be updated to be larger than γ's, and δ will contain γ in its dependencies. Therefore, when executing δ, δ's graph must contain γ either in the same strongly connected component as δ (but δ's sequence number will be higher), or in a component ordered before δ's in inverse topological order. Regardless, by the execution algorithm, γ will be executed before δ. □

Finally, **liveness** is ensured as long as a majority of replicas are non-faulty. A client keeps retrying a command until a replica gets a majority to accept it.

3.5.4 Keeping the Dependency List Small

Instead of including all interfering instances, we include only N dependencies in each list: the instance number $R.i$ with the highest i for which the current replica has seen an interfering command (not necessarily committed). If interference is transitive (usually the case in practice) the most recent interfering command suffices, because its dependency graph will contain all interfering instances $R.j$, with $j < i$. Otherwise, every replica must assume that any unexecuted commands in previous instances $R.j$ ($j < i$) are possible dependencies and independently check them at execute time. This is a fast operation when commands are executed soon after commit.

3.5.5 Recovering from Failures

A replica may need to learn the decision for an instance because it has to execute commands that depend on that instance. If a replica times out waiting for the commit for an instance, the replica will try to take ownership of that instance by running *Explicit Prepare*, at the end of which it will either learn what command was proposed in this problem instance (and then finalize committing it), or, if no other replica has seen a command, will commit a no-op to finalize the instance.

If clients are allowed to time-out and re-issue commands to a different replica, the replicas must be able to recognize duplicates and execute the command only once. This situation affects any replication protocol, and standard solutions are applicable, such as unique command IDs or ensuring that commands are idempotent.

3.5.6 Avoiding Execution Livelock

With a fast stream of interfering proposals, command execution could livelock: command γ will acquire dependencies on newer commands proposed between sending and receiving the *PreAccept*(γ). These new commands in turn gain dependencies on even newer commands. To prevent this, we prioritize completing old commands over proposing new commands. Even without this optimization, however, long dependency chains increase only execution latency, not commit latency. They also negligibly affect throughput, because executing a batch of n inter-dependent commands at once adds only modest computational overhead: finding the strongly connected components has linear time complexity (the number of dependencies for each command is usually constant—Section 3.5.4), and sorting the commands by their sequence attribute adds only an $O(\log n)$ factor.

3.5.7 Read Leases

As in any other state machine replication protocol, a *Read* must be committed as a command that interferes with updates to the objects it is reading to avoid reading stale data. However, Paxos-based systems are often optimized for read-heavy scenarios in one of two ways: assume the clients can handle stale data and perform reads locally at any replica, as in ZooKeeper [25]; or grant a read lease to the stable leader so that it can respond without committing an operation [13]. EPaxos can use read leases just as easily, with the understanding that a (infrequent) write to the leased object must be channeled through the node holding the lease. In wide-area replication, the leaderless design of EPaxos and Mencius allows different sites to hold leases for different objects simultaneously (e.g., based on the observed demand for each object). This property makes EPaxos and Mencius excellent fits for *quorum read leases*, which we introduce in the next chapter, because it means that these protocols can retain their high write performance even with read leases.

3.5.8 Formal Proofs of Properties

We prove that together, the commit protocol and execution algorithm guarantee the properties stated in Section 3.4.

Theorem 1 (Nontriviality). *Any command committed by any EPaxos replica must have been proposed by a client.*

Proof: For any command that reaches the Commit phase, a replica must have executed the Init phase. Init is only executed for commands proposed by clients. □

Definition 2. If γ is a command with attributes seq_γ and $deps_\gamma$, we say that the tuple $(\gamma, seq_\gamma, deps_\gamma)$ is *safe* at instance $Q.i$ if $(\gamma, seq_\gamma, deps_\gamma)$ is the only tuple that is or will be committed at $Q.i$ by any replica.

Lemma 1. *EPaxos replicas commit only safe tuples.*

Proof:

1 The same ballot number cannot be used twice in the same instance.

PROOF:

1.1 No two different replicas can use the same ballot number.

PROOF: The ballot number chosen by a replica is based on its id, which is unique.

1.2 A replica never uses the same ballot number twice for the same instance

PROOF:

1.2.1 *Case:* If replicas store the command log in persistent memory, then a replica will never reinitiate the same instance twice with the same ballot number.

1.2.2 *Case:* If a crashed replica can forget the command log, it will be assigned a new id when it recovers.

1.2.3 *Q.E.D.*
Cases 1.2.1 and 1.2.2 are exhaustive.

1.3 *Q.E.D.*
Immediately from 1.1 and 1.2.

2 For any instance $Q.i$ there is at most one attempt (i.e., the *default* ballot $0.Q$) to choose a tuple without running Explicit Prepare first.

PROOF:

2.1 A replica Q starts an instance $Q.i$ at most once.

PROOF: A replica starts an instance only in the Init phase of the algorithm and it increments the instance number atomically every time it executes Init. The instance number never decreases. If a replica loses the content of its memory (e.g., after a crash), it will be assigned a previously unused replica id by a safe external configuration service—so the same instance can never be started twice.

2.2 No replica other than Q can start instance $Q.i$.

PROOF:

2.2.1 A replica with a different id $R \neq Q$ starts only instances $R.i \neq Q.i$

2.2.2 A new replica is never assigned the id of a previously started replica

2.2.3 *Q.E.D.*
Immediately from 2.2.1 and 2.2.2.

2.3 *Q.E.D*

When not running Explicit Prepare, a replica tries to choose a command in an instance only if it starts that instance, and only for the default ballot. By 2.1 and 2.2, this can happen at most once per instance $Q.i$, in ballot $0.Q$.

3 Let $b_{smallest}$ be the smallest ballot number for which a tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ has been committed at instance $Q.i$. Then any other commit at instance $Q.i$ commits the same tuple.

PROOF:

By induction on the ballot number b of all ballots committed for $Q.i$:

3.1 Base case: if $b = b_{smallest}$, then the same tuple is committed in both b and $b_{smallest}$.

Proof:

By 1, b and $b_{smallest}$ must be the same ballots.

3.2 Induction step: if tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ has been committed in ballot b_1, then the next higher successful ballot $b > b_1$ will commit the same tuple.

Proof:

Let b_2 be the next highest ballot number of a ballot attempted at instance $Q.i$. By 2, and since b_2 cannot be the default ballot for $Q.i$ (because there is a ballot b_1 smaller than it), b_2 is attempted after running Explicit Prepare. Furthermore, by the recovery procedure, any ballot attempted after Explicit Prepare must run the Paxos-Accept Phase.

3.2.1 *Case:* Ballot b_1 is committed directly after Phase 1.

Since b_1 is successful after Phase 1, then a fast quorum ($N - 1$ replicas) have recorded the same tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ for instance $Q.i$. For b_2 to start, its leader must receive replies to *Prepare* messages from at least $\lfloor N/2 \rfloor + 1$ replicas. Therefore, at least $\lfloor N/2 \rfloor$ replicas will see a *Prepare* for b_2 *after* they have recorded $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ for ballot b_1 (if they had seen the larger ballot b_2 first, they would not have acknowledged any message for ballot b_1). b_2's leader will therefore receive at least $\lfloor N/2 \rfloor$ *PrepareReplies* with tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ marked as pre-accepted.

If the leader of b_1 is among the replicas that reply to the *Prepare* of ballot b_2, then it must have replied after the end of Phase 1 (otherwise it couldn't have completed the smaller ballot b_1), so it will have committed tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ by then. The leader of b_2 will then know it is safe to commit the same tuple.

Below, we assume that the leader of b_1 is *not* among the replicas that reply to the *Prepare* of ballot b_2.

3.2.1.1 *Subcase:* $N > 3$

The $\lfloor N/2 \rfloor$ replies with tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ constitute a majority among the first $\lfloor N/2 \rfloor + 1$ *PrepareReplies*. The leader of ballot b_2, will therefore be able to identify tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ as potentially committed, and use it in a Paxos-Accept Phase.

3.2.1.2 *Subcase:* $N = 3$

$\lfloor N/2 \rfloor = 1$ is not a majority among the first $\lfloor N/2 \rfloor + 1 = 2$ *PrepareReplies*. However, for $N = 3$, a command leader commits a tuple after Phase 1 only if a *PreAcceptReply* matched the attributes in the initial *PreAccept*. The acceptor that has sent such a *PreAcceptReply* in ballot b_1 will convey this information in a *PrepareReply* for ballot b_2. The leader of ballot b_2 will therefore use the correct tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ in a Paxos-Accept Phase.

For ballots higher than b_2 to start, their leaders will follow the recovery procedure, and will receive either the same type of replies received by the leader of b_2 (as above), or it will receive at least one *PrepareReply* from a replica whose highest ballot is b_2 and has marked $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ as accepted. In either case, by the recovery procedure, the replica trying to take over instance $Q.i$ will have to use tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ in a Paxos-Accept Phase. By simple induction, any ballot higher than b_1 will use tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ in a Paxos-Accept Phase, including successful ballots.

3.2.2 *Case:* Ballot b_1 is committed after the Paxos-Accept Phase.

The tuple $(\gamma, \mathit{seq}_\gamma, \mathit{deps}_\gamma)$ is safe by the guarantees of classic Paxos.

3.2.3 *Q.E.D.*

Cases 2.2.1 and 2.2.2 are exhaustive.

3.3 *Q.E.D.*

The induction is complete.

4 *Q.E.D.*

Immediately from 3.

□

Theorem 2 (Consistency). *Two replicas can never have different commands committed for the same instance.*

Proof: We have already proved a stronger property: by Lemma 1, two replicas can never have different tuples (i.e., commands along with their commit attributes) committed for the same instance. □

Theorem 3 (Stability). *For any replica, the set of committed commands at any time is a subset of the committed commands at any later time. Furthermore, if at time t_1 a replica R has command γ committed at some instance $Q.i$, then R will have γ committed in $Q.i$ at any later time $t_2 > t_1$.*

Proof: By Theorem 2 and the extra assumption that committed commands are recorded in persistent memory. □

So far, we have shown that tuples are committed consistently across replicas. They are also stable, as long as they are recorded in persistent memory. We now show that having consistent attributes committed across all replicas is sufficient to guarantee that all interfering commands are executed in the same order on every replica:

Theorem 4 (Execution consistency). *If two interfering commands γ and δ are successfully committed (not necessarily by the same replica), they will be executed in the same order by every replica.*

Proof:

1 If γ and δ are successfully committed and $\gamma \sim \delta$, then either γ has δ in its dependency list when γ is committed (more precisely, γ has δ's instance in its dependency list, but, for simplicity of notation, we use a command name to denote the pair comprising the command and the specific instance in which it has been committed), or δ has γ in its dependency list when δ is committed.

PROOF:

1.1 The attributes with which a command c is committed are the union of at least $\lfloor N/2 \rfloor + 1$ sets of attributes computed by as many replicas.

PROOF:

1.1.1 *Case:* c is committed immediately after Phase 1.
$N - 1$ replicas have input their attributes for c.

1.1.2 *Case:* c is committed after the Paxos-Accept phase.

1.1.2.1 *Subcase:* The Paxos-Accept phase starts after the execution of Phase 1.
Phase 1 ends after $\lfloor N/2 \rfloor$ replicas have replied to a *PreAccept* with the command leader's updated attributes (so the attributes are the union of $\lfloor N/2 \rfloor + 1$ sets of attributes, from as many replicas, including the command leader).

1.1.2.2 *Subcase:* The Paxos-Accept phase starts after $\lfloor N/2 \rfloor$ *PrepareReplies* in the recovery phase, none of which is from the initial command leader.

Then $\lfloor N/2 \rfloor$ replicas, plus the initial command leader ($\lfloor N/2 \rfloor + 1$ replicas in total), have contributed to the set of attributes used for the subsequent Paxos-Accept phase.

1.1.2.3 *Subcase:* The Paxos-Accept phase starts after a *PrepareReply* from a replica R that had marked c as accepted.
Then some replica has to have previously initiated the Paxos-Accept phase that resulted in R receiving an *Accept*, so this subcase is reducible to one of the previous subcases.

1.1.2.4 *Q.E.D.*
The subcases enumerated above describe all possible circumstances in which a command is committed after the Paxos-Accept Phase.

1.1.3 *Case:* γ is committed after the current replica receives a *Commit* for γ from another replica.
The replica that initiates the *Commit* must be in one of the previous two cases.

1.1.4 *Q.E.D.*
The cases enumerated above are exhaustive.

1.2 *Q.E.D.*

By 1.1, at least one replica R contributes for both γ's and δ's final attributes. Because R records every command that it sees in its command log, and because $\gamma \sim \delta$, R will include the command it sees first in the dependency list of the command it sees second.

2 *Q.E.D.*

By 1, the final dependency *graphs* of γ and δ are in one of three cases:

2.1 *Case:* γ and δ are both in each other's dependency graph.

Then, by the execution algorithm, γ and δ have each other in their dependency graphs, and moreover, they are in the same strongly connected components of their respective graphs. By the execution algorithm, whenever one command is executed, the other is also executed. Since the execution algorithm is deterministic, and since, by Lemma 1, every replica builds the same dependency graphs for γ and δ, every replica will execute the commands in the same order.

2.2 *Case:* γ is in δ's dependency graph, but δ is not in γ's dependency graph.

The commands are in different strongly connected components in δ's graph, and δ's component is ordered after γ's component in reversed topological order.

We show that γ is executed before δ by every replica:

2.2.1 *Subcase:* A replica tries to execute γ first.
The replica will execute γ without having executed δ.

2.2.2 *Subcase:* A replica tries to execute δ first.
By the execution algorithm, the replica will build δ's dependency graph, which also contains γ in a strongly connected component that is ordered before δ's component in reversed topological order. Then γ is executed before δ is executed.

2.3 *Case:* δ is in γ's dependency graph, but γ is not in δ's dependency graph.

Just like the previous case, with γ and δ interchanged.

2.4 *Q.E.D.*

The above three cases are exhaustive. In all cases, the commands are executed in the same order by every replica.

□

Theorem 5 (Execution linearizability). *If two interfering commands γ and δ are serialized by clients (i.e., δ is proposed only after γ is committed by any replica), then every replica will execute γ before δ.*

Proof:

1 γ will be in δ's dependency graph.

PROOF:

By the time δ is proposed, γ will have been pre-accepted by at least $\lfloor N/2 \rfloor + 1$ replicas. For δ to be committed, it too has to be pre-accepted by at least $\lfloor N/2 \rfloor + 1$ replicas. Therefore, at least one replica R whose pre-accept is taken into account when establishing δ's dependency list pre-accepts δ after it has pre-accepted γ. Since $\gamma \sim \delta$, R will put γ in δ's dependency list.

2 The sequence number with which δ is committed will be higher than that with which γ is committed.

PROOF:

2.1 By the time any replica receives a request for δ from a client, at least $\lfloor N/2 \rfloor + 1$ replicas will have logged the final sequence number for γ.

PROOF:

2.1.1 *Case*: γ is committed directly after Phase 1.
Then $N - 1$ replicas have logged the same sequence number for γ, and this is the sequence number with which γ is committed.

2.1.2 *Case*: γ is committed after the Paxos-Accept Phase.
Then at least $\lfloor N/2 \rfloor + 1$ replicas have logged γ as accepted with its final attributes, including its sequence number.

2.1.3 *Q.E.D.*
Cases 2.1.1 and 2.1.2 are exhaustive.

2.2 *Q.E.D.*
By 2.1, at least one of the replicas that pre-accepts δ, whose *PreAcceptReply* is taken into account when establishing δ's final attributes, will update δ's sequence number to be higher than γ's final sequence number.

3 *Q.E.D.*

At any replica R, there are two possible cases:

3.1 *Case*: R tries to execute γ before it tries to execute δ.

3.1.1 *Subcase*: δ is in γ's dependency graph.
Then, by 1, δ and γ are in the same strongly connected component. By the execution algorithm and by 2, γ will be executed before δ.

3.1.2 *Subcase*: δ is not in γ's dependency graph.
Then, by the execution algorithm, γ will be executed (at a moment when δ won't have been executed).

3.2 *Case*: R tries to execute δ before it tries to execute γ.

3.2.1 *Subcase*: δ is in γ's dependency graph.
Then, by 1, δ and γ are in the same strongly connected component. By the execution algorithm and by 2, γ will be executed before δ.

3.2.2 *Subcase*: δ is not in γ's dependency graph.
Then, by 1, γ is in a different strongly connected component than δ, and γ's component is first in reversed topological order. By the execution algorithm, γ is executed before δ.

3.3 *Q.E.D.*
The above cases are exhaustive. In all cases γ is always executed before δ.

□

Finally, **liveness** is guaranteed with high probability as long as a majority of replicas are non-faulty: clients and replicas use time-outs to resend messages, and a client keeps retrying a command until a replica succeeds in committing that command.

3.6 Reducing the Fast-Path Quorum Size

We have described the core concepts of our protocol in the previous section. We now describe modifications that allow EPaxos to use a smaller fast-path quorum—only $F + \left\lfloor \frac{F+1}{2} \right\rfloor$ replicas, including the command leader. We call this version *Optimized EPaxos* and present a TLA+ specification for it in the Appendix.

This is an important optimization because, by decreasing the number of replicas that must be contacted, EPaxos has lower latency (especially in the wide area) and higher throughput, because replicas process fewer messages for each command. For three and five replicas, this fast path quorum is optimal (two and three replicas respectively).

3.6.1 Intuition for the New Fast-Path Quorum Size

The new fast-path quorum size of $F + \left\lfloor \frac{F+1}{2} \right\rfloor$ is necessary for correct recovery after at most F concurrent failures. More precisely, it is necessary for the correct recovery of every command that appears as if it may have been committed on the fast path by a now unresponsive command leader—commands committed on the slow path are easily recoverable just like in classic Paxos.

Let A be a command that may have been committed on the fast path by an unresponsive command leader. This means that some of the responsive replicas have marked it as *PreAccepted* with exactly the same ordering attributes, but none has marked it as *Accepted* or *Committed*. Therefore, A was either committed on the fast path by its failed command leader, or it has not yet been committed. It is then safe for the recovery procedure to try to commit A with the pre-accepted attributes, because these are the only attributes that A may have already been committed with.

On the other hand, there may be other interfering commands (whose commit status may or may not be known) that have ordering attributes that conflict with those of A. For example, consider a command B that interferes with A, also in the process of being

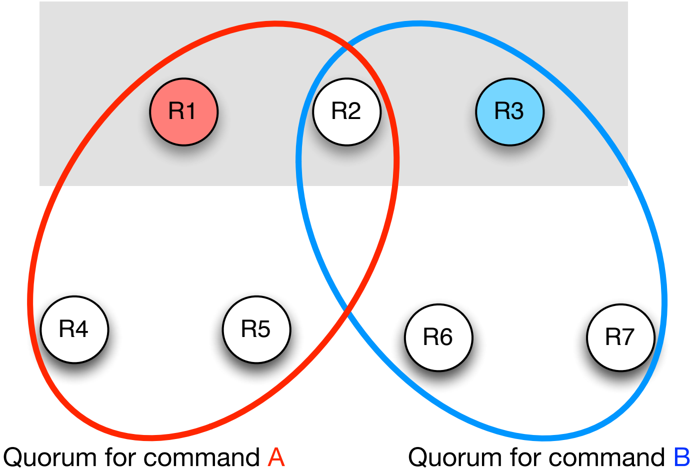


Figure 3.4: A scenario where minimal majorities are not enough for fast-path EPaxos quorums. R1 and R3 are the command leaders for interfering commands A and B respectively, but, along with R2, they fail after sending *PreAccepts*. The recovery procedure cannot decide which one of A or B might have been committed on the fast path. That information is stored on the now failed R2 node, where the two quorums intersect.

recovered, and neither one of these two commands has the other as a dependency. This is clearly wrong and, if A and B are committed with these sets of attributes, it can lead to inconsistencies. The recovery procedure must gather enough information to prove which one of A or B (or both) could not have been committed on the fast path, and change its attributes by essentially restarting the voting process for it. The new size of the fast-path quorum makes this possible.

First, we will show why a simple majority of $F+1$ replicas is not enough. Consider the diagram in Figure 3.4 which depicts a scenario where of the seven replicas, the command leaders for interfering commands A and B both failed just after sending *PreAccepts*. The only replica where the two quorums intersect has also failed, so the recovery procedure cannot decide which one of A or B might have been committed on the fast path (i.e., which one of A or B was received first by R2).

On the other hand, notice how $F + \left\lfloor \frac{F+1}{2} \right\rfloor$ (i.e. 4 out of a total of 7 replicas), is sufficient: in this case, either the quorums intersect in some of the nodes left alive after F failures, or, as exemplified in Figure 3.5, the quorums intersect in the command leaders. In the latter situation, the recovery procedure can safely conclude that none of the two

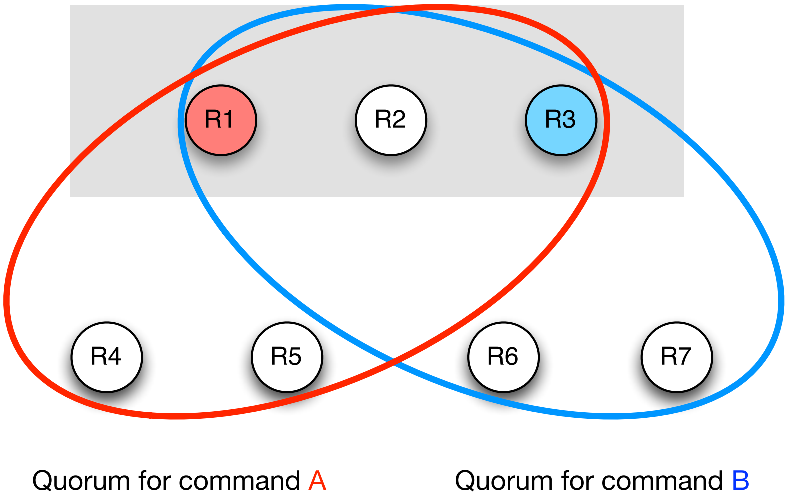


Figure 3.5: $F + \left\lfloor\frac{F+1}{2}\right\rfloor$ replicas in every fast-path quorum is sufficient for the recovery procedure to correctly decide which commands may or may not have been committed on the fast path. In this example, the two command leaders are in each other's quorums, which makes it impossible for either command to have been committed on the fast path. The gray rectangle indicates the failed replicas.

commands could have been committed on the fast path because the opposite command leader would have added an extra dependency on its own command.

3.6.2 Summary of Changes

The recovery procedure (i.e., the Explicit Prepare Phase) changes substantially, starting with line 32 in our pseudocode description. The new command leader Q looks for only $\left\lfloor\frac{F+1}{2}\right\rfloor$ replicas that have pre-accepted a tuple $(\gamma, \mathit{deps}_\gamma, \mathit{seq}_\gamma)$ in the current instance with identical attributes. Upon discovering them, it tries to convince other replicas to pre-accept this tuple by sending *TentativePreAccept* messages. A replica receiving a *TentativePreAccept* will pre-accept the tuple only if it does not conflict with other commands in the replica's log—i.e., an interfering command that is not in deps_γ and does not have γ in its *deps* either, or one that is in deps_γ but has a *seq* attribute at least as large as seq_γ. If the tuple does conflict with such a command, and that command is committed, Q will know γ could not have been committed on the fast path. If a un-committed conflict exists, Q defers recovery until that command is committed. Finally, if Q convinces $F + 1$ replicas (counting the failed command leader and the remainders of the fast-path quorum) to pre-accept $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$, it commits this tuple by running the Paxos-Accept phase for it.

One corner case of recovery is the situation where a dependency has changed its *seq* attribute to a value higher than that of the command being recovered. We can preclude this situation by allowing command leaders to commit command γ on the fast path only if for each command in $\textit{deps}_\gamma$ at least one acceptor has recorded it as committed. For $N \leq 7$, a more efficient solution is to attach updated *deps* attributes to *Accept* and *AcceptReply* messages, and ensure that the recipients of these messages record them. This information will be used only to aid recovery.

The next subsections contain a detailed description of this new recovery procedure, as well as proofs that recovery can always make progress if a majority of replicas are alive—the new size of the fast-path quorum is necessary and sufficient for this to hold—and that optimized EPaxos provides the guarantees enumerated in Section 3.4.

Another important implication of the new fast-path quorums size is that after F failures there may be as few as $\left\lfloor \frac{F+1}{2} \right\rfloor$ surviving members of a fast quorum, which will not constitute a majority among the remaining replicas. Therefore, if the command leader sends *PreAccept* messages to every replica (instead of sending *PreAccepts* to only the replicas in a fast quorum), the recovery procedure may not be able to correctly identify which replicas' replies the failed command leader took into consideration if it committed the instance. Still, such redundancy is sometimes desirable because the command leader may not know in advance which replicas are still live or which replicas will reply faster. When this is the case, we change the fast-path condition as follows: a command leader will commit on the fast path only if it receives $F + \left\lfloor \frac{F+1}{2} \right\rfloor - 1$ *PreAcceptReplies that match its initial ordering attributes*—and every replica that replies without updating these attributes marks this in its log so the recovery procedure can take only these replicas into consideration.

When *not* sending redundant *PreAccepts* (we call this version *thrifty*), a three-replica system will always be able to commit on the fast path—there can be no disagreement in a set with only one acceptor. This is the variant that we focus on in the next subsections, although the proofs of correctness can be easily adapted for the version that uses redundancy.

3.6.3 Preferred Fast-Path Quorums

Instead of sending *PreAccept* messages to every replica, a command leader sends *PreAccepts* to only those replicas in a fast-path quorum that includes itself. We call this mode of operation *thrifty*. The fast-path quorum can be static per command leader, or it can change for every new command—depending on inter-replica communication latency and dynamic load assessment.

Using this optimization has the immediate benefit of decreasing the overall number of messages processed by the system for each command, thus increasing the system throughput. Another consequence is that for 3 replicas, there is no chance of conflicts, even when all commands interfere. This is because the command leader sends only one *PreAccept*, so the corresponding *PreAcceptReply* has no other reply to conflict with. As long as there are no failures and replicas reply timely, a 3-replica thrifty EPaxos state machine will commit every command after just one round of communication.

Finally, the most important consequence of using the thrifty optimization is that we can decrease the fast-path quorum size from $2F$ to $F + \lfloor \frac{F+1}{2} \rfloor$, where F is the maximum number of failures the system can tolerate (the total number of replicas is therefore $N = 2F + 1$).

To achieve this, we modify the fast path condition in Phase 2 (line 10 of the pseudocode in Figure 3.2), and we also modify the recovery procedure (i.e., the Explicit Prepare Phase) as described in the next subsection. The rest of the algorithm remains the same as previously described.

The command leader commits a command on the fast path if both of the following conditions are fulfilled:

1. **FP-quorum**: The command leader receives $F + \lfloor \frac{F+1}{2} \rfloor - 1$ *PreAcceptReplies* with identical *deps* and *seq* attributes, and

2. **(option 1) FP-deps-committed**: For every command in *deps*, at least one of the replicas in the quorum (including the command leader itself) has recorded that command as Committed—acceptors pass this information to the command leader with at most one bit per each command included in *deps*.

   (option 2) Accept-Deps: Every sender of an Accept or AcceptReply message attaches a dependency list (*deps*) updated right before the *Accept* or *AcceptReply* is sent; every receiver of an *Accept* or *AcceptReply* will store the message in its log permanently.

For the second condition, we can use **option 1** to ensure that the *seq* attribute for every command in *deps* is final (it will not change)—this will aid in recovering from failures, as explained in the next subsesction. Alternatively, for up to seven total replicas, we can use the **option 2**; the updated dependency information in the recorded *Accept* and *AcceptReply* messages is only used during the recovery procedure, and has no role in the execution algorithm. Although less straightforward, using the second option has the important benefit that it increases the chance of committing on the fast path when commands interfere frequently.

When using the Accept-Deps variant of the protocol (option 2), it is important for replicas that have not received a command to not set dependencies on its corresponding instance. This may happen in implementations that use the optimization described in Section 3.5.4 (which involves setting implicit dependencies on all instances with IDs smaller than a given instance ID) and may result in the approximate sequence number not being updated correctly. There are two ways an implementation can avoid this problem: (a) acceptors attach the ranges of instances that they have not seen to *PreAcceptReplies* or (b) acceptors do not reply to *PreAccepts* before receiving messages for all instances that they would set implicit dependencies on.

For 3 and 5 replicas, the new fast-path quorum sizes become 2 and 3, respectively, which is optimal (just like for classic Paxos).

3.6.4 Failure Recovery in Optimized Egalitarian Paxos

We now describe in detail the new recovery procedure (i.e., the new Explicit Prepare Phase) that allows us to use smaller fast-path quorums.

The recovery procedure guarantees that a command committed on the fast path will be committed even if its command leader and $F - 1$ other replicas have since failed.

Let R be a replica trying to decide instance $Q.i$ of a potentially failed replica Q:

1. R sends *Prepare* messages to all other replicas, with a higher ballot number than the initial ballot number for $Q.i$.

   Each replica replies with the information recorded for $Q.i$, if any. R waits for at least $F + 1$ replies (including itself). If R does not receive $F + 1$ ACKS (because some replicas have received messages with higher ballots, and reply with NACKS), R increases the ballot number and retries.

2. **If** no replica has any information about $Q.i$, R exits recovery and starts the process of choosing a *no-op* at $Q.i$ by proposing it in the Paxos-Accept Phase.

3. **If** at least one replica has committed command γ in $Q.i$ (there is at most one such command), with attributes *deps*$_\gamma$ and *seq*$_\gamma$, R commits γ locally, sends *Commit*$(Q.i,$ γ, *deps*$_\gamma$, *seq*$_\gamma$) to every other replica, and exits recovery.

4. **If** at least one replica has accepted command (γ, *deps*$_\gamma$, *seq*$_\gamma$) in $Q.i$, R exits recovery and starts a Paxos-Accept Phase for this tuple at $Q.i$ (it will choose the one accepted with the highest ballot number, if there are multiple different accepted tuples at $Q.i$).

5. **If** at least $\lfloor \frac{F+1}{2} \rfloor$ replicas have pre-accepted γ *with the same attributes* (γ, *deps*$_\gamma$, *seq*$_\gamma$), **in** $Q.i$**'s default ballot** then **goto** 6.

   Else R exits recovery and starts the process of choosing γ at $Q.i$, on the slow path (i.e., Phase 1, Phase 2, Paxos-Accept, Commit).

6. R sends *TentativePreAccept*$(Q.i, \gamma,$ *deps*$_\gamma$, *seq*$_\gamma$) to all the respondents that have not pre-accepted γ.

   When receiving a *TentativePreAccept*$(Q.i, \gamma,$ *deps*$_\gamma$, *seq*$_\gamma$) a replica pre-accepts (γ, *deps*$_\gamma$, *seq*$_\gamma$) at $Q.i$ if it has not already recorded an interfering command with conflicting attributes—i.e., any command δ such that:

   i. $\gamma \sim \delta$, and

   ii. $\gamma \notin$ *deps*$_\delta$, and

   iii. (a) $\delta \notin$ *deps*$_\gamma$, or

   (b) $\delta \in$ *deps*$_\gamma$, but *seq*$_\delta \geq$ *seq*$_\gamma$ (this subcase has one exception that does not constitute a conflict: δ and γ have the same initial command leader, and δ is recorded as pre-accepted).

   Otherwise, if such a command δ with conflicting attributes exists, the receiver of the *TentativePreAccept* replies with NACK, δ's instance, the identity of the command leader that has sent δ, and the status of δ (pre-accepted, accepted or committed).

7. (a) **If** the total number of replicas that have pre-accepted or tentatively pre-accepted (γ, *deps*$_\gamma$, *seq*$_\gamma$) is at least $F + 1$ (and we can count Q here too, even if it does not reply), R exits recovery and starts a Paxos-Accept Phase for this tuple at Q.i.

   (b) **Else if** a *TentativePreAccept* NACK returns a status of committed, R exits recovery and starts the process of choosing γ at $Q.i$, on the slow path.

(c) **Else if** a *TentativePreAccept* NACK returns an instance not in γ's dependency list, with a command leader that must have been part of γ's fast quorum for γ to have been committed on the fast path, then R exits recovery and starts the process of choosing γ at $Q.i$, on the slow path.

(d) **Else if** There exists command γ_0 such that R has deferred the recovery of γ_0 because of a conflict with γ, and γ_0's initial command leader must have been part of γ's fast quorum for γ to have been committed on the fast path, then R exits the recovery of γ and starts the process of choosing γ at $Q.i$, on the slow path.

(e) **Else** R defers γ's recovery, and tries to decide one of the uncommitted commands that conflicts with γ.

If $F \leq 3$ and we implement the **Accept-Deps** protocol modification, then step 7 becomes:

7'. (a) **If** the total number of replicas that have pre-accepted or tentatively pre-accepted $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$ is at least $F + 1$ (and we can count Q here too, even if it does not reply), R exits recovery and starts a Paxos-Accept Phase for this tuple at Q.i.

(b) **Else if** a *TentativePreAccept* NACK returns a status of committed for a command δ such that $\delta \in \textit{deps}_\gamma$ and $\textit{seq}_\delta \geq \textit{seq}_\gamma$, R checks the additional *Accept* and *AcceptReply* dependencies recorded at F other replicas. If there exists an *Accept* or *AcceptReply* for the tuple with which δ has been committed, such that γ is not part of its additional dependencies, and the sender of that message is part of γ's fast quorum (i.e., must be part of the quorum for the fast-path hypothesis to hold), then R exits the recovery procedure and starts the process of choosing γ at $Q.i$ on the slow path. If, on the other hand, none of the F replicas has recorded such an *Accept* or *AcceptReply* message, then R resends the *TentativePreAccept* specifying that δ is no longer a conflict for $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$.

(c) **Else if** a *TentativePreAccept* NACK returns a status of committed, R exits recovery and starts the process of choosing γ at $Q.i$, on the slow path.

(d) **Else if** a *TentativePreAccept* NACK returns an instance not in γ's dependency list, with a command leader that must have been part of γ's fast quorum for γ to have been committed on the fast path, then R exits recovery and starts the process of choosing γ at $Q.i$, on the slow path.

(e) **Else if** There exists command γ_0 such that R has deferred the recovery of γ_0 because of a conflict with γ, and γ_0's initial command leader must have been part of γ's fast quorum for γ to have been committed on the fast path, then R exits the recovery of γ and starts the process of choosing γ at $Q.i$, on the slow path.

(f) **Else** R defers γ's recovery, and tries to decide one of the uncommitted commands that conflicts with γ.

This decision process is depicted in Figure 3.6.

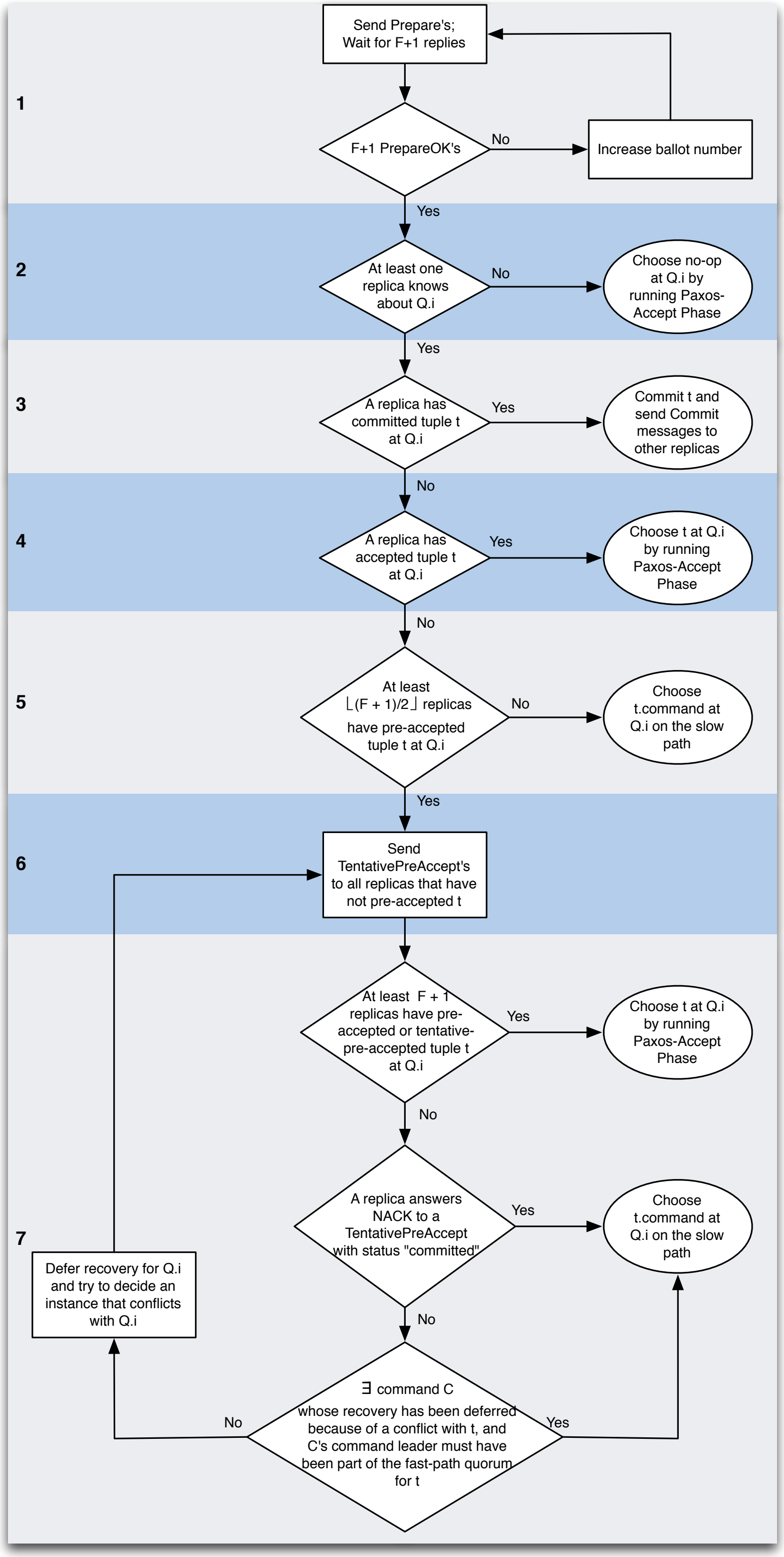

1
Send Prepare's; Wait for F+1 replies
F+1 PrepareOK's
No
Increase ballot number
Yes
2
At least one replica knows about Q.i
No
Choose no-op at Q.i by running Paxos-Accept Phase
Yes
3
A replica has committed tuple t at Q.i
Yes
Commit t and send Commit messages to other replicas
No
4
A replica has accepted tuple t at Q.i
Yes
Choose t at Q.i by running Paxos-Accept Phase
No
5
At least ⌊(F + 1)/2⌋ replicas have pre-accepted tuple t at Q.i
No
Choose t.command at Q.i on the slow path
Yes
6
Send TentativePreAccept's to all replicas that have not pre-accepted t
At least F + 1 replicas have pre-accepted or tentative-pre-accepted tuple t at Q.i
Yes
Choose t at Q.i by running Paxos-Accept Phase
No
7
A replica answers NACK to a TentativePreAccept with status "committed"
Yes
Choose t.command at Q.i on the slow path
No
Defer recovery for Q.i and try to decide an instance that conflicts with Q.i
∃ command C whose recovery has been deferred because of a conflict with t, and C's command leader must have been part of the fast-path quorum for t
No
Yes

Figure 3.6: Decision process for recovery in optimized EPaxos.

3.6.5 Formal Proofs of Properties for Optimized Egalitarian Paxos

We are now ready to explain why the fast-path quorum must be $F + \lfloor \frac{F+1}{2} \rfloor$: so that the following lemma holds:

Lemma 2. *The recovery procedure for Thrifty Egalitarian Paxos can always make progress (as long as the system is live).*

Proof:

The recovery procedure blocks only if there exist commands $c_1, c_2, ..., c_n$ such that the recovery for c_i defers to c_{i+1} for any $i = 1..n-1$, and c_n defers to c_1. The recovery procedure must assume about every one of these commands that it might have been committed on the fast path.

I **Case 1:** The chain contains two consecutive commands γ and δ such that $\gamma \notin \mathit{deps}_\delta$ and $\delta \notin \mathit{deps}_\gamma$.

Let R be a replica trying to recover γ. R must believe that γ may have been committed on the fast path. Eventually, R will defer γ and try to decide δ, and, by our initial assumption, it must believe that δ too may have been committed on the fast path.

R must be aware of the following sets and their properties:

1. RESP_γ, the set of all the replicas in γ's fast quorum (QUOR_γ) that have responded to R's prepare messages, does not include L_γ, the initial command leader for γ (otherwise γ could be decided);
2. RESP_δ, the set of all the replicas in δ's quorum (QUOR_δ) that have responded to R's prepare messages, does not include L_δ, the initial command leader for δ;
3. $|\mathit{RESP}_\gamma| \geq \lfloor \frac{F+1}{2} \rfloor$;
4. $|\mathit{RESP}_\delta| \geq \lfloor \frac{F+1}{2} \rfloor$;
5. $\mathit{RESP}_\gamma \cap \mathit{RESP}_\delta = \varnothing$ (because a replica cannot pre-accept both commands with conflicting attributes);
6. R infers that $L_\gamma \notin \mathit{QUOR}_\delta$—otherwise δ could not have been committed on the fast path, since L_γ has set conflicting attributes for γ.
7. R infers that $L_\delta \notin \mathit{QUOR}_\gamma$—otherwise γ could not have been committed on the fast path, since L_δ has set conflicting attributes for δ.

8. Since there are at most F replicas that do not reply to R, and L_γ (the possibly failed command leader for γ) must be one of them (otherwise R could decide γ), by 6, there are at most $F-1$ replicas that may be part of QUOR_δ (we denote this superset by $\overline{\mathit{QUOR}_\delta}$) and that R does not receive replies from. Then, for R to believe δ may have been committed on the fast path, it must be the case that $|\mathit{RESP}_\delta| \geq \lfloor \frac{F+1}{2} \rfloor + 1$

By 2, 5 and 7, R must infer that the following sets are disjoint: $\overline{\mathit{QUOR}_\gamma}$ (i.e., the set of replicas that may be part of QUOR_γ), $RESP_\delta$, and $\{L_\delta\}$. By 8 and our fast-path quorum requirement, the cardinality of the union of these sets must be at least $F + \lfloor \frac{F+1}{2} \rfloor + \lfloor \frac{F+1}{2} \rfloor + 1 + 1 > 2F + 1$. But this is impossible, because this union must be a subset of the replica set, and its cardinality is $2F+1$. Therefore, some of these sets overlap, so R cannot be simultaneously uncertain about γ and δ. Our assumption that the recovery procedure could deadlock is false.

II **Case 2**: $c_{i+1} \in \mathit{deps}_{c_i}$, for all $i = 1..n$ (with $c_{n+1} \equiv c_1$).

For recovery to defer, it must be the case that $c_i \sim c_{i+1}$, $c_i \notin \mathit{deps}_{c_{i+1}}$, $c_{i+1} \in \mathit{deps}_{c_i}$, and $\mathit{seq}_{c_i} \geq \mathit{seq}_{c_{i+1}}$ for any i (and $c_{n+1} \equiv c_1$). Then $\mathit{seq}_{c_1} = \mathit{seq}_{c_2} = ... = \mathit{seq}_{c_n}$. Note also that this is only possible for $n \geq 3$.

1. **Subcase**: There exist c_i and c_{i+1} that have the same initial command leader.

   Since c_{i+1} has not been decided, it must be the case that all the information available about c_{i+1} is that it has been pre-accepted by various replicas. By our definition of conflicts (step 6 of the recovery procedure, point iii.(b)), c_{i+1} will not conflict with c_i, which contradicts our assumption for this subcase.

2. **Subcase**: No two consecutive commands in the chain have the same command leader.

   As noted earlier, it must be the case that $n \geq 3$, otherwise there is no conflict. Consider commands c_1, c_2 and c_3. It is impossible that any replica that replies to *Prepare* messages has pre-accepted two consecutive commands in the chain (because then they would not have been pre-accepted with the same sequence numbers). Furthermore, at least $\left\lfloor \frac{F+1}{2} \right\rfloor$ responding replicas have pre-accepted each command in the chain (for c_i we denote this set of replicas as $RESP_{c_i}$). We show that the recovery procedure concludes that either c_1's leader must have been part of c_2's fast quorum or that c_2's leader must have been part of c_3's fast quorum:

   2.1 **Sub-subcase**: F is odd.

Then $2F + 1 - \left\lfloor \frac{F+1}{2} \right\rfloor = F + \left\lfloor \frac{F+1}{2} \right\rfloor$, and since c_2 has not been pre-accepted by the $\left\lfloor \frac{F+1}{2} \right\rfloor$ replicas in $RESP_{c_1}$, all the other replicas, including c_1's leader, must have been part of c_2's fast quorum.

2.2 **Sub-subcase**: F is even.
Let $LIVE$ be a set of $F + 1$ replicas that respond to *Prepare* messages (more replica may reply, we only consider $F+1$ of them). No more than $\left\lfloor \frac{F+1}{2} \right\rfloor + 1$ of the replicas in $LIVE$ can be part of any one command's fast quorum, because at least $\left\lfloor \frac{F+1}{2} \right\rfloor$ will be part of the fast quorum for the subsequent command in the chain.
If $|LIVE \cap RESP_{c_2}| = \left\lfloor \frac{F+1}{2} \right\rfloor$, then all replicas outside $LIVE$ must have been part of c_2's fast quorum, including c_1's leader.
If $|LIVE \cap RESP_{c_2}| = 1 + \left\lfloor \frac{F+1}{2} \right\rfloor$, then $|LIVE \cap RESP_{c_3}| = \left\lfloor \frac{F+1}{2} \right\rfloor$, so all replicas outside $LIVE$ must have been part of c_3's fast quorum, including c_2's leader.

The sub-subcases enumerated above are exhaustive. In all situations, the leader of a command c_i must have been part of the fast quorum for c_{i+1}. It is therefore impossible for c_{i+1} to have been committed on the fast-path, since c_i's leader couldn't have pre-accepted c_{i+1} with the same sequence number as c_i without adding c_i to c_{i+1}'s dependency list. As per 7 (or 7') in the recovery procedure, the recovery procedure will eventually abandon the fast-path recovery for c_{i+1}.

The subcases enumerated above are exhaustive.

□

Finally, we show that the recovery procedure is correct. We start by showing that it commits only safe tuples:

Theorem 6. *The Optimized Egalitarian Paxos recovery procedure commits only safe tuples.*

Proof:

Assume the recovery procedure is trying to recover instance $Q.i$. We show that the tuple that it commits at $Q.i$ is safe.

1 *Case:* No tuple is committed at instance $Q.i$ before the recovery procedure commits a tuple at $Q.i$.

In all cases, the recovery procedure ends by choosing a tuple on the slow path, by running classic Paxos. The tuple is thus safe by the classic Paxos guarantees.

2 *Case:* A tuple $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$ has been committed at $Q.i$ before the recovery procedure terminates.

2.1 *Subcase:* $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$ has previously been committed on the slow path.
Then there must be at least $F + 1$ replicas that have accepted $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$. Since the recovery procedure terminates by running classic Paxos in all cases, it will use the same tuple in a Paxos-Accept Phase. By the guarantees of the classic Paxos algorithm, only this tuple can ever be committed at $Q.i$.

2.2 *Subcase:* $(\gamma, \textit{deps}_\gamma, \textit{seq}_\gamma)$ has previously been committed on the fast path.
Then there must be $F + \lfloor \frac{F+1}{2} \rfloor$ replicas that have pre-accepted this tuple at $Q.i$ before processing the *Prepare*s of the recovery procedure (otherwise the initial command leader would have received NACKs for the initial *PreAccept*s and not taken the fast path). Since at most F replicas can be faulty, the recovery procedure will take into account the *PrepareReplies* of at least $\lfloor \frac{F+1}{2} \rfloor$ of them, and by step 5 of the recovery procedure, it will try to obtain a quorum for this tuple. We show that it will succeed:

2.2.1 No interfering command $\delta \sim \gamma$, can be committed such that $\delta \notin \textit{deps}_\gamma$ and $\gamma \notin \textit{deps}_\delta$.
Proof: δ must be pre-accepted by a majority of replicas, and that majority will intersect γ's quorum (itself a majority) in at least one replica, which will ensure that at least one command will be in the other's *deps* set.

2.2.2 If the protocol does not implement **Accept-Deps**, but does implement **FP-deps-committed**, then no interfering command $\delta \sim \gamma$, $\delta \in \textit{deps}_\gamma$, can be committed such that $\gamma \notin \textit{deps}_\delta$ and $\textit{seq}_\delta \geq \textit{seq}_\gamma$.
Proof:
We prove this by generalized induction. The relation that we run the induction on is $a \prec b \equiv$ "command a has been committed (in a particular instance) by the recovery procedure for the first time before command b has been committed (in a particular instance) by the recovery procedure for the first time".

2.2.2.1 *Base case:* Let γ_0 be the first command initially committed on the fast path and then committed again as a result of the recovery procedure (or one of the first, if multiple such commands are committed at the exact same time).
Assume there existed $\delta \sim \gamma_0$, $\delta \in \textit{deps}_{\gamma_0}$, committed such that $\gamma_0 \notin \textit{deps}_\delta$ and $\textit{seq}_\delta \geq \textit{seq}_{\gamma_0}$ at the time of γ_0's recovery. Since γ_0 had

been committed on the fast path, then by the *additional condition for the fast-path in optimized EPaxos* (**FP-deps-committed**), all its dependencies, including δ must have been committed before seq_{γ_0} had been computed. Then, δ must have been committed again in the meantime with different attributes (thus breaking safety). But by Lemma 1, 1, 2.1, 2.2.1, and the recovery procedure, this could only have occurred if δ had been committed incorrectly by the recovery procedure (before γ_0), after initially having been committed on the fast-path—all other commit paths preserve safety. By our base case assumption, this is impossible, since $\gamma_0 \prec \delta$.

2.2.2.2 *Induction step:* The property holds for γ if it holds for every $\delta \prec \gamma$. Assume there exists $\delta \sim \gamma$, $\delta \in deps_\gamma$, committed such that $\gamma \notin deps_\delta$ and $seq_\delta \geq seq_\gamma$. Since γ has been committed on the fast path, then, by the additional condition for the fast-path in optimized EPaxos, all its dependencies, including δ must have been committed before seq_γ had been computed. Then, δ must have been committed again with different attributes (thus breaking safety). But by Lemma 1, 1, 2.1, 2.2.1 and the recovery procedure, this could only occur if δ has been committed incorrectly by the recovery procedure after initially having been committed on the fast-path—we have shown that all other commit paths preserve safety. Since γ has not been committed by the recovery procedure yet, $\delta \prec \gamma$. By the induction hypothesis and by 2.2.1, the recovery procedure would have exited δ's recovery by correctly committing its initial fast-path attributes. Then seq_δ cannot be larger or equal to seq_γ, since seq_γ has been updated to be larger than seq_δ at δ's initial commit time.

2.2.2.3 *Q.E.D*
The induction is complete.

2.2.3 If the protocol does not implement **FP-deps-committed**, but does implement **Accept-Deps**, the recovery procedure will not exit on the first Else case of step 7'.
PROOF: For the recovery procedure to exit on the first Else case of step 7', there must exist a committed tuple $(\delta, deps_\delta, seq_\delta)$, with $\delta \sim \gamma$, $\delta \in deps_\gamma$, $\gamma \notin deps_\delta$ and $seq_\delta \geq seq_\gamma$, and there must exist an *Accept* or *AcceptReply* message for this tuple sent by a replica R' in $\gamma's$ fast-path quorum such that the additional dependencies for this message do not include γ. Then R' must have accepted $(\delta, deps_\delta, seq_\delta)$ before pre-accepting γ. This is im-

possible, since then R' would not have pre-accepted γ with $\mathit{seq}_\gamma \leq \mathit{seq}_\delta$, as we know it must have for γ to be committed on the fast-path.

2.2.4 The recovery procedure will not exit on branches 7.c or 7d (7'.d or 7'.e, respectively): No replica in γ's fast quorum can start instances for commands that interfere with γ and set conflicting attributes (as per the definition of conflicting attributes in step 6 of the recovery procedure), because all these replicas have pre-accepted γ with its attributes.

2.2.5 *Q.E.D*

By the recovery procedure, 2.2.1, 2.2.2, 2.2.3, 2.2.4 and Lemma 2 the recovery procedure will be successful in getting F replicas to pre-accept tuple $(\gamma, \mathit{deps}_\gamma, \mathit{seq}_\gamma)$ (not counting the implicit pre-accept of the initial command leader), and it will start the Paxos-Accept Phase for this tuple.

2.3 *Q.E.D*

Subcases 2.1 and 2.2 are exhaustive and safety is preserved in both.

3 *Q.E.D.*

Cases 1 and 2 are exhaustive and safety is preserved in both.

□

Next, we show that the recovery procedure preserves execution consistency:

Theorem 7. *The Optimized Egalitarian Paxos preserves execution consistency.*

Proof:

Let γ and δ be two commands that interfere and have been committed. We show that all replicas execute γ and δ in the same order.

1 *Case:* Both γ and δ have first been committed by their respective command leaders, without running the recovery procedure.

This is no different from simplified EPaxos: the different fast-path condition influences only the recovery path. By Theorem 6 and Theorem 4, γ and δ will be executed in the same order by every replica.

2 *Case:* γ is first committed as a result of the recovery procedure, while δ is first committed by its initial command leader without running the recovery procedure.

2.1 *Subcase:* γ is committed before step 7 of the recovery procedure, or after exiting one of the Else branches in step 7.

Then γ must have been pre-accepted by a majority of replicas and then committed after running the Paxos-Accept Phase. This too is reducible to the simple EPaxos case, so, by Theorem 4, γ and δ will be executed in a consistent order across all non-faulty replicas.

2.2 *Subcase:* γ is committed after exiting the recovery procedure on the If branch in step 7.

We show that either γ has δ as a dependency or δ has γ as a dependency:

2.2.1 *Sub-subcase:* γ had been pre-accepted with $\delta \in \textit{deps}_\gamma$.
γ's pre-accepted attributes as received in the recovery procedure at step 7 do not change, so γ will be committed with δ as a dependency.

2.2.2 *Sub-subcase:* γ had been pre-accepted with $\delta \notin \textit{deps}_\gamma$.
Since the recovery procedure exits on the If branch of step 7, at least $F+1$ replicas, including γ's original command leader have pre-accepted γ as a result of a *PreAccept* or a *TentativePreAccept*. δ will also have been pre-accepted by a majority of replicas, so there is at least one replica that has pre-accepted both δ and γ, and whose replies are taken into account both when establishing δ's commit attributes and in the recovery procedure for γ. Let this replica be R:

2.2.2.1 *Sub-sub-subcase:* R pre-accepts γ as a result of receiving a *PreAccept* from γ's initial command leader.
Then R must have learned about γ before receiving a *PreAccept* for δ, so $\gamma \in \textit{deps}_\delta$.

2.2.2.2 *Sub-sub-subcase:* R pre-accepts γ after receiving a *TentativePreAccept* during the recovery procedure.
Then, according to the conditions in step 6 of the recovery procedure, either R had already pre-accepted δ such that $\gamma \in \textit{deps}_\delta$, or δ reaches R after the *TentativePreAccept* for γ. In either case, $\gamma \in \textit{deps}_\delta$ when δ commits.

In conclusion $\gamma \in \textit{deps}_\delta$

2.2.3 *Q.E.D.*
Sub-subcases 2.2.1 and 2.2.3 are exhaustive.

By step 2 of the proof for Theorem 4, since at least one command is committed with the other in its dependency list, every replica will execute the commands in the same order.

3 *Case:* δ is first committed as a result of the recovery procedure, while γ is first committed by its initial command leader without running the recovery procedure.

Just like case 2, with γ and δ interchanged.

4 *Case:* Both γ and δ are first committed after the recovery procedure.

If at least one of the commands is committed before step 7 in the recovery procedure, or after exiting step 7 on one of the Else branches, the situation is reducible to one of the previous cases.

The only remaining subcase is that when both commands are committed after exiting step 7 on the If branch.

Assume no command has the other in its dependency list when exiting step 7 of the recovery procedure. But each command has been pre-accepted by a majority or replicas (either as a result of *PreAccepts* or *TentativePreAccepts*). Then there must be at least one replica R that pre-accepts both commands, and whose replies are taken into account when establishing each command's commit attributes. If R pre-accepts γ before δ, then, by the conditions in step 6 of the recovery procedure, R will not acknowledge δ without a dependency for γ (and vice-versa). This contradicts our assumption.

Then at least one command is in the other's dependency list, and by step 2 in the proof for Theorem 4, the commands will be executed in the same order on every replica.

5 *Q.E.D*

Cases 1, 2, 3 and 4 are exhaustive.

□

Finally, we show that the recovery procedure preserves execution linearizability:

Theorem 8. *The Optimized Egalitarian Paxos preserves execution linearizability.*

Proof: Let γ and δ be two interfering commands serialized by clients: δ is proposed only after a replica has committed γ. We show that γ will always be executed before δ

By the time δ is proposed, a majority of replicas have either pre-accepted or accepted γ with its final (commit) attributes. At least one of these replicas will pre-accept δ as a result of receiving a *PreAccept* or a *TentativePreAccept*, and its reply will be considered in deciding δ's final attributes. Let this replica be R:

1 *Case:* R receives a *PreAccept* for δ.

Then R will put γ in deps_δ and it will increment seq_δ to be larger than seq_γ. Since R's reply is considered when deciding δ's final attributes, δ's dependency list will include γ and its sequence number will be larger than γ's at commit time. By the execution algorithm, γ will always be executed before δ.

2 *Case:* R receives a *TentativePreAccept* for δ at some point other than step 7' in the recovery procedure.

Since R will ACK (otherwise δ would not be committed), and $\delta \notin \mathit{deps}_\gamma$ (because δ was proposed after γ was committed), by the conditions in step 6 of the recovery procedure, it must hold that $\gamma \in \mathit{deps}_\delta$ and $\mathit{seq}_\gamma < \mathit{seq}_\delta$. By the execution algorithm, γ will always be executed before δ.

3 *Case:* R receives a *TentativePreAccept* for δ at step 7' in the recovery procedure.

In this case, there must exist a command γ' such that $\gamma' \sim \delta$, $\gamma' \in \mathit{deps}_\delta$, $\delta \notin \mathit{deps}_{\gamma'}$ and $\mathit{seq}_\delta \leq \mathit{seq}_{\gamma'}$, and R is instructed to ignore the conflict between the attributes of δ and those of γ'. We show that $\gamma \neq \gamma'$.

Assume $\gamma' = \gamma$. Then γ must have first been committed on the slow-path, because otherwise δ couldn't have acquired a stale dependency on γ (i.e., a dependency where seq_δ hasn't been updated to be larger than the committed seq_γ)—this remains true for implicit dependencies, as per the discussion in Section 3.6.3. Then there must exist a replica R' that has participated in the Paxos-Accept phase for γ, and that was also supposed to be part of δ's fast-path quorum. We first note that $L_\delta \neq R'$, where L_δ is the initial command leader for δ, because it wouldn't have set the conflicting sequence number for δ after having accepted the commit-time attributes for γ.

3.1 *Subcase* $F = 2$.

The recovery procedure for δ must not have received a response from L_δ (otherwise it would have exited before step 7'). Then there is at most one more replica that fails to respond during the recovery procedure, and this replica must be R' (otherwise R' would have replied with non-conflicting attributes for δ, and the recovery procedure would have ended).

3.1.1 *Sub-subcase* R' was the leader of the Paxos-Accept phase that first committed γ. Then it would have sent *AcceptReply* messages to two other replicas, and these messages would have included additional dependencies (as per **Accept-deps**) that would not have included δ (because δ had not been proposed at this point). Since at most $F = 2$ replica are faulty, these acceptors must be part of the replicas that reply during the recovery for δ, so the recovery procedure will discover the recorded *AcceptReplies* and will not send the *TentativePreAccept* in step 7'.

3.1.2 *Sub-subcase* R' was an acceptor in the Paxos-Accept phase that first committed γ. Then the leader for this Paxos-Accept phase must have responded during the recovery for δ with the *Accept* message from R'. The additional dependencies for this message would not contain δ, so the recovery procedure will not send the *TentativePreAccept* in step 7'.

3.2 *Subcase* $F = 3$.

In this case, the recovery procedure for δ reaches the Else branches of step 7' only if exactly two replicas part of δ's fast-path quorum reply to *Prepare* messages (otherwise the recovery procedure either exits before step 7', or on the If branch of step 7'). Then of the three un-responsive replicas, one is L_δ, and the other two must be part of both δ's fast-path quorum and the Paxos-Accept quorum that has first committed γ. Then at least one of the two other replicas part of γ's Paxos-Accept quorum must have received and recorded an *Accept* or *AcceptReply* message with additional dependencies that did not include δ. Since these replicas are both responsive during the recovery for δ, the recovery procedure could not have sent the *TentativePreAccept* in step 7'.

3.3 *Q.E.D.* Step 7' can be executed only if $F = 2$ or $F = 3$. Both subcases have ended in contradiction, so our assumption that $\gamma = \gamma'$ is false. Then R cannot receive a *TentativePreAccept* that forces it to pre-accept δ such that $seq_\delta \leq seq_\gamma$. By 3.1, 3.2 and the execution algorithm, γ will always be executed before δ.

3 *Q.E.D*

Cases 1 and 2 and 3 are exhaustive.

□

3.7 Minimizing the Fast-Paxos Quorum Size

In the previous section we found that the size of the fast-path quorum must be $F + \lfloor \frac{F+1}{2} \rfloor$ for the recovery procedure to be able to identify those commands that may have been

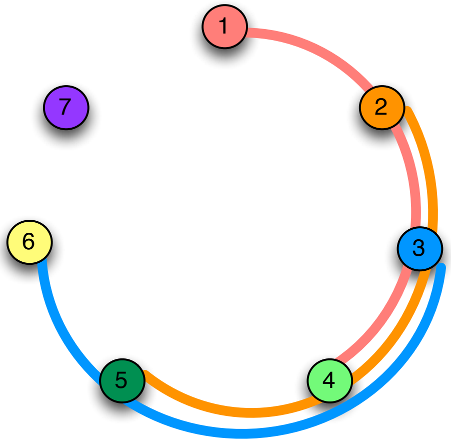


Figure 3.7: An example of how fixed fast-path quorums can be pre-configured so that for any two quorums, the command leader of one is part of the other. The diagram depicts only the quorums for replicas 1, 2 and 3. In this example with seven total replicas, fast-path quorums are minimal (four replicas each).

committed on the fast-path. This size guaranteed that any two quorums intersected in sufficient replicas for recovery to decide which one of two interfering commands could have been committed. This condition was necessary because we allowed command leaders the flexibility to choose any sufficiently large subset of replicas as a quorum.

Another option is to pre-configure the fast-path quorums in a way that guarantees they intersect in at least one of the two corresponding command leaders. This way, for two interfering commands, the recovery procedure will be able to immediately discount the one whose quorum includes the other's command leader—this command could not have committed on the fast path because one of the replicas in its fast path quorum was able to propose an interfering command with conflicting attributes. Figure 3.7 shows a simple configuration where this condition is met: replicas are arranged on a logical circle and the fast-path quorum for each replica contains itself and the F following clock-wise peers. Slow-path quorums remain as flexible as before.

The major advantage of this option is that fast-path quorums are now minimal for any total number of replicas: $F + 1$. This benefits throughput and may benefit wide-area latency for suitable geographical configurations. The disadvantage is that failures are more difficult to tolerate: a failed replica will force all command leaders who have it in their quorums to commit all commands on the slow path until fast-path quorums are reset globally.

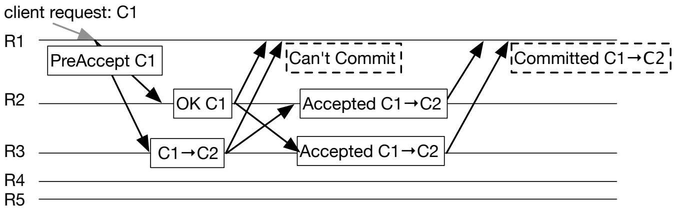


Figure 3.8: Even when a command (C1) must be committed on the slow path, it will incur only three message delays in failure-free runs if we forward PreAccept replies to all members of a quorum.

3.8 Reducing the Length of the Slow Path in the Wide-Area to Three Message Delays

The length of the slow path in Egalitarian Paxos is four message delays (two round trips), plus the client-to-closest-replica message delay that is insignificant in the wide area. Here, we show how to reduce this length to only three wide-area message delays. We call this version *EPaxos 3-WAN-Delays*.

We summarize the modifications to (optimized) EPaxos:

1. After receiving a *PreAccept*, an acceptor will send the corresponding *PreAcceptReply* not only to the command leader, *but also to all the other members of a* **pre-established** *slow quorum* (it is the command leader who decides what the slow quorum is, and describes it in the *PreAccepts* it sends).

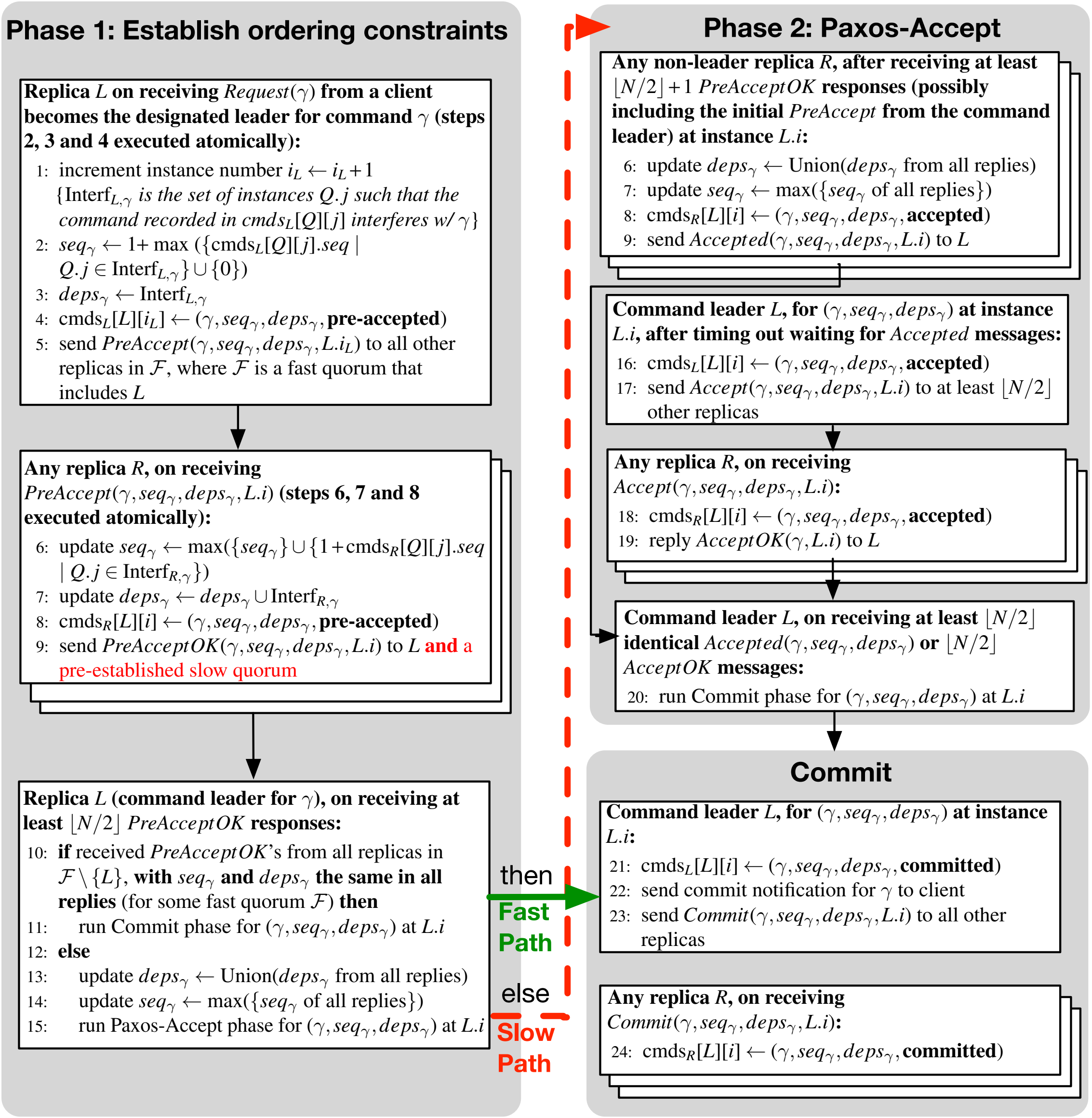
2. After receiving F *PreAcceptReplies* (possibly including from itself), in addition to the initial *PreAccept* from the command leader, an acceptor will compute the updated ordering attributes and accept them. The acceptor will then send an *Accepted* message to the command leader, with the accepted attributes.

3. If the command leader cannot commit on the fast path, it waits for F identical *AcceptReplies*. If it receives them, then it can commit directly. Otherwise, it starts the second phase, sending *Accepts* to a slow-path quorum.

Figure 3.8 depicts an example of how this version of EPaxos commits commands on the slow path.

The pseudocode corresponding to the modified algorithm is presented in Figure 3.9.

This optimization preserves safety: a tuple will be committed as a result of this optimization only if a majority of replicas have first accepted it, and there is only one such



Replica L on receiving $Request(\gamma)$ from a client becomes the designated leader for command γ (steps 2, 3 and 4 executed atomically):

```
1:  increment instance number $i_L \leftarrow i_L + 1$
    {$\text{Interf}_{L,\gamma}$ is the set of instances $Q.j$ such that the command recorded in $cmds_L[Q][j]$ interferes w/ $\gamma$}
2:  $seq_\gamma \leftarrow 1 + \max(\{\text{cmds}_L[Q][j].seq \mid Q.j \in \text{Interf}_{L,\gamma}\} \cup \{0\})$
3:  $deps_\gamma \leftarrow \text{Interf}_{L,\gamma}$
4:  $\text{cmds}_L[L][i_L] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{pre-accepted})$
5:  send $PreAccept(\gamma, seq_\gamma, deps_\gamma, L.i_L)$ to all other replicas in $\mathcal{F}$, where $\mathcal{F}$ is a fast quorum that includes $L$
```

Any replica R, on receiving $PreAccept(\gamma, seq_\gamma, deps_\gamma, L.i)$ (steps 6, 7 and 8 executed atomically):

```
6:  update $seq_\gamma \leftarrow \max(\{seq_\gamma\} \cup \{1 + \text{cmds}_R[Q][j].seq \mid Q.j \in \text{Interf}_{R,\gamma}\})$
7:  update $deps_\gamma \leftarrow deps_\gamma \cup \text{Interf}_{R,\gamma}$
8:  $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{pre-accepted})$
9:  send $PreAcceptOK(\gamma, seq_\gamma, deps_\gamma, L.i)$ to $L$ and a pre-established slow quorum
```

Replica L (command leader for γ), on receiving at least $\lfloor N/2 \rfloor$ $PreAcceptOK$ responses:

```
10: if received $PreAcceptOK$'s from all replicas in $\mathcal{F} \setminus \{L\}$, with $seq_\gamma$ and $deps_\gamma$ the same in all replies (for some fast quorum $\mathcal{F}$) then
11:     run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
12: else
13:     update $deps_\gamma \leftarrow \text{Union}(deps_\gamma$ from all replies)
14:     update $seq_\gamma \leftarrow \max(\{seq_\gamma$ of all replies$\})$
15:     run Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
```

Any non-leader replica R, after receiving at least $\lfloor N/2 \rfloor + 1$ $PreAcceptOK$ responses (possibly including the initial $PreAccept$ from the command leader) at instance $L.i$:

```
6:  update $deps_\gamma \leftarrow \text{Union}(deps_\gamma$ from all replies)
7:  update $seq_\gamma \leftarrow \max(\{seq_\gamma$ of all replies$\})$
8:  $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{accepted})$
9:  send $Accepted(\gamma, seq_\gamma, deps_\gamma, L.i)$ to $L$
```

Command leader L, for $(\gamma, seq_\gamma, deps_\gamma)$ at instance $L.i$, after timing out waiting for $Accepted$ messages:

```
16: $\text{cmds}_L[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{accepted})$
17: send $Accept(\gamma, seq_\gamma, deps_\gamma, L.i)$ to at least $\lfloor N/2 \rfloor$ other replicas
```

Any replica R, on receiving $Accept(\gamma, seq_\gamma, deps_\gamma, L.i)$:

```
18: $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{accepted})$
19: reply $AcceptOK(\gamma, L.i)$ to $L$
```

Command leader L, on receiving at least $\lfloor N/2 \rfloor$ identical $Accepted(\gamma, seq_\gamma, deps_\gamma)$ or $\lfloor N/2 \rfloor$ $AcceptOK$ messages:

```
20: run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
```

Command leader L, for $(\gamma, seq_\gamma, deps_\gamma)$ at instance $L.i$:

```
21: $\text{cmds}_L[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{committed})$
22: send commit notification for $\gamma$ to client
23: send $Commit(\gamma, seq_\gamma, deps_\gamma, L.i)$ to all other replicas
```

Any replica R, on receiving $Commit(\gamma, seq_\gamma, deps_\gamma, L.i)$:

```
24: $\text{cmds}_R[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, \textbf{committed})$
```

Figure 3.9: The Egalitarian Paxos version that incurs only three wide-area message delays on the slow path.

tuple that can be accepted in the initial ballot—safety is therefore guaranteed due to the classic Paxos guarantees. The optimization also preserves execution consistency and linearizability, because the accepted tuple is computed based on the *PreAcceptReplies* of $\lfloor N/2 \rfloor$ replicas, which together with the command leader form a majority (as we showed earlier in proving the EPaxos properties, the input of a majority in establishing the ordering guarantees is the necessary and sufficient condition for ensuring execution consistency and linearizability).

3.9 Strict Serializability

Egalitarian Paxos guarantees that interfering commands are executed in a linearizable order. If the interference relation is transitive (which is equivalent with a setting where each command refers to the state of a single object) then execution linearizability implies the whole system is linearizable, as defined by Herlihy and Wing: linearizability is a *local* property, meaning that *"a system is linearizable if each individual object is linearizable"* [23].

The corresponding consistency property in a system where each command can update and read multiple objects is strict serializability. This is a setting where the interference relation is not necessarily transitive. EPaxos does not guarantee strict serializability without a simple modification that we describe in this section.

For example, imagine a system with two distinct objects A and B, and two concurrent clients *client1* and *client2*. If each client issues the following commands sequentially (i.e., they wait for the a command to be committed before issuing the next)

client 1: *update A*; *update B*
client 2: *read B*; *read A*

it is impossible for *client2* to see an updated *B* and an unmodified *A*. In this case, all operations are linearizable, because they each refer to a single object. However, if *client2* were to issue the composite command *read A and B* instead, it is possible that it will see an updated *B* and an unmodified *A*:

client 1: *update A*; *update B*
client 2: *read A and B*

This is because the read may be concurrent with both updates, and since the updates do not interfere, the system can choose the following ordering: *update B; read A and B; update A*.

We can modify EPaxos to guarantee strict serializability: clients are sent the commit notification for a command only after all instances in the dependencies graph of the current command have been committed. We call this version *EPaxos-strict.* We show that it guarantees strict serializability.

Theorem 9 (Strict serializability). *Every execution in EPaxos-strict is equivalent to a serial execution where non-concurrent commands are executed in their temporal order.*

Proof:

1 We define $a <_i b$ to be true if commands a and b interfere, and a has been executed by a replica before b (by execution consistency, if $a <_i b$ then *every* replica will execute a before b). It is easy to see that $<_i$ is a partial order relation.

2 Because commands are executed serially, every execution in EPaxos-strict (as well as in EPaxos) defines a total order that is an extension of $<_i$ (by execution consistency).

3 We define $a <_{time} b$ to be true if any client has been notified that a has been committed (along with its entire dependency graph) before b is proposed.

4 We show that $<_i \cup <_{time}$ is a partial order relation. We define $<$ to be $(<_i \cup <_{time})$. Assume $<$ is not a partial order relation. Then there exists a sequence of commands $c_1, ..., c_n$ such that $c_1 < c_2 < ... < c_n < c_1$. Consider the shortest such cycle.

4.1 Because $<_i$ is a partial order relation, there must be at least one pair of consecutive commands c_{j-1} and c_j such that $c_{j-1} <_i c_j$ is not true. Then it must be true that $c_{j-1} <_{time} c_j$. By re-indexing, let these two commands be c_1 and c_2.

4.2 The cycle must have more than two commands: it is impossible for $c_2 <_{time} c_1$ since $c_1 <_{time} c_2$; $c_2 <_i c_1$ implies that c_2 is in c_1's dependency graph which also contradicts $c_1 <_{time} c_2$, since c_1 could not have ended before its entire dependency graph was committed (by EPaxos-strict).

4.3 There is no $j > 2$ such that $c_{j-1} <_i c_j$ does not hold: Assume there is such a j. Then $c_{j-1} <_{time} c_j$. If c_1 ends (i.e., its corresponding client is notified) before c_j starts, then $c_1 <_{time} c_j$, and we have the shorter cycle $c_1, c_j, ..., c_n$. If c_1 ends after c_j starts, then c_2 must start after c_{j-1} ends, so $c_{j-1} <_{time} c_2$, and we have the shorter cycle $c_2, ..., c_{j-1}$.

Client notification	Interference must be transitive	Consistency guarantee
After commit	Yes	**Linearizability**
After commit	No	**Per-object linearizability**
After command and entire dependency graph have been committed	No	**Strict serializability**

Table 3.3: Consistency guarantees in EPaxos.

4.4 By 4.3 $c_2 <_i \dots <_i c_n <_i c_1$. But $c_{j-1} <_i c_j$ implies that c_{j-1} is in c_j's dependency graph. Since this relation is transitive, it must hold that c_2 is in c_1's dependency graph. But, by EPaxos-strict, this implies that c_2 was committed before the commit notification for c_1 was sent to its corresponding client. This contradicts $c_1 <_{time} c_2$.

5 By the definition of *interference*, all serial executions that extend $<_i$ are compatible—i.e., they produce the same state and read results. In particular, any execution that extends the partial order $<_i \cup <_{time}$ will be compatible with any serial execution that extends $<_i$. Then by 2, all executions in Epaxos-strict are compatible with a serial execution that extends $<_i \cup <_{time}$. *Q.E.D*

□

Table 3.3 summarizes the consistency guarantees of EPaxos.

Because the dependency graph of a command is complete (all the dependencies are committed) by the time the client receives a notification for it, EPaxos-strict does not require approximate sequence numbers to guarantee linearizability, and this simplifies the recovery procedure. The drawback of EPaxos-strict is that clients may experience higher latency for committing (but not for executing) conflicting commands. The latency for committing and executing non-interfering commands remains the same

3.10 Reconfiguring the Replica Set

Reconfiguring a replicated state machine is an extensive topic [38, 39, 41]. In EPaxos, ordering ballots by their *epoch* prefix enables a solution that resembles Vertical Paxos [38] [3] with majority read quorums: A new replica, or one that recovers without its memory, must receive a new ID and a new (higher) *epoch* number, e.g., from a configuration service or a human. It then sends *Join* messages to at least $F+1$ live replicas that are not themselves in the process of joining. Upon receiving a *Join*, a live replica updates its membership information and the *epoch* part of each ballot number it uses or expects to receive for new instances. It will thus no longer acknowledge messages for instances initiated in older epochs (instances that it was not already aware of). The live replica will then send the joining replica the list of committed or ongoing instances that the live replica is aware of. The joining replica becomes live (i.e., it proposes commands and participates in voting the proposals of other replicas) only after receiving commits for all instances included in the replies to at least $F+1$ *Join* messages. Production implementations optimize this process using snapshots [13].

This strategy ensures that a joining replica and a replica that has been excluded from the new configuration cannot participate in voting commands at the same time—thus preserving the property that any two quorums overlap.

Once $F+1$ live replicas have switched to the new configuration, a replica that has been excluded from the new configuration will not be able to act as command leader (its messages will not be acknowledged by a majority), nor can it participate in successful ballots. The excluded replica might still initiate instances, but they can be finalized only by live replicas that have committed to the new configuration. This eventually stops when all live replicas have committed to the new configuration, or when the excluded replica receives a *Join* message that makes it aware of its exclusion.

The strategy described in this section preserves the protocol guarantees. An instance is either finalized as if the joining replica has not joined yet (if the id of that instance was contained in one of the $F+1$ replies to *Join* messages), or it is treated like a new command in the new configuration.

[3] Vertical Paxos is a generalization of Paxos designed to handle frequent reconfigurations.

3.11 Empirical Evaluation of Egalitarian Paxos

In this section we show the results of our empirical comparison between Egalitarian Paxos, Multi-Paxos, Mencius and Generalized Paxos. The EPaxos variant that we evaluate is Optimized EPaxos, with the exception of the wide-area experiments, where we evaluate both Optimized EPaxos and EPaxos 3-WAN-Delays (the version that has only three wide-area message delays on the slow path).

The experimental setup consisted of Amazon EC2 large instances for both state machine replicas and clients, running Ubuntu Linux 11.10. An Amazon EC2 large instance consists of two 64-bit virtual cores with 2 EC2 Compute Units each and 7.5 GB of memory. For local-area experiments, the typical RTT inside an EC2 cluster is 0.4 ms.

3.11.1 Implementation

We implemented EPaxos, Multi-Paxos, Mencius, and Generalized Paxos in Go, version 1.0.2.

Language-specific details

Behind our choice of Go was the goal of comparing the performance of the four Paxos variants within a common framework in which the protocols share as much code as possible to reduce implementation-related differences. While subjective, we believe we achieved this, applying roughly equal implementation optimization to each; we are releasing our implementations for others to perform comparisons or further optimization [46].

Go presented two challenges: First, the garbage collection that eased implementation of four complete Paxos variants adds performance variability; and second, its RPC implementation is slow. We solved the latter by implementing our own RPC stub generator. We have not fully mitigated the GC penalty, but EPaxos is more affected than the other protocols because its attribute-containing messages are larger, so our results are fair to the other protocols.

Thrifty Operation

For all protocols except Mencius, we used an optimization that we call *thrifty*. In thrifty, a replica in charge of a command (the command leader in EPaxos, or the stable leader in Multi-Paxos) sends *Accept* and *PreAccept* messages to only a quorum of replicas, including itself, not the full set. This reduces message traffic and improves throughput. The drawback is that if an acceptor fails to reply quickly, there is no quick fall-back to another reply. However, *thrifty* can aggressively send messages to additional acceptors when a reply is not received after a short wait time; doing so does not affect safety and only slightly reduces throughput. Mencius cannot be *thrifty* because the replies to *Accept* messages contain information necessary to commit the current instance (i.e., whether previous instances were skipped or not).[4]

3.11.2 Typical Workloads

We evaluate these protocols using a replicated key-value store where client requests are updates (puts). This is sufficient to capture a wide range of practical workloads: From the point of view of replication protocols, reads and writes are typically handled the same way (reads might be serviced locally in certain situations, as discussed in Section 3.5.7). Nevertheless, writes are the more difficult case because reads do not interfere with other reads. Our tests also capture conflicts, an important workload characteristic—a conflict is a situation when potentially interfering commands reach replicas in different orders. Conflicts affect EPaxos, Generalized Paxos, and, to a lesser extent, Mencius. One example of conflicts are those experienced by a lock service, where conflicts are equivalent to write-write conflicts from multiple clients updating the same key. A read-heavy workload is where concurrent updates rarely target the same key, corresponding to low conflict rates. Importantly, lease renewal traffic—constituting over 90% of the requests handled by Chubby [10]—generates no conflicts, because only one client can renew a particular lease.

From the available evidence, we believe that 0% and 2% command interference rates are the most realistic. For completeness, we also evaluate 25% and 100% command interference (for 25%, $\frac{1}{4}$ of commands target the same key while $\frac{3}{4}$ target different keys). In Chubby, fewer than 1% of all commands (observed in a ten-minute period [10]) could possibly generate conflicts. In Google's advertising back-end, F1, which uses the geo-replicated table store Spanner (which, in turn, uses Paxos) fewer than 0.3% of all operations may generate conflicts, since more than 99.7% of operations are reads [16].

[4] A Mencius replica must receive *Accept* replies from the owners of all instances it has not received messages for. We tried Mencius-thrifty, in which the current leader sends *Accept*s first to the replicas it must hear from, and to others only if quorum has not yet been reached. It did not improve throughput, however: under medium and high load, only rarely are all previous instances "filled" when a command is proposed.

We indicate the percentage of interfering commands as a number following the experiment (e.g., "EPaxos 0%"').

3.11.3 Latency In Wide Area Replication

We validate empirically that EPaxos has optimal median commit latency in the wide area with three replicas (tolerating one failure) and five replicas (tolerating two failures). The replicas are located in Amazon EC2 data centers in California (CA), Virginia (VA) and Ireland (EU), plus Oregon (OR) and Japan (JP) for the five-replica experiment. At each location there are also ten clients co-located with each replica (fifty in total). They generate requests simultaneously, and measure the commit and execute latency for each request. Figure 3.10 shows the median and 99%ile latency for EPaxos, Multi-Paxos, Mencius and Generalized Paxos.

With three replicas, an EPaxos replica can always commit after one round trip to its nearest peer even if that command interferes with other concurrent commands. In contrast, Generalized Paxos's fast quorum size when $N = 3$ is three. Its latency is therefore determined by a round-trip to the *farthest* replica. The high 99%ile latency experienced by Generalized Paxos is caused by checkpoint commits. Furthermore, conflicts cause two additional round trips in Generalized Paxos (for any number of replicas). Thus, in this experiment, EPaxos is not affected by conflicts, but Generalized Paxos experiences median latencies of 341 ms with 100% command interference.

With five replicas, EPaxos avoids the two most distant replicas, while Generalized Paxos avoids only the most distant one. Thus, EPaxos has optimal commit latency for the common case of non-interfering concurrent commands, with both three and five replicas. For five replicas, interfering commands cause one extra round trip to the closest two replicas for EPaxos, but up to two additional round trips for Generalized Paxos.

Mencius performs relatively well with multiple clients at every location and all locations generating commands at the same aggregate rate. Imbalances force Mencius to wait for more replies to *Accept* messages. In the worst case, with active clients at only one location at a time, Mencius experiences latency corresponding to the round trip time to the replica that is farthest away from the client, for any number of replicas.

Multi-Paxos has high latency because the local replica must forward all commands to the stable leader.

The results in Figure 3.10 refer to commit latency. For EPaxos, execution latency differs from commit latency only for high conflict rates because a replica must delay executing a command until it receives commit confirmations for the command's dependencies.

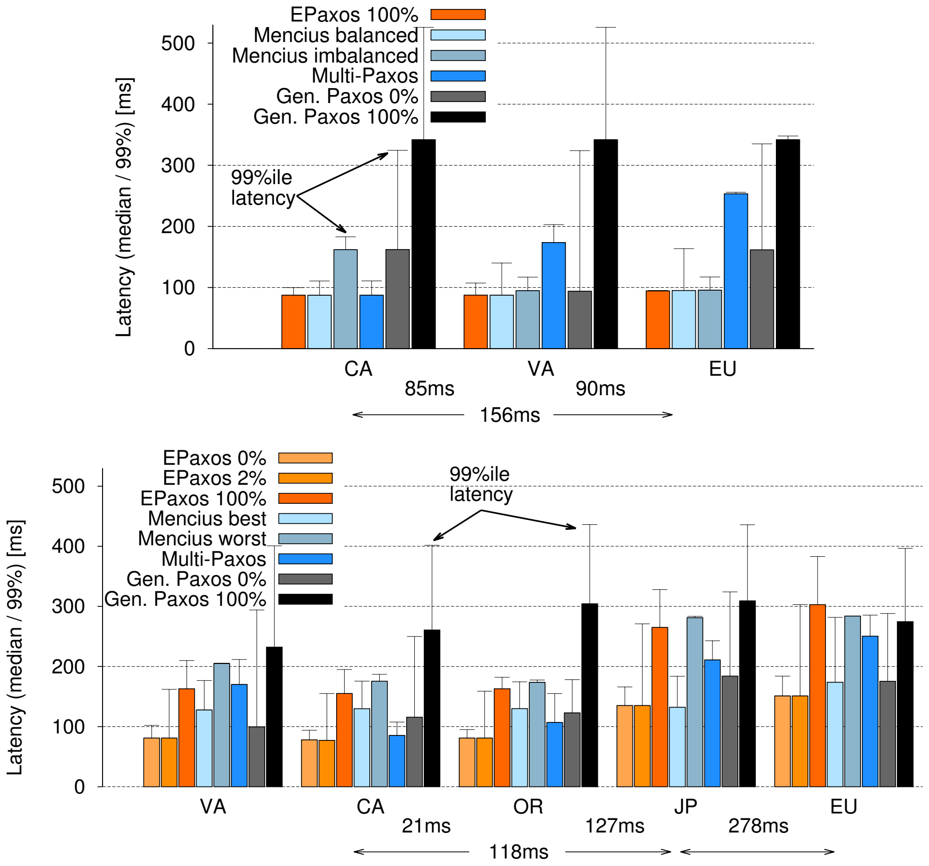


Figure 3.10: Median commit latency (99%ile indicated by lines atop the bars) at each of 3 (top graph) and 5 (bottom graph) wide-area replicas. The Multi- and Generalized Paxos leader is in CA. In *Mencius imbalanced*, EU generates commands at half the rate of the other sites (no other protocol is affected by imbalance). In *Mencius worst*, only one site generates commands at a given time. The bottom of the graph shows inter-site RTTs.

With 100% interference rate (i.e., worst case), three-replicas EPaxos experiences median execution latencies of 125 ms to 139 ms (depending on the site), whereas for five replicas, median execution latencies range from 304 ms to 319 ms (compared to 274 ms to 296 ms for Mencius, and unchanged latencies for Multi-Paxos and Generalized Paxos 100%). As explained in the previous section, this worst case scenario is highly unlikely to occur in practice. Furthermore, commit latency is the only one that matters for writes[5], while for reads, which have a lower chance of generating conflicts, there is a high likelihood that commit and execution latency are the same. Furthermore, reads will also benefit from read leases, which allow reads to be serviced locally.

However, if high command interference is common, there is a wide range of techniques that we can use to reduce latency: e.g., forwarding *PreAcceptReplies* among fast quorum members to reduce slow-path commit latency by one message delay (see below), or reverting to a partitioned Multi-Paxos mode, where the same site acts as command leader for all commands in a certain group (thus eliminating conflicts among the commands within that group).

Reducing Slow-Path Latency in EPaxos

As explained in Section 3.8, we can reduce the slow-path latency in EPaxos by having the members of the fast-path quorum send *PreAcceptReplies* not only to the command leader, but also to each other. The equivalent optimization for Multi-Paxos is to send *AcceptReplies* not only to the stable leader, but also to the site of the client [12] (so that this site can conclude that the command has been committed before being notified by the leader).

We implemented and evaluated these optimizations for both EPaxos and Multi-Paxos (we call them EPaxos-forward and Multi-Paxos-forward). The results are presented in Figure 3.11.

The result is that the slow path in EPaxos becomes just as fast or faster than the regular path in Multi-Paxos (depending on the actual topology of the setup and the choice of the stable Multi-Paxos leader). The only exception is the Multi-Paxos leader site, which experiences a latency of only two wide-area message delays. The EPaxos fast-path latency, on the other hand, is fundamentally lower than that of the Multi-Paxos commit path. This is not reflected in Figure 3.11 because in this particular setup, the stable leader (California) has a peer that is very close (Oregon), and all replicas are almost colinear.

[5] From the client's perspective, there is no difference between a committed but not-yet-executed write, and a write that has been executed (it is, however, guaranteed that execution will occur before subsequent interfering reads).

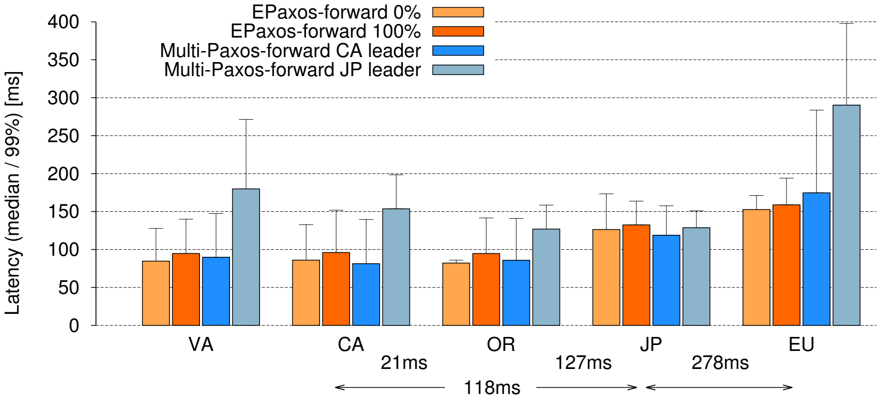


Figure 3.11: Median and 99%ile (as error bars) commit latency when reducing the slow-path latency in EPaxos and the regular commit latency in Multi-Paxos to three wide-area message delays. The median latency for EPaxos 100% corresponds to the slow path, while the median latency for EPaxos 0% corresponds to the fast path. For Multi-Paxos we present two scenarios, one with the leader in California, and the other with the leader in Japan.

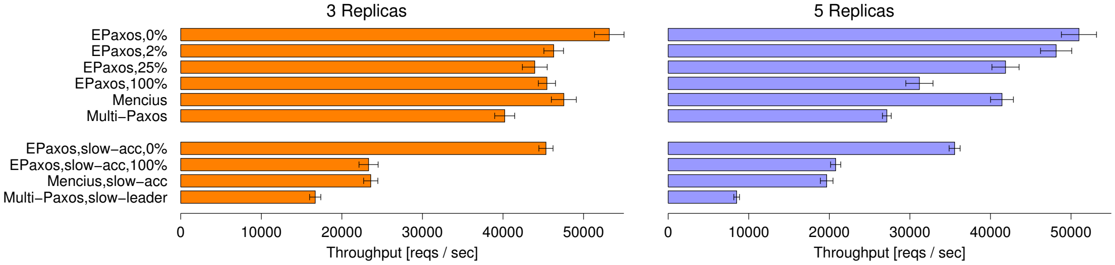


Figure 3.12: Throughput for small (16 B) commands (error bars show 95% CI).

For bad choices of leaders (e.g., Japan in this setup), or for different, non-colinear setups, EPaxos would be always faster on the fast path and at most as slow as Multi-Paxos on the slow path.

3.11.4 Throughput in a Cluster

We compare the throughput achieved by EPaxos, Multi-Paxos, and Mencius, within a single EC2 cluster. We omit Generalized Paxos from these experiments because it was

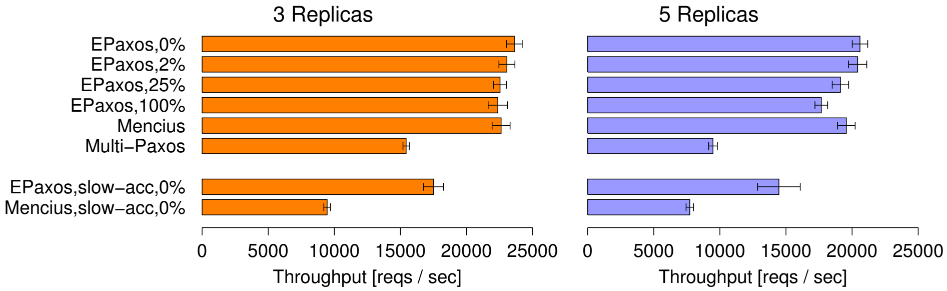


Figure 3.13: Throughput for large (1 KB) commands (with 95% CI).

not designed for high throughput: It runs at less than $\frac{1}{4}$ the speed of EPaxos, and its availability is tied to that of the leader, as for Multi-Paxos.[6]

A client on a separate EC2 instance sends batched requests in an open loop[7] (only client requests are batched; messages between replicas are not), and measures the rate at which it receives replies. For EPaxos and Mencius, the client sends each request to a replica chosen uniformly at random. Replicas reply to the client only *after executing the request*. Although it is often sufficient to acknowledge after commit, we wished to also assess the effects of EPaxos's more complex execution component.

Figure 3.12 shows the throughput achieved by 3 and 5 replicas when the commands are small (16 B). Figure 3.13 shows the throughput achieved with 1 KB requests.

EPaxos outperforms Multi-Paxos because the Multi-Paxos leader becomes bottlenecked by its CPU. By being *thrifty* (Section 3.11.1), EPaxos processes fewer messages per command than Mencius, so its throughput is generally higher—with the notable exception of many conflicts for more than three replicas, when EPaxos executes an extra round per command (Mencius is not significantly influenced by command interference—there was no interference in the Mencius tests). EPaxos messages are slightly larger because they carry attributes, hence our EPaxos implementation incurs more GC overhead.

Processing large commands narrows the gap between protocols: All replicas spend more time sending and receiving commands (either from the client or from the leader), but Mencius and EPaxos exhibit significantly higher throughput than leader-bottlenecked Multi-Paxos.

[6] Learners handle $\Theta(N)$ messages per command and the leader must frequently commit checkpoints—see Section 2.3.

[7] In practice, a client needing linearizability must wait for commit notifications before issuing more commands; the open loop mimics an unbounded number of clients to measure maximum throughput.

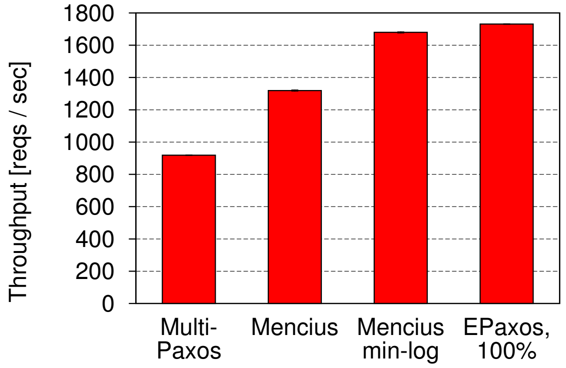


Figure 3.14: Tput for 3 replicas, 16 B commands, sync. log to Flash (w/ 95% CI).

Figures 3.12 and 3.13 also show throughput when one node is slow (for Multi-Paxos that node is the leader—otherwise its throughout is mostly unaffected). In these experiments, two infinite loop programs contend for the two virtual cores on the slow node. EPaxos handles a slow replica better than Mencius or Multi-Paxos because the other replicas can avoid it: Each replica monitors the speed with which its peers process pings over time, and excludes the slowest from its quorums. Mencius, by contrast, fundamentally runs at the speed of the slowest replica because its instances are pre-ordered and a replica cannot commit an instance before learning about instances ordered before it—and $1/N$ of those instances belong to the slow replica.

3.11.5 Logging Messages Persistently

To resume immediately after a crash, a replica must preserve the contents of its memory intact, otherwise it may break safety (for all of the protocols we evaluate). This implies persistently logging every state change before acting upon or replying to any message. The preceding experiments did not include this overhead, because it is avoidable in some circumstances: if power failure of *all* replicas is not a threat, replicas can recover from failures as presented in Section 3.10; in addition, persistent memory technologies keep improving, and battery-backed memory is sometimes feasible. We nevertheless wanted to evaluate whether EPaxos is fundamentally more I/O intensive than Multi-Paxos or Mencius.

For the experiments in this section we used Amazon EC2 High-I/O instances equipped with high-performance solid state drives. Every replica logs its state changes synchronously to an SSD-backed file, for all protocols.

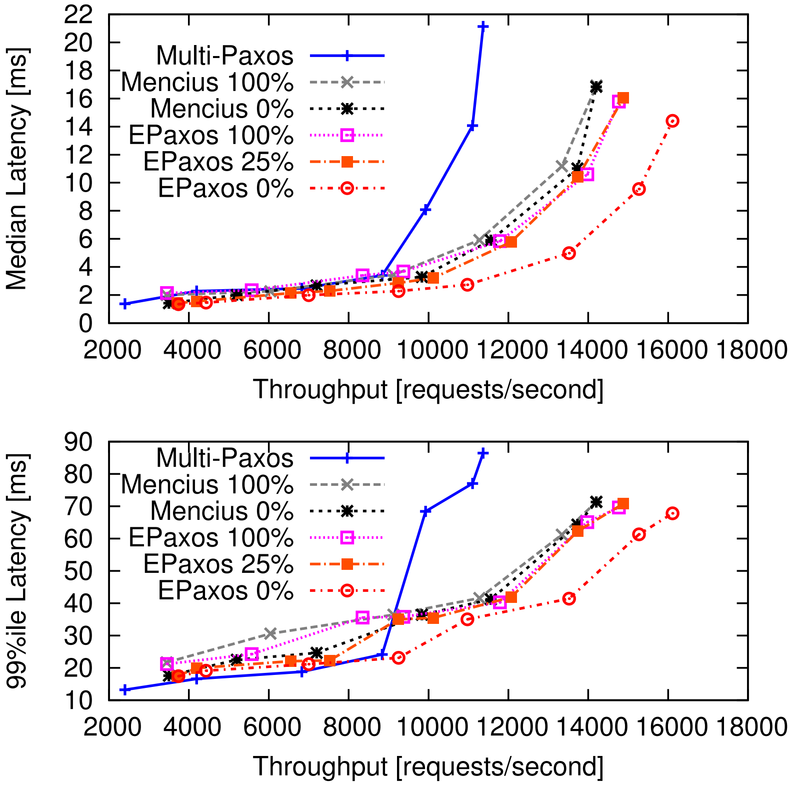


Figure 3.15: Latency vs. throughput for 3 replicas.

Here (Figure 3.14), all protocols are I/O bound, but Multi-Paxos places a higher I/O load on the stable leader than on non-leader replicas, making it slower. EPaxos outperforms Mencius due to the thrifty optimization: in EPaxos, unlike in Mencius, it is sufficient to send (pre-)accept messages to only a quorum of replicas, and therefore EPaxos requires fewer logging operations per command than Mencius. However, we make the (novel, to our knowledge) observation that while every Mencius acceptor must reply to accept messages, not all acceptors must log their replies synchronously—it is sufficient that a quorum of acceptors log synchronously before responding, and the command leader commits only after receiving their replies. "Mencius min-log" (in Figure 3.14), needs only slightly more synchronous logging than EPaxos (every min-log replica must still log its own skipped instances synchronously).

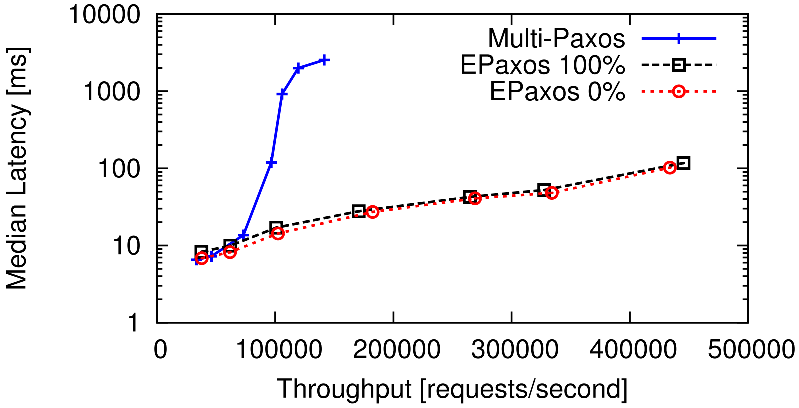


Figure 3.16: Latency vs. throughput for 5 replicas when batching small (16 B) commands every 5 ms.

3.11.6 Execution Latency in a Cluster

This section examines client-perceived execution latency using three replicas. Despite its more complex execution algorithm, EPaxos has lower execution latency than either Multi-Paxos or Mencius, regardless of interfering commands. In addition, our strategy for avoiding livelock in EPaxos's execution algorithm (Section 3.5.6) is effective.

Figure 3.15 shows median (top graph) and 99%ile latency under increasing load in EPaxos, Mencius and Multi-Paxos. We increase throughput by increasing the number of concurrent clients sending commands in a closed loop (each client sends a command and waits until it has been executed before sending the next) from 8 to 300. The maximum throughput is lower than in the throughput experiments because here, replicas bear the additional overhead of hundreds of simultaneous TCP connections.

3.11.7 Batching

Batching increases the maximum throughput of Multi-Paxos by 5x and of EPaxos by 9x (Figure 3.16). Commands are generated open loop from a separate machine in the cluster. Every 5 ms, each proposer batches all requests in its queue, up to a preset maximum batch size: 1000 for EPaxos, 5000 for Multi-Paxos. Command leaders issue notifications to clients only after execution. Each point is the average over ten runs.

EPaxos's advantage here still arises from sharing the load more evenly across replicas, whereas Multi-Paxos places it all on the stable leader. Under the same client throughput,

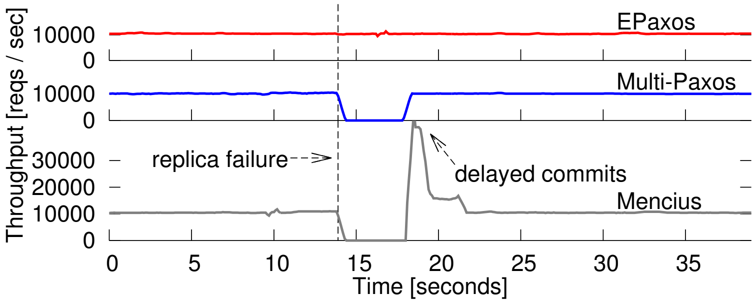


Figure 3.17: Commit throughput when one of three replicas fails. For Multi-Paxos, the leader fails.

Mencius and EPaxos will send up to 5x more messages: each leader will send batches, instead of having one leader aggregate the commands into a single larger batch. However, the cost of these extra messages is amortized rapidly across large batches, becoming negligible versus processing and executing the commands.

Importantly, and perhaps counter-intuitively, batching diminishes the negative effects of command interference in EPaxos. This is because (1) the cost of the extra round of communication for handling a conflict is amortized across multiple commands, and becomes insignificant for large batch sizes (second phase messages are short, because command leaders send only the new attributes to replicas that have already received the batch in the first phase); and (2) at low throughputs, even if all commands interfere, conflicts are less frequent because the possibility of there being multiple batches in flight at the same time (and arriving at different replicas in different orders) diminishes. As a result, EPaxos with 100% interference is effectively as fast as EPaxos with no interference.

Although we have not tested Mencius with batching, as long as replicas do not experience performance variability, we expect it to be as fast as EPaxos, since the difference in messaging patterns has a diminished effect with batching.

3.11.8 Service Availability under Failures

Figure 3.17 shows the evolution of the commit throughput in a three-replica setup that experiences the failure of one replica. A client sends requests in an open loop, at the same rate for every system—approximately 10,000 requests per second (a rate at which none of the systems is close to saturation, hence the steady throughput).

With Multi-Paxos, or any variant that relies on a stable leader, a leader failure prevents the system from processing client requests until a new leader is elected. Although clients could direct their requests to another replica (after they time out), a replica will usually not try to become the new leader immediately. False suspicions can degrade performance by causing stalls, so the fail-over time will usually be on the order of seconds [10, 44]. The failure of a non-leader replica (a situation not depicted in Figure 3.17) does not affect the availability of the system.

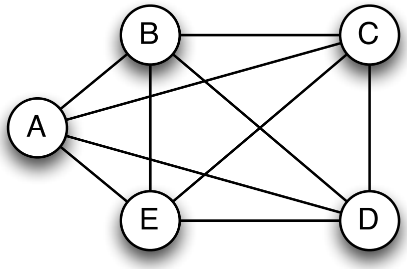
In contrast, any replica failure disrupts Mencius: a replica cannot finalize an instance before knowing the outcome of (or at least which commands are being proposed in) all instances that precede it, and instances are pre-assigned to replicas round-robin. Unlike in Multi-Paxos, clients can continue to send requests to the remaining replicas; they will be processed up to the point where they are ready to be committed. Eventually, a live replica will time out and commit no-ops on behalf of the failed replica, thus freeing the instances waiting on them. At this point, the delayed commands are committed and acknowledged, which causes the throughput spike depicted in Figure 3.17. Live replicas commit no-ops periodically until the failed replica recovers, or until a reconfiguration.

Both in Multi-Paxos and Mencius, the timeout duration is a trade-off between the availability of the service and the impact that acting too frequently on false positives has on throughput and latency. EPaxos avoids this dilemma because it can operate uninterrupted by the crash of a minority of replicas. Clients with commands outstanding at a failed replica will time out and retry those requests at another replica. Although live replicas will commit commands unhindered, some of these commands may have acquired dependencies on commands proposed by the failed replica. Executing the former (as opposed to committing them) will therefore be delayed until another replica finalizes committing the latter. Unlike in Mencius, this occurs only once: an inactive replica cannot continue to generate dependencies. Moreover, it occurs rarely for workloads with low conflict rates.

3.12 Summary of Egalitarian Paxos Benefits

Egalitarian Paxos is the first state machine replication protocol that simultaneously offers constant availability, perfect load balancing across replicas and optimal latency in the wide-area. While other SMR protocols can also achieve high throughput by using batching, EPaxos has lower wide-area latency than any previous strong-consistency protocol. Furthermore, its completely decentralized design means that EPaxos obviates a number of SMR implementation problems: there is no need for leader election or optimizing leader placement in wide-area setups, and adapting to slow replicas becomes significantly easier.

The most important benefit of EPaxos is its low wide-area latency. For example, consider the following setup with five sites:



An EPaxos client at location A would experience a commit delay corresponding to the sum of message propagation delays across the longest of the ABA and AEA paths—max(ABA, AEA)—on the fast path, and max$(ABEA, AEBA) = ABEA$ on the slow path. Generalized Paxos, having a larger fast-path quorum, will incur max$(ABA, AEA,$ min$(ACA, ADA))$ on the fast path (and multiple round trips on the slow path). The latency experienced by a Multi-Paxos client depends on the placement of the leader. For example, if the leader is at B, a client in A will wait for messages to travel across the $ABEA$ path—the same as the slow path in EPaxos—but will have to wait longer for a less advantageous leader placement.

The cost of using Egalitarian Paxos is having to efficiently decide which commands interfere and which do not. This is not, however, a new or unique problem, and previous solutions apply. One solution is using explicitly-specified dependency keys as in Google's High Replication Datastore [20] and Megastore [7]. A better solution is inferring interference automatically. For example, for NoSQL key-value stores where all (or most) operations identify the keys they are targeting, it is straightforward to determine interference. Even for relational databases, the transactions that usually constitute the bulk of the workload are simple and can be examined before execution to determine which rows they will update (e.g. the New-Order transaction in the TPC-C benchmark [53]). For other transactions it will be difficult to predict what exact state they will modify, but it is safe to simply assume they interfere with any other transaction.

Chapter 4

Quorum Read Leases

If EPaxos was about improving the performance of updates for a replicated state machine, in this chapter we discuss about reading the replicated state efficiently. Our contribution is a new mechanism based on time leases that we call *quorum read leases*. While applicable to EPaxos, quorum read leases *are not EPaxos-specific*, and for clarity we present them in the context of Multi-Paxos.

4.1 Overview

Given a replicated state machine implemented using Paxos or a similar quorum-based protocol, simply reading the state of a single replica is not guaranteed to produce consistent results. For example, the replica in question may have stale state (because, in general, not all replicas must be updated synchronously when modifying the state). Furthermore, a previous read may have already read the newer state from a different replica, and therefore using this simple approach will not guarantee that reads are monotonic.

The simplest way of implementing strongly consistent reads is to commit and then execute all reads exactly like any other command. This ensures strict serializability, but has the drawback that reads incur the same high latency in the wide-area as updates. A better approach is to send the read request to any majority of replicas, have these replicas wait for ongoing updates to finish, and then return their result to the client. The client would use the result corresponding to the most recent update (i.e., the one with the highest sequence number). Although better (for example, in leader-based systems we must no longer contact the leader for every read), this approach still incurs wide-area communication delays. Instead, many practical implementations of Paxos use a mechanism based on time leases [7, 10, 13, 16].

The most common lease-based approach is the "leader lease", and it consists in granting the stable Multi-Paxos leader a lease for the entire replicated state [10, 13, 16]. The leader will be able to service reads directly from its local copy of the state because, while the lease is active, all other live replicas guarantee that they will not commit an update that was not proposed by the current leader. This guarantee holds trivially while everyone recognizes the leader as alive, because it is the only one who proposes updates, but it also holds if a new leader is elected while the lease is active.

While the leader lease improves throughput and latency at the leader site, the other non-leader sites still incur high wide-area latency. Megastore [7] introduced the following change: *all* replicas hold a lease for the entire state, so all replicas can read locally. With this approach, read latency is minimal and read throughput is maximal. However, the price that we pay is decreased update performance: (1) all updates must be acknowledged by all replicas synchronously, incurring higher wide-area latency, and (2) if any replica becomes unresponsive, no updates can be performed until the global lease expires.

Here, we argue that the leader lease and the Megastore lease are only two points in a larger design space. A more general approach would grant multiple leases simultaneously to subsets of the replicas (instead of either one replica or all replicas) for parts of the state (instead of the entire state). In particular, the Paxos communication patterns make granting leases to majorities of replicas the sweet spot in the design space, because updates must be acknowledged by majorities anyway, and thus leases have a lower impact on update performance.

Despite the intuitiveness of this approach, implementing quorum leases is nontrivial. Compared to approaches in which the set of nodes with the lease is fixed—either a single master or all replicas—an implementation of quorum leases must be able to consistently determine which objects belong to which lease quorum, automatically determine appropriate lease durations, and efficiently refresh the leases in a way that balances overhead, a high hit rate on leased objects, and rapid lease expiration in the event of a node or network failure. Solving these problems and evaluating the resulting benefits is our primary objective.

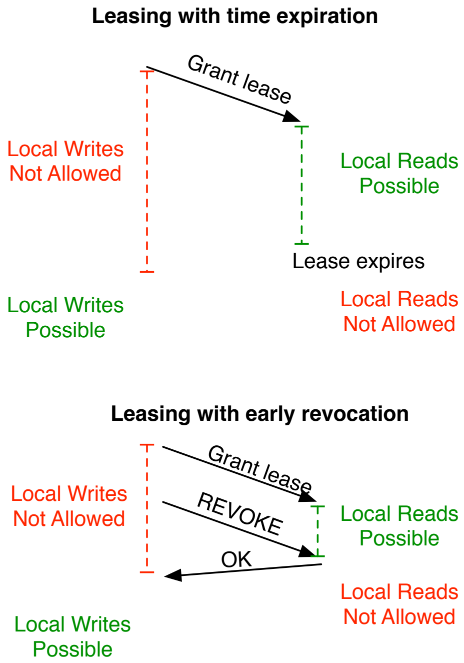


Figure 4.1: Leasing with and without revocation.

4.2 Quorum Leases: Intuition

Recall that a lease is a time-limited promise from one process to another to not modify an object during the *lease duration*. Leases are often coupled with a *revocation* mechanism: If the lease grantor wishes to modify the object before the lease has expired, it must contact each lease holder and receive confirmation that the holder will stop using its local copy of the object, as shown in Figure 4.1.

Quorum leases intertwine the idea of leasing with the natural set of nodes that must be contacted for a Paxos write, bundling the Paxos write operation with the lease revocation. In Paxos, a replica can commit a command only after a majority quorum—possibly including itself—has acknowledged the command. As is apparent from the revocation example in Figure 4.1, if the write quorum includes *all* nodes that hold leases on the object to be modified, then receiving acknowledgements from the write quorum *also* means that the lease has been properly revoked at all replicas.

Of course, this simple design has several complications: (1) if any member of the quorum becomes unavailable, no command can be committed until the lease expires; and (2) replicas outside the quorum cannot perform local reads.

We solve the first problem by carefully choosing the lease duration relative to the round-trip time between the replicas. A quorum is notified synchronously only for a set amount of time, after which none of its members can rely on their state being updated synchronously anymore. If a quorum member crashes, the system will be unavailable only until the lease expires.

To reduce the opportunity cost of not leasing to the full set of nodes, we grant leases on a per-object basis, where each lease might have a different set of nodes holding it. In this way, while not *all* nodes can read locally, the nodes generating the most read traffic for each object can do so. This design is particularly appropriate when the popularity of each object differs substantially across replicas—such as might be the case, e.g., in the popularity of users across a geo-distributed social network.

In conclusion, quorum leases take advantage of the existing communication patterns in Paxos to allow replicas to perform local reads, without substantially decreasing the availability of the system, and without significantly increasing write latency.

4.3 Designing Quorum Leases

We begin by describing the assumptions about the Paxos systems that use quorum leases. We then describe our quorum leases design goals and motivate our design choices.

4.3.1 Assumptions

Communication between the nodes is asynchronous: messages may be lost or delayed indefinitely. Replicas *do not synchronize clocks*, but their clock rates are assumed to be similar, such that a modest guard time can account for clock drift over a short interval. Failures are non-Byzantine: replicas can crash or fail to respond indefinitely, but they will not take actions that do not conform to the protocol.

We assume that the replicated state consists of multiple objects that can be updated and read separately. Clients submit operations that specify which objects will be updated, and queries for reading object values. Multiple updates and queries can be batched in the same command.

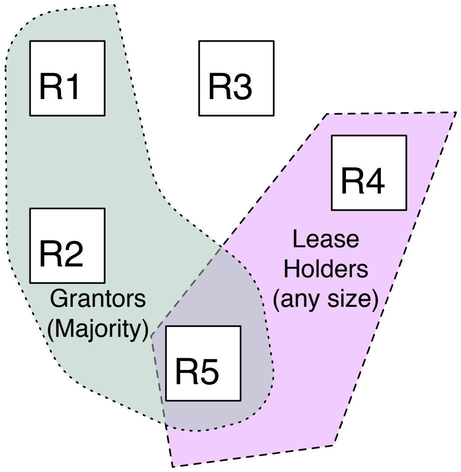


Figure 4.2: An example lease in which a majority of replicas (R1, R2, and R5) have granted leases to two lease holders (R4 and R5).

4.3.2 Design Goals

A *quorum lease* provides a subset of Paxos replicas the guarantee that they will be notified synchronously of any update to a particular set of objects before those updates are committed by any replica. Thus, let $\mathcal{R}$ be the set of all replicas in a Paxos group, and let $\mathcal{O}$ be the set of all objects replicated by this Paxos group. A quorum lease is a pair (Q, O), where $Q \subseteq \mathcal{R}$ and $O \subseteq \mathcal{O}$. We call Q the *lease quorum* for this lease, and O the set of *granted objects*. Every replica $r \in Q$ is a *lease holder*.

Unlike a Paxos quorum, which must include a majority of replicas, a lease quorum can be of any size—e.g., it can contain fewer than half the replicas.

A lease becomes active after a majority of replicas (i.e., at least $\lfloor N/2 \rfloor + 1$ replicas, where $N = |\mathcal{R}|$) have *granted* the lease to at least one lease holder. An example of such a lease is shown in Figure 4.2, where a majority have granted read leases to two nodes in the set. A replica g grants a lease (Q, O) to a replica $r \in Q$ by making a promise:

1. to notify r synchronously before committing any update to any object in O that g proposes (i.e., g must not commit until it receives a message from r in response to its notification), AND

2. to acknowledge Accept and Prepare messages[1] for updates to objects in O only with the condition that the proposer must also notify r synchronously before committing these updates.

To enforce the first condition without extra communication, g can include r in any Paxos quorum of acceptors before committing a Paxos command that it proposes (i.e., for which it is the leader). To enforce the second condition, every replica must attach enough information to its replies to the Paxos Accept and Prepare messages from a proposer that the proposer can determine which replicas the responder has granted leases to; the proposer uses this information to synchronously notify the lease holders before committing. We explain in Section 4.3.4 how to implement this exchange efficiently by adding only a short lease identifier to each message.

To prevent unbounded periods of unavailability, a promise is valid for only a set amount of time, after which it expires. In a correct implementation, a promise must expire at the lease holder before expiring at the grantor. When a lease holder has fewer than $\lfloor N/2 \rfloor$ valid promises from different replicas (the holder itself counts as an implicit grantor), the lease is said to be inactive for that holder.

This mechanism achieves the following: while the lease is active at a lease holder, that lease holder can assess whether an update to any of the leased objects is ongoing (i.e., in the process of being committed), and if not, the lease holder can read the most up-to-date value of that object from its local store.

A quorum lease can be granted to any subset of replicas, of any size. However, because updates in Paxos must be accepted synchronously by a majority of replicas even when not using leases, it is advantageous for both latency and availability to make lease quorums be simple majorities ($\lfloor N/2 \rfloor + 1$ out of the total of N replicas). For reduced write latency, it is also useful that every lease quorum include the current distinguished proposer (i.e., the current stable leader, if the Paxos variant used relies on a stable leader—such as, for example, Multi-Paxos).

Different Paxos replicas may need to read different sets of objects at different times. We therefore wish to be able to update both the set of leased objects, and the lease quorums. We call the totality of quorum leases agreed upon at any given moment a *lease configuration*, and we use Paxos to achieve consensus on lease configuration changes. Conceptually, there is a separate, independent Paxos-replicated state machine for the configuration state; in our implementation, the configuration Paxos instantiation runs on

[1] The condition for acknowledging Prepare messages applies only to instances where the grantor has already accepted an update for an object in O.

the same nodes as the main Paxos instantiation. Because replica clocks are not synchronized, we must establish timing dependencies when granting and refreshing leases. We do this through separate communication, independent of the lease configuration change protocol (see below).

Because an unresponsive lease holder will prevent updates to the leased objects until the lease expires, it is useful for leases to be granted for short periods of time, which are refreshed before they expire.

4.3.3 Design Overview

Although we believe that quorum leases apply to any variant of Paxos, we focus for clarity on the most popular: Multi-Paxos.

While using quorum leases, the replicas of a Paxos system communicate using two categories of messages: (1) The normal Paxos messages for choosing commands; and (2) Lease management messages. We separate these in both design and implementation to make it easier to reuse the leases with other Paxos variants, and to modify the lease management protocol.

Lease management consists of two message sub-types: (2a) Paxos messages for agreeing on lease configuration changes—what are the quorums and what is the set of object IDs granted to each quorum; and (2b) messages for granting and refreshing leases.

To avoid a dependency on external clock synchronization, we set lease timers based upon the causal ordering of messaging events using a lightweight peer-to-peer protocol described in detail in Section 4.3.5. This protocol efficiently combines the computation of lease timers and the granting and refreshing of leases.

Separating lease configuration and granting conveys several advantages. First, it means that the granting protocol could be improved independent of the rest of the system to take advantage of, e.g., hardware clock synchronization or stronger assumptions about delays based upon knowledge of the physical connections between machines. Importantly, the simplicity of the lease granting protocol also makes short lease intervals feasible, which has important benefits for availability. The lease renewal messages are short and can be piggybacked on existing traffic, as they refer to the current lease configuration through a short configuration ID, instead of needing to explicitly describe quorums and granted objects as the lease configuration messages must.

4.3.4 Lease Configurations

A lease configuration describes the quorum composition and the set of granted objects for all the quorum leases in a Paxos system at a given time.

A lease configuration is built incrementally from a sequence of lease configuration changes. Replicas agree on lease configuration changes through Paxos—they participate in Paxos instances that are separate from the normal command-choosing instances. The result is a sequence of lease configuration changes that all (non-faulty) replicas agree upon. A lease configuration is then simply referred to by the instance number of the latest change.

Our basic implementation of quorum leases strives to assign leases such that: (a) the lease is granted to a simple majority of nodes; and (b) the number of locally-satisfied read operations is maximized; and (c) the total number of leases being managed is modest. [2]

It accomplishes this by having replicas track the frequency of reads. The current Multi-Paxos leader replica periodically gathers this information and determines the next lease configuration such that each object is granted to the majority quorum that consists of the stable leader and the $\lfloor N/2 \rfloor$ additional replicas that have read the object most often; it then proposes this new configuration—a set of (quorum, object ID list) pairs—in one of the special Paxos instances. Because it is already guaranteed to see all writes by virtue of being the Multi-Paxos leader, the stable leader is granted a *default lease* that covers every object not in another lease, and is also included in every lease quorum.

There are many possible ways to maintain read statistics. In our current implementation, when a replica receives a read request that it cannot service locally, it forwards it to the stable Multi-Paxos leader. The leader counts the number of forwarded read requests for each object-replica pair. If the number of reads from a given replica is larger than the number of reads from another replica previously included as a lease holder for the object, the leader will include the new replica as a lease holder in the next lease configuration update.

Lease configuration changes happen sufficiently infrequently that they can be written to a stable log without affecting the performance of the system.

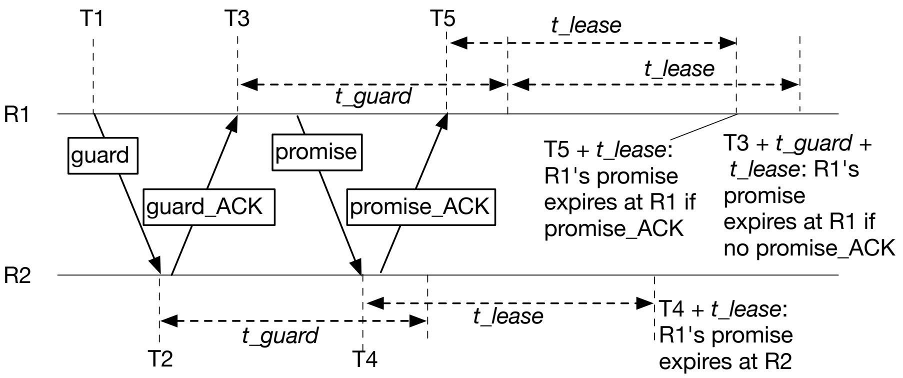

T1
T3
T5
t_lease
t_lease
t_guard
R1
R2
guard
guard_ACK
promise
promise_ACK
T5 + t_lease: R1's promise expires at R1 if promise_ACK
T3 + t_guard + t_lease: R1's promise expires at R1 if no promise_ACK
t_guard
t_lease
T2
T4
T4 + t_lease: R1's promise expires at R2

Figure 4.3: When clocks are not synchronized, the lease activating procedure uses acknowledgements to safely manage lease intervals. In this diagram, the lease duration is *t_lease*. A grantor (R1) will only send a promise if its guard is acknowledged by the holder (R2). The grantor can thus bound the promise duration even if the holder does not reply to the promise.

4.3.5 Activating Leases

Replicas agree on a lease configuration as described in the previous section, but the quorum leases that constitute this configuration become active only after being granted as explained in this section.

The lease configuration covers all leases granted on all objects (which may involve many different sets of holder replicas), and so replicas affirm this configuration in an all-to-all manner. Every replica sends a *promise* to every other replica in the system (thus becoming a grantor). The promise includes the number of the most recent lease configuration that the grantor is aware of, the lease duration, and a timestamp. A receiver (i.e., a lease holder) rejects promises for lease configurations older than the newest one it is aware of.

A lease represents a time during which the grantor will not modify an object without contacting the holder, which gives the holder permission to read that object locally. For safety, the grantor's "will not modify" window of time must be inclusive of the holder's "can read locally" window of time. For high availability, these times should be as short as possible, so that a failed holder can only block writes for a short period of time. These goals are accomplished by the lease establishing and renewal protocol described in Figures 4.3 and 4.4.

[2] Many other optimization goals are both possible and reasonable; exploring them more deeply is an interesting avenue of future work.

Before sending a promise, the grantor sends the holder a *guard* message, which the holder must acknowledge. The guard specifies a time duration t_{guard}. The subsequent promise contains a lease duration t_{lease}. Importantly, the promise is *only* valid if received by the holder before t_{guard} has expired. This ensures that even if the holder does not respond to the promise, the grantor knows that the holder will not believe it has a lease more than $t_{\text{guard}} + t_{\text{lease}}$ seconds after it received the guard ACK.

A holder starts a lease timer as soon as it receives a promise. The promise expires after its specified lease duration (t_{lease}) has passed. A lease holder can consider the lease active while there are at least $\lfloor N/2 \rfloor$ promises received from different peers that have not yet expired.

When renewing active leases, there is no need to send the guard anymore. The most recent acknowledged promise plays the role of the guard: when sending a new promise, the grantor indicates that it must be received within a time $t' + t_{\text{guard}}$ from the most recent received acknowledgment (the grantor indicates which ACK this was), where t' is the time elapsed at the grantor since receiving this acknowledgment. Therefore, the grantor will be able to safely relinquish its promise after $t_{\text{guard}} + t_{\text{lease}}$ seconds from sending the renewal.

When a grantor becomes aware that a new lease configuration has been agreed upon while its most recent promises are still valid, it can take one of two approaches: (1) the grantor can let its current promises expire, and then send promises for the new configuration; or (2) the grantor can immediately send promises for the current configuration, but while both sets of promises are valid, it must abide by the lease rules of both the previous and the current configuration.[3] For simplicity, our current implementation takes the first approach.

The lease establishing and renewing logic is described in pseudocode in Figure 4.4.

4.3.6 Ensuring Strong Consistency

Paxos and Multi-Paxos provide strong consistency: operations are strictly serializable (they are both serializable and the temporal order of non-overlapping operations is respected). In this section we show that quorum leases maintain this consistency guarantee.

Write-only and composite read-write operations will be committed through the normal Paxos protocol and executed atomically. They will thus be strictly serializable. Every

[3] For example, if an object has been granted to a different quorum, the grantor must ensure that it notifies synchronously every replica in the union of the two quorums before committing an update to the object.

Establishing leases

Every replica $\boldsymbol{R}$ becomes a grantor:

```
send Guard(guard_duration) to every other replica
for every GuardACK from any replica $H$ do
    set $grant\_timer_R[H] \leftarrow guard\_duration + lease\_duration$
    send Promise(lease_duration) to H
for every PromiseReply from any replica $H$ do
    if reply received before $grant\_timer_R[H]$ expired then
        set $grant\_timer_R[H] \leftarrow lease\_duration$
```

Any replica $\boldsymbol{H}$, on receiving a

***Guard*(*guard_duration*) from a replica $\boldsymbol{R}$:**

```
set $guard\_timer_H[R] \leftarrow guard\_duration$
reply with a GuardACK
wait for a Promise(lease_duration) from $R$
if Promise received before $guard\_timer_H[R]$ expires then
    set $lease\_timer_H[R] \leftarrow lease\_duration$
    reply with PromiseReply to $R$
```

Renewing leases

Every replica $\boldsymbol{R}$ that is a grantor:

```
for every other replica $H$ do
    set $grant\_timer_R[H] \leftarrow lease\_duration + guard\_duration$
    set $t' \leftarrow$ the time since the most recent ACK from $H$
    set $seq_{\mathrm{ACK}} \leftarrow$ the sequence number of most recent ACK from $H$
    send Promise(lease_duration, $t'$, $seq_{\mathrm{ACK}}$) to $H$
for every PromiseReply from any replica $H$ do
    set $grant\_timer_R[H] \leftarrow \min(grant\_timer_R[H], lease\_duration)$
```

Any replica $\boldsymbol{H}$, on receiving a

***Promise(lease_duration, t′, seq*$_{\mathrm{ACK}}$*)* from a replica $\boldsymbol{R}$:**

```
if Promise received before time $t' + guard\_duration$ since sending ACK with sequence $seq_{\mathrm{ACK}}$ then
    set $lease\_timer_H[R] \leftarrow lease\_time$
    reply with PromiseReply to $R$
```

{A lease holder H can consider the lease active if at least $\lfloor N/2 \rfloor$ promises from different replicas have yet to expire (where N is the total number of replicas).}

Figure 4.4: Establishing and renewing quorum leases.

simple or composite read-only operation will either be serviced atomically by a replica that holds leases for all the objects in question, or will be committed through normal Paxos if no such replica exists. Thus, the system ensures serializability.

To ensure *strict* serializability, notice that it is necessary and sufficient to show that given a read-write or write-only operation W and a read-only operation R, where the write set of W intersects with the read set of R, then the system observes their temporal order. That is to say (1) if R completes before W begins at any replica, R does not observe W; and (2) if W is committed at any replica before R is received by any replica, R observes W. If R is committed through Paxos, this is true by virtue of the Paxos protocol guarantees. We must show that the property also holds when R is serviced locally by some replica.

If R completes before W begins, the property holds trivially: R cannot return a version that does not yet exist anywhere. We are therefore left with the case where W was committed before R was received.

The replica that services R locally (we will call it reader$_R$) can only do so if it has a lease for the objects that R refers to. We must therefore be in one of the following situations:

Case 1: reader$_{\boldsymbol{R}}$ acquired the lease before $\boldsymbol{W}$ began. In this case, reader$_R$'s lease was active throughout W's Paxos commit process. Because the lease is active, by the causal ordering of grantors and holders starting their lease timers, it must be the case that at least $\lfloor N/2 \rfloor + 1$ replicas (possibly including reader$_R$) have granted reader$_R$ the lease, and their promises are still active (i.e., binding). The quorum of replicas that accepted W will intersect this quorum of grantors in at least one replica grantor$_R$. Because grantor$_R$ accepts W, it must be the case that W's proposer will learn that there is an active lease when grantor$_R$ replies to its Accept. Therefore, W's proposer will know that it must notify reader$_R$ before committing W (even if this proposer has not made any promises itself), and reader$_R$ must execute W before executing any further read, including R. In conclusion, R will observe W.

Case 2: reader$_{\boldsymbol{R}}$ acquired the lease after $\boldsymbol{W}$ began. This case has three sub-cases: (1) If there exists an acceptor that is part of W's Paxos quorum and that grants reader$_R$ the lease before accepting W, then this scenario reduces to the previous case. (2) If reader$_R$ itself accepts W before its lease becomes active, it must execute W before R. (3) There exists an acceptor grantor$_R$ that first accepts W and then makes a promise that reader$_R$ takes into account when activating its lease. Because promises contain the index of the most recent accepted Paxos instance at the grantor (therefore at least as recent as W's instance at grantor$_R$), reader$_R$ knows that it must wait for commits for all instances up to and including W's instance before replying to any read (including R). These three sub-cases are exhaustive.

In conclusion, a Paxos system that implements quorum read leases ensures strict serializability.

4.3.7 Recovering after a Replica Failure

The failure of a lease holder will prevent the system from committing updates to any object for which the lease holder has a lease until enough of the promises made to it expire. The system can resume committing once a majority of replicas are no longer bound to notify the faulty replica synchronously. In practice, the time for which it is blocked is on the order of a few to ten seconds, depending on the round-trip time between the replicas, as we analyze more carefully in Section 4.3.8.

A grantor suspects that a replica may have failed if that replica stops replying to its promises or heartbeat messages (common in many Paxos implementations). After a grace period, the grantor will stop trying to renew its promises, and it will let the ones made so far expire. In the meantime, the grantor will request a special lease configuration update that specifies that the replica suspected of failure should be excluded from all quorums it was part of. Replicas that switch to this new configuration no longer need to synchronously notify the possibly-failed replica of updates, and the system can safely resume using leases.

A replica that rejoins the replica set after a failure must wait for $t_{\text{grace}} + t_{\text{lease}}$ seconds before it can accept and/or propose any commands to ensure that all of its promises have expired.

4.3.8 Lease Time and Failures Analysis

In this section we analyze the relationship between the inter-replica RTTs, the lease duration, and the maximum window of write unavailability after a lease holder crashes.

The guard period t_{guard} (Section 4.3.5) must be larger than the maximum round-trip time between any two replicas; otherwise, the soon-to-be lease holder will reject the subsequent promise when it arrives. The lease duration t_{lease}, on the other hand, can be arbitrarily small because grantors can send the lease renewal traffic "blind"—that is, without knowing whether or not its previous lease renewals had been received successfully. However, for renewals to be consistently successful even when messages are lost and retransmitted, the lease duration should be higher than one RTT.

A grantor will stop issuing new leases once it believes a replica is down, as indicated by a failure to reply to some previous message. Such a message loss can only be detected after (at least) one round-trip time between the grantor and grantee. In practice, a grantor will likely wait some additional time before believing a replica failed. We term this total time, from the instant that a replica could have failed to the time its grantor stops issuing lease renewals, the "grace period" t_{grace}.

At the end of the grace period, the grantors will conclude that the node in question has crashed. The grantors can exclude the crashed node from the lease configuration state immediately, but they cannot update any objects leased by the crashed node until they can be certain the crashed (or partitioned) node will not read its objects locally. We term this time t_{wait}, and it is calculated as follows: in the worst case, the grantor sent a promise right before the grace period expired, and therefore, by the renewal logic, it must wait for $t_{\text{wait}} = t_{\text{guard}} + t_{\text{lease}}$ before it can consider this promise expired.

The maximum period of write unavailability caused by a failed lease holder is therefore $t_{\text{lease}} + t_{\text{grace}} + t_{\text{wait}} = 2t_{\text{lease}} + t_{\text{grace}} + t_{\text{guard}}$. For wide-area setups, this will be on the order of seconds (approximately 11 seconds for 2 seconds lease and grant periods and a 5 seconds grace period.)

After t_{grace} expires, but before t_{wait} expires, the live replicas will initiate a lease configuration change (Section 4.3.7) so that they can resume using leases after t_{wait} expires.

If the Paxos leader fails, the unavailability time will be the maximum of the time to elect a leader and $t_{\text{grace}} + t_{\text{wait}}$.

4.3.9 Multi-object Operations and Batching

With quorum leases, different objects may be granted to different quorums. Therefore, multi-object update operations must be synchronously acknowledged by a super-quorum—the union of all quorums that lease an object updated by the multi-object operation. If such operations are frequent, this may affect the performance and the availability of the system. A possible solution is to track those objects that are frequently updated together and ensure they are always granted to the same quorum.

A similar problem arises with batching. Batching increases the throughput of Paxos systems by grouping multiple concurrent operations in a single Paxos command. If these operations update objects leased by different quorums, the batch must be accepted synchronously by the union of all corresponding quorums. To avoid this, replicas must separate objects granted to different quorums into different batches before proposing them. Under heavy request load (usually the situation that warrants batching to sustain a high throughput) there will usually be enough operations of each type for batching to still be effective.

4.4 Evaluating Quorum Leases

We implemented quorum leases, classic leader leases and Megastore-type leases within our Multi-Paxos system written in Go. Because our main focus is on the message and round-trip time reductions (or increases) due to leasing in the wide area, we usually run the system at throughput levels where the implementation details are not the bottleneck. This reduces the importance of a specific choice of Paxos framework.

Because a major focus for all leasing strategies is to reduce latency in the wide area, we implemented as part of our baseline the latency optimization described in [12] (and in Section 3.11.3). This optimization reduces the commit latency perceived by clients that are co-located with a replica other than the Multi-Paxos stable leader, by having other replicas transmit their *AcceptReplies* to both the stable leader and to the replica near the client. Thus, in the common case, the client does not need to wait the additional time for a message to come back from the stable leader, which reduces commit time from four one-way delays to three.

We implemented this optimization because, while it does not appear to be commonly deployed, it is a straightforward algorithmic tweak that reduces commit latency and therefore more accurately represents the state of the art of the write latency that can be achieved by a Paxos-based system. This optimization benefits all three implementations (leader leases, Megastore-type leases, and quorum leases), but in some cases confers slightly more of an advantage to traditional leader leases than to quorum or Megastore-leases.

Without leases, the near-client replica will be able to commit as soon as it receives $\lfloor N/2 \rfloor$ *AcceptReplies*, or $\lfloor N/2 \rfloor - 1$ *AcceptReplies* and an *Accept* from the leader (because it is implicit that the leader must have accepted too). With leases, the commit condition is more strict: the replica closest to the client can commit an operation after receiving *Accept* or *AcceptReplies* messages from all the replicas that hold the lease for the objects updated by the operations (in addition to the previous condition that at least $\lfloor N/2 \rfloor$ replicas in total signal that they have accepted).

	JP	CA	OR	VA	IRL
Japan	0.4	120	120	180	270
California		0.4	20	85	150
Oregon			0.4	75	170
Virginia				0.4	92
Ireland					0.4

Table 4.1: Approximate round-trip times between data centers in milliseconds.

4.4.1 Evaluation Setup

We run our implementations of quorum leases, classic leader leases and Megastore-type leases, both in a single Amazon EC2 cluster, and in a geo-distributed setting: five Multi-Paxos replicas run in five Amazon EC2 data centers, located in Virginia, Northern California, Oregon, Ireland and Japan. Ten clients are co-located (i.e., in the same data center) with each replica. Replicas and clients run on large Amazon EC2 instances: two 64-bit virtual cores with two EC2 Compute Units each and 7.5 GB of memory. The typical RTT in an EC2 cluster is 0.4 ms. The round-trip times between data centers are summarized in Table 4.1.

4.4.2 The Workload

In our experiments, we use Multi-Paxos to replicate a key-value store. Lacking access to, e.g., user traces from a major Internet service, we use the YCSB [15] key-value workload to benchmark it.[4] Every client in our system proposes Put and Get operations with keys drawn from either a Zipf distribution (with an exponent of 0.99—the default YCSB implementation of a Zipf generator) or a uniform distribution. For the Zipf distribution, the most popular items differ across data centers: the sequence of keys ordered by popularity is a different random permutation for each data center. We ran experiments for two workload ratios of Puts to Gets—1:1 and 1:9—with each client choosing the operation type at random. The skewed Zipf distribution is the ideal workload for quorum leases because clients in different data centers will mostly access different objects. The uniformly distributed workload, on the other hand, is the worst case scenario for quorum leases because an object is equally likely to be accessed by replicas that hold a lease for it as by replicas that do not.

[4] For ease of integration with our implementation, we implemented a custom workload generator that uses the same distributions as YCSB via a direct translation of the YCSB code into Go.

4.4.3 Latency

We ran this workload in the wide area, for one hundred thousand keys[5], with fifty simultaneous clients, ten at each of the five locations. Each client sends ten thousand requests to the co-located replica, in a closed loop, and measures the latency (there are thus 500,000 requests in total). For writes, clients are notified as soon as the operation has been committed, so we do not wait for it to be executed. This is sufficient to ensure strict serializability: after receiving the commit notification, the client can safely assume that any subsequent operations (proposed by itself or other clients) will be globally ordered after its write. A read, on the other hand, is executed before notifying the client, so that the read value can be returned with the notification.

In all experiments, the Multi-Paxos leader is in California. Based on the round-trip times shown in Table 4.1, California is the data center closest to the center.

We set the lease parameters as follows: the lease duration was 2 seconds, every grantor renewed the current lease after 500 milliseconds, and the lease configuration was updated every 10 seconds.

The cumulative distribution of latencies, separate for reads and writes, is presented in Figure 4.5 for all three lease strategies: quorum leases, single-leader lease, and Megastore-type leases. For quorum leases we run both Zipf-distributed and uniformly distributed workloads. For the skewed workload, we let the system adapt for 5000 requests from each client before we begin measuring latencies. The ratio of reads to writes is 1:1. This high frequency of writes increases the chance that reads will have to wait for concurrent writes to the same object to finish before they can execute, so, for completeness, we also present a summary of quorum lease results for the 9:1 read to write ratio in Table 4.2 (the other two leasing strategies benefit only marginally from this workload change, their read latencies being alread very low and very high, respectively).

[5] A larger key space would be advantageous for quorum leases, making it less likely for a key to be accessed by a non-lease-holder.

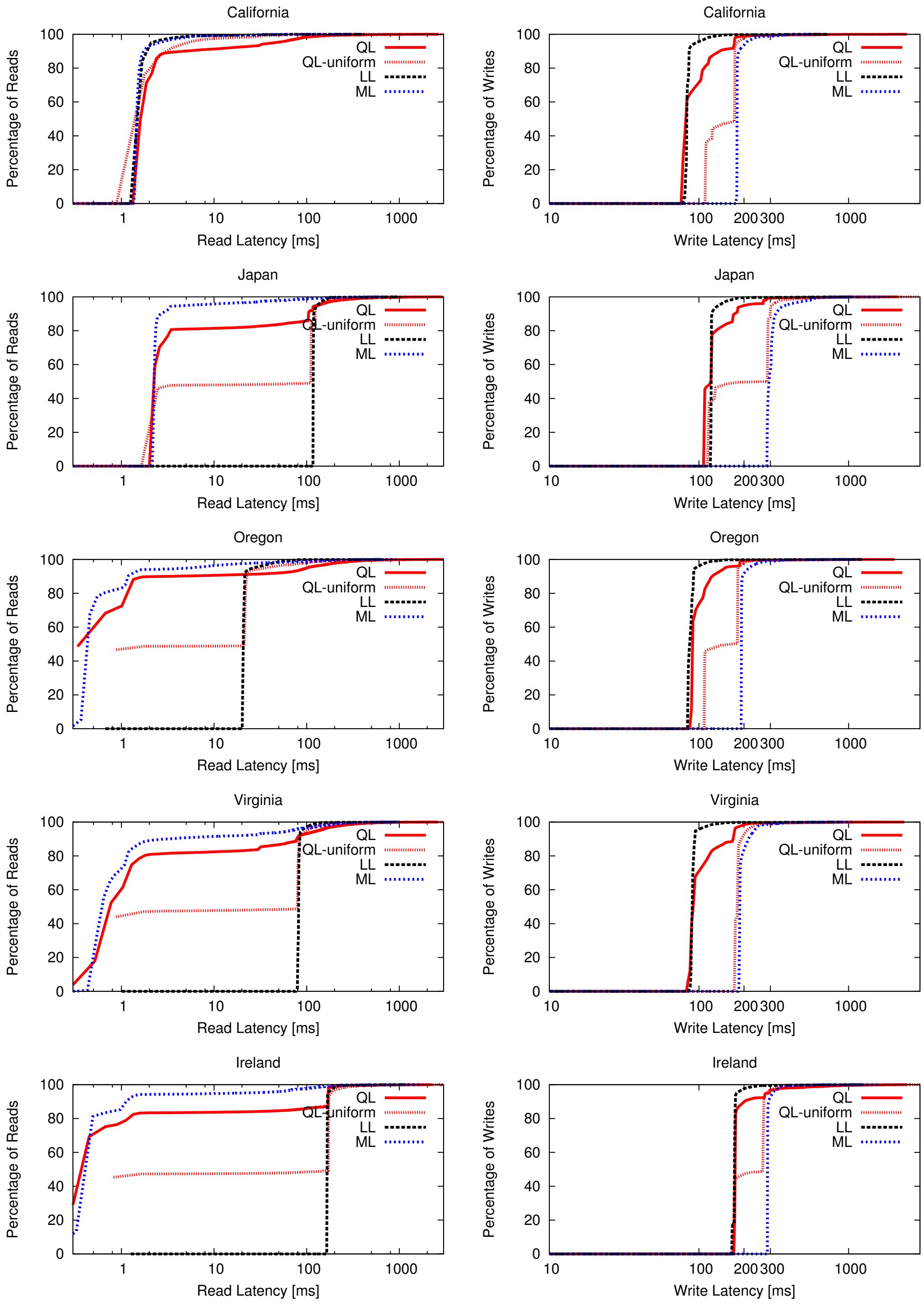


Figure 4.5: CDFs of client-observed latency for each site, with all three lease techniques: quorum lease (QL), single leader lease (LL), and Megastore-type lease (ML). QL-uniform corresponds to quorum leases for a uniformly-distributed workload. The read-to-write ratio in these experiments was 1:1. The Multi-Paxos leader is always located in California. Note the log scale on the X axis.

	Fast local reads
Japan	81%
California	95%
Oregon	89%
Virginia	89%
Ireland	81%

Table 4.2: Percentages of fast local reads (< 10 ms) for wide-area quorum leases with 10% writes and 90% reads, Zipf-distributed.

The single leader lease provides the best write performance because the leader can always choose the fastest quorum for committing a write. The leader is the sole owner of the lease, so no particular other replica must be notified synchronously when an update occurs. The fastest quorum always includes the replica closest to the client, to take full advantage of the *AcceptReplies* forwarding optimization described at the beginning of this section. This low write latency, however, comes at a cost: only the leader can read locally. Thus, the single leader lease suffers mean read latency at least two orders of magnitude larger than that of the competing strategies.

The Megastore-type lease allows every replica to read any object locally, but requires proposers to notify *all* other replicas synchronously before committing a write. Typical (95%) read latencies are under 10 ms—one to two orders of magnitude faster than when communication with a remote data center is required. A small percentage of read requests are delayed by concurrent writes to the same objects: when a write is ongoing, a replica must delay interfering reads until it can be sure that the write will be committed (i.e., typically until it has received a commit notification). The price of this near-optimal read performance is increased write latency, by more than 100 ms in most cases compared to the other leasing schemes. Each replica must wait for the replica *farthest* away to acknowledge an update before committing it and notifying the client. Furthermore, this scheme incurs more risk of unavailability for writes: any unresponsive replica will prevent all updates from progressing until the replica's lease expires.

Quorum leases are a compromise between the two previous schemes. Over 80% of reads at every site are performed locally. Over 70% of updates have the minimum latency achievable by a geo-distributed Multi-Paxos system, matching that provided by the single leader lease, and 2× to 3× faster (i.e. 100 to 200 milliseconds lower latency) than writes with the Megastore approach. Because all replicas are likely to be part of at least one quorum lease, any replica failure will cause some unavailability for writes. However, this does not affect all writes (as it would for Megastore-type leases), but only writes to the objects granted to the failed replica.

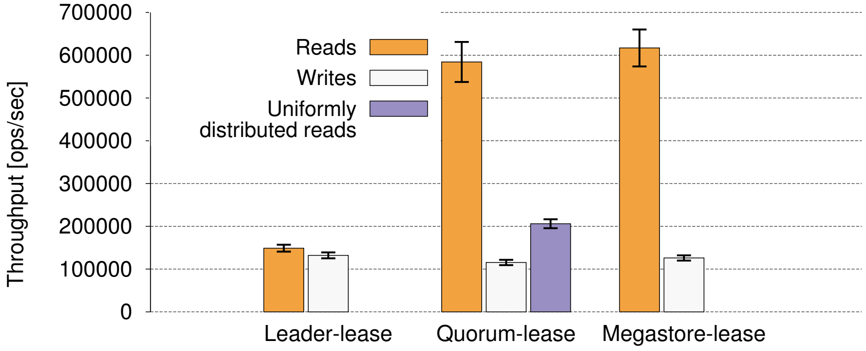


Figure 4.6: Local-area read and write throughput for different leasing strategies. The "Uniformly-distributed reads" for quorum leases corresponds to the situation when clients do not know which replicas can read locally which objects. Error bars represent 95% confidence intervals.

The worst scenario for quorum leases is that when the workload is uniformly distributed (also presented in Figure 4.5). For that experiment, we set the quorum leases statically, based on geographical proximity: one lease includes the replicas in California, Japan and Oregon, the other includes California, Virginia and Ireland. Each leases half the key space. As expected, approximately half the request at each location will therefore have the minimum latency, while the other half will exhibit the worst case latency.

4.4.4 Throughput in a Cluster

While our focus is on wide-area replication, we evaluate the throughput of the three leasing strategies in a local-area cluster as well. Figure 4.6 compares the maximum read and write throughputs achieved with all three leasing strategies. For quorum leases, we present results for two situations: (1) different objects are popular at different replicas—i.e., when trying to read a certain object, all clients know which replica has a lease for that object; and (2) clients are oblivious of lease assignment, and direct their reads uniformly at random across all replicas.

Megastore leases have the best read throughput, narrowly surpassing quorum leases when clients are aware of lease assignments, and 4× higher than single-leader leases. Because we use batching to commit writes (the leader batches up to 5000 updates at a time), the difference in messaging patterns between the three lease implementations is less important, so their write throughputs are approximately the same. Quorum leases have a 10% lower write throughput because we must separate the updates into per-quorum batches (i.e., only updates leased by the same quorum are batched together). In the local cluster, Megastore leases achieve higher throughput, at the cost of suffering more impact upon node failure: *all* leases will be stalled when any replica becomes available, whereas with a quorum lease, an unresponsive replica will stall only updates for the objects it has leased. In general, our results suggest that the differences between Megastore-type leases and quorum leases are less important in a local cluster than they are in the wide area, if throughput is the main consideration.

4.4.5 Discussion

These experiments evaluated only single-object operations. For multi-object operations, if the same objects are usually accessed together, then they will be leased together, and therefore the corresponding read and write latencies will be similar to those reported here. On the other hand, the write latency of multi-object writes that target objects leased by different quorums will approach that of the Megastore leases. If such operations are common, and multi-object reads that span quorums are also common, the Megastore lease may be a more suitable.

4.5 Related Work

Time-based leases have been introduced as a way of improving the latency of reads while maintaining strong data consistency in distributed systems [5, 10, 21, 24, 58]. Quorum leases preserve this goal in the context of Paxos replicated state machines.

Previous systems have used leases to improve the read performance of Paxos systems in two ways. First, Chandra et al. [13] proposed the Paxos leader lease (also used in Chubby [10] and Spanner [16]), which gives the Multi-Paxos leader the ability to perform strongly-consistent reads locally, for a specified time interval. Second, Megastore [7], a wide-area deployment of Paxos, effectively grants every replica in the system a lease for every object: a write in Megastore must synchronously send an invalidate message to every replica that has not accepted the write—so that the remote replica knows its copy of the object (*entity group*, in Megastore terminology) is stale. As shown in our evaluation, the leader lease and the Megastore lease are at opposite extremes of the design spectrum for leases in Paxos: leader leases allow for optimal Multi-Paxos write latency, while Megastore leases sacrifice write latency for optimal read latency at every replica. By contrast, quorum leases allow for a more fine-grained exploration of the design space:

leases can be granted to any subset of replicas and they refer to only a specified portion of the replicated state, so that different replicas can hold different leases at the same time. For a geo-distributed replicated state machine that observes locality in the popularity of its replicated objects, quorum leases can approximate the benefits of both previous approaches simultaneously. This comes at the expense of simplicity, because quorum leases are more complex to manage.

Spanner [16] is a leader-leased, Paxos-based system that uses TrueTime—accurate clock synchronization clock synchronization that requires special hardware—to improve geo-distributed read performance. It does so in two ways: first, it improves the management of leader leases, by taking advantage of their synchronized clocks. Second, it can let remote replicas read locally by issuing *snapshot reads*, at timestamps speficied by clients. Like a quorum lease, a snapshot read executes locally. Unlike a quorum lease, however, a snapshot read may not be up-to-date with respect to updates already committed at other replicas.

Other systems, such as ZooKeeper [25] and MDCC [30], allow fast local reads, but make no freshness guarantees (i.e., the results may be stale).

Previous systems have used Paxos as a mechanism for granting leases: FaTLease [26] and PaxosLease [54] are replicated state machines that grant leases to individual clients. They use Paxos to ensure the fault tolerance of the lease-management system, and take advantage of the perishable property of leases to avoid logging Paxos state to disk. By contrast, quorum leases are granted to the replicas themselves (not the clients) based on the popularity of replicated objects, and they can be granted to multiple entities (replicas) at the same time instead of just one. Furthermore, unlike FaTLease and PaxosLease, quorum leases specify how leases are enforced, not just how they are granted.

The use of leases in improving the performance of distributed protocols is not restricted to Paxos. Zzyzx [22] is a Byzantine fault tolerant system that gives clients exclusive locks to objects. It does so in order to preclude competing client requests, and thus to reduce the common commit path by one message delay (half a round trip) when compared to Zyzzyva [29]. Compared to the mechanism in Zzyzx, quorum leases are granted to the replicas themselves, so multiple (or all) clients can benefit from each lease, and solve a different problem: allowing single-replica consistent reads, an operation generally incompatible with Byzantine fault tolerant systems.

Per-client leases are also used in systems such as Chubby [10] and Farsite [5] to allow clients to operate on cached sub-parts of the replicated data (i.e., cached files).

Quorum leases are superficially similar to the notion of *preferred quorums* [4, 17]. The key distinction is that a quorum lease constitutes a requirement to synchronously update each lease holder, while preferred quorums are a performance optimization. In Q/U [4], clients try to communicate with the same preferred quorum of replicas every time they want to access a particular object, to increase the chance that those replicas are up-to-date with all the latest versions. Quorum leases, by contrast, *guarantee* that replicas in the lease holder subset are up-to-date, so clients can read from any one replica in that subset and not an entire quorum. HQ replication [17] makes only a quorum of replicas execute the full protocol in failure-free situations to reduce the messaging overhead.

4.6 Conclusion

By exploiting the existing messaging requirements for Paxos operations, quorum leases provide a natural, and therefore efficient, mechanism for allowing local reads in a Paxos-replicated system without substantially increasing the delay for write operations to commit. Our evaluation on geo-distributed replicas in Amazon EC2 data centers shows that these leases work well in practice: More nodes can perform reads locally than with simple master-only leases, but the write latency increases only modestly and only does so for typically fewer than 10-20% of operations. We therefore believe that quorum leases are an excellent general-purpose leasing mechanism for Paxos-based systems.

Although we have presented quorum leases only in the context of Multi-Paxos, the most widely used variant of Paxos, we believe they can also be applied to other Paxos variants [37, 45, 47], as well as systems using similar majority-consensus replication protocols [25]. Leaderless Paxos variants, in particular, such as EPaxos and Mencius are a good fit for quorum leases because they do not require a single replica to be part of every write quorum—this gives more flexibility in choosing lease quorums, and benefits both write latency and availability, especially with EPaxos, which optimizes for wide-area Paxos commit latency.

Chapter 5

Conclusion and Future Work

State machine replication is one of the most important building blocks of large-scale distributed systems. This thesis improves the state of the art in the design of strongly consistent replicated state machines.

We introduced Egalitarian Paxos, a new state machine replication protocol that allows for SMR implementations with very high update performance. Egalitarian Paxos is the first protocol to achieve optimal wide-area latency, and simultaneously enable high update throughput and constant availability. Its decentralized design substantially improves the ability of the system to tolerate slow replicas or slow links.

Given the increasing popularity of geo-replicated databases, the main benefit of Egalitarian Paxos is its low wide-area latency. Previous latency optimizations for state machine replication [36, 37] or consistent broadcast protocols [50, 59] minimize the total number of message delays until commit. EPaxos, by contrast, has the same minimal number of inter-replica message delays—which correspond to messages across the wide-area—but one more client-to-replica message delay, which in the wide area is negligible because clients are often co-located with the closest replicas in the same data centers. This allows EPaxos to also minimize the number of replicas that must be contacted, thus achieving lower latency.

If latency is important for updates, it is perhaps even more important for reads, because many databases have a high read-to-write ratio. The mechanism introduced in this thesis, quorum read leases, generalizes previously proposed lease-based read optimizations for Paxos. The result is the ability to perform very low latency, highly-consistent local reads at every replica, with only a minimal impact on write performance.

Quorum read leases are highly effective when the workload exhibits locality—different objects being popular at different replicas. This is likely the case for those setups that straddle multiple continents and time zones.

Besides the performance benefits, EPaxos and quorum leases offer practical implementation benefits. For example, they obviate such implementation problems as leader election or leader placement. Furthermore, achieving high throughput and high performance resiliency requires less engineering effort than with previous algorithms, because EPaxos is designed for load balance and maximum flexibility in choosing quorums. Looking forward, this work would be complemented by taking more of these practical state machine replication implementation aspects and addressing them algorithmically. One example would be proactive reconfigurations—replacing possibly failed replicas proactively with minimum throughput disruption so that the replicated state machine can tolerate many (non-concurrent) failures in a short period of time.

Bibliography

[1] Seattle, WA, November 2006. [Cited on page 104.]

[2] Cascais, Portugal, October 2011. [Cited on pages 107 and 108.]

[3] Hollywood, CA, October 2012. [Cited on page 106.]

[4] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable byzantine fault-tolerant services. In *Proc. 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 59–74, Brighton, UK, October 2005. [Cited on pages 98 and 99.]

[5] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proc. 5th USENIX OSDI*, pages 1–14, Boston, MA, December 2002. [Cited on pages 97 and 98.]

[6] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Thrifty generic broadcast. In *Proc. 14th International Conference on Distributed Computing*, DISC '00, pages 268–282, London, UK, UK, 2000. Springer-Verlag. [Cited on page 12.]

[7] Jason Baker, Chris Bond, James C. Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proc. of the Conference on Innovative Data system Research (CIDR)*, pages 223–234, 2011. [Cited on pages 1, 2, 5, 75, 78 and 97.]

[8] M. Biely, Z. Milosevic, N. Santos, and A. Schiper. S-paxos: Offloading the leader for high throughput state machine replication. In *Reliable Distributed Systems (SRDS), 2012 IEEE 31st Symposium on*, 2012. [Cited on page 12.]

[9] Eric Brewer. Towards robust distributed systems, July 2000. PODC Keynote. [Cited on page 7.]

[10] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. 7th USENIX OSDI* osd [1]. [Cited on pages 2, 10, 64, 74, 78, 97 and 98.]

[11] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proc. 26th annual ACM symposium on Principles of distributed computing*, PODC '07, pages 316–317, New York, NY, USA, 2007. ACM. [Cited on page 11.]

[12] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, November 2002. [Cited on pages 67 and 91.]

[13] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proc. 26th ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM. [Cited on pages 4, 27, 62, 78 and 97.]

[14] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43:225–267, March 1996. [Cited on page 12.]

[15] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010. [Cited on page 92.]

[16] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX OSDI*. USENIX, 2012. [Cited on pages 1, 2, 10, 64, 78, 97 and 98.]

[17] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proc. 7th USENIX OSDI* osd [1], pages 177–190. [Cited on pages 98 and 99.]

[18] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411. [Cited on page 19.]

[19] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002. [Cited on pages 1 and 7.]

[20] Google AppEngine. High replication datastore, 2012. `https://developers.google.com/appengine/docs/java/datastore/overview`. [Cited on page 75.]

[21] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM. [Cited on page 97.]

[22] J. Hendricks, S. Sinnamohideen, G.R. Ganger, and M.K. Reiter. Zzyzx: Scalable fault tolerance through byzantine locking. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 363–372, June 2010. [Cited on page 98.]

[23] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990. [Cited on pages 3, 7, 19 and 59.]

[24] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1), February 1988. [Cited on page 97.]

[25] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, USENIX-ATC'10, Berkeley, CA, USA, 2010. USENIX Association. [Cited on pages 12, 27, 98 and 99.]

[26] Felix Hupfeld, Björn Kolbeck, Jan Stender, Mikael Högqvist, Toni Cortes, Jonathan Marti, and Jesús Malo. Fatlease: scalable fault-tolerant lease negotiation with paxos. In *Proceedings of the 17th international symposium on High performance distributed computing*, HPDC '08, pages 1–10, 2008. [Cited on page 98.]

[27] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, DSN '11, 2011. [Cited on page 12.]

[28] M. Kaptritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. Eve: Execute-verify replication for multi-core servers. In *Proc. 10th USENIX OSDI* osd [3]. [Cited on page 13.]

[29] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *Proc. 21st ACM Symposium on Operating Systems Principles (SOSP)*, pages 45–58, Stevenson, WA, October 2007. [Cited on page 98.]

[30] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. MDCC: Multi-data center consistency. In *Proc. 8th ACM European Conference on Computer Systems (EuroSys)*, April 2013. [Cited on pages 11 and 98.]

[31] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21:558–565, July 1978. [Cited on page 12.]

[32] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2), April 1984. [Cited on page 12.]

[33] Leslie Lamport. The part-time parliament. Technical Report 49, DEC Systems Research Center, 1989. [Cited on page 12.]

[34] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. ISSN 0734-2071. [Cited on pages 2, 10 and 12.]

[35] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), December 2001. [Cited on pages 1, 2, 8, 10 and 17.]

[36] Leslie Lamport. Generalized consensus and Paxos. http://research.microsoft.com/apps/pubs/default.aspx?id=64631, 2005. [Cited on pages 11 and 101.]

[37] Leslie Lamport. Fast Paxos. http://research.microsoft.com/apps/pubs/default.aspx?id=64624, 2006. [Cited on pages 11, 99 and 101.]

[38] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. Technical report, Microsoft Research, 2009. [Cited on page 62.]

[39] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1), March 2010. [Cited on page 62.]

[40] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In *Proc. 10th USENIX OSDI* osd [3]. [Cited on pages 2 and 8.]

[41] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, 2012. [Cited on pages 12 and 62.]

[42] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)* sos [2]. [Cited on pages 2 and 8.]

[43] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013. [Cited on pages 2 and 8.]

[44] John MacCormick, Nick Murphy, Marc Najork, Chandramohan A. Thekkath, and Lidong Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX OSDI*, San Francisco, CA, December 2004. [Cited on pages 2, 10 and 74.]

[45] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proc. 8th USENIX OSDI*, pages 369–384, San Diego, CA, December 2008. [Cited on pages 4, 10 and 99.]

[46] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Epaxos code base. `https://github.com/efficient/epaxos`, August 2013. [Cited on page 63.]

[47] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proc. 24th ACM Symposium on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013. [Cited on page 99.]

[48] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, 1988. [Cited on page 12.]

[49] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, 2014. [Cited on page 12.]

[50] Fernando Pedone and André Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15:97–107, April 2002. [Cited on pages 12 and 101.]

[51] Fernando Pedone and André Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291:79–101, January 2003. [Cited on page 12.]

[52] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)* sos [2]. [Cited on pages 2 and 8.]

[53] tpc-c. TPC benchmark C. http://www.tpc.org/tpcc/spec/tpcc_current.pdf, 2010. [Cited on page 75.]

[54] Marton Trencseni, Attila Gazso, and Holger Reinhardt. PaxosLease: Diskless Paxos for Leases. http://arxiv.org/pdf/1209.4187.pdf, 2012. [Cited on page 98.]

[55] www-ebs-outage. Summary of the Amazon EC2 and Amazon RDS service disruption in the US East Region. https://aws.amazon.com/message/65648/, 2011. Accessed June 2014. [Cited on page 1.]

[56] www-microsoft-infrastructure. Microsoft now has one million servers – less than Google, but more than Amazon, says Ballmer. http://www.extremetech.com/extreme/161772-microsoft-now-has one-million-servers-less-than-google-but-more-than-amazon-says-ballmer, 2013. Accessed June 2014. [Cited on page 1.]

[57] www-sandy-outage. Hurricane Sandy takes data centers offline with flooding, power outages. http://arstechnica.com/information-technology/2012/10/ hurricane-sandy-takes-data-centers-offline-with-flooding-power-outages/, 2012. Accessed Nov. 2012. [Cited on page 1.]

[58] Jian Yin, Lorenzo Alvisi, Michael Dahlin, and Calvin Lin. Volume leases for consistency in large-scale systems. *IEEE Trans. on Knowl. and Data Eng.*, 11(4):563–576, July 1999. [Cited on page 97.]

[59] Piotr Zieliński. Optimistic generic broadcast. In *Proc. 19th International Symposium on Distributed Computing (DISC)*, pages 369–383, Kraków, Poland, September 2005. [Cited on pages 12 and 101.]

APPENDIX

The TLA+ specification of Optimized Egalitarian Paxos.

```
MODULE EgalitarianPaxos

EXTENDS Naturals, FiniteSets

Max(S) ≜ IF S = {} THEN 0 ELSE CHOOSE i ∈ S : ∀ j ∈ S : j ≤ i
```

Constant parameters:
Commands: the set of all possible commands
Replicas: the set of all EPaxos replicas
FastQuorums(r): the set of all fast quorums where r is a command leader
SlowQuorums(r): the set of all slow quorums where r is a command leader

```
CONSTANTS Commands, Replicas, FastQuorums(_), SlowQuorums(_), MaxBallot

ASSUME IsFiniteSet(Replicas)
```

Quorum conditions: (simplified)

```
ASSUME ∀ r ∈ Replicas :
  ∧ SlowQuorums(r) ⊆ SUBSET Replicas
  ∧ ∀ SQ ∈ SlowQuorums(r) :
    ∧ r ∈ SQ
    ∧ Cardinality(SQ) = (Cardinality(Replicas) ÷ 2) + 1

ASSUME ∀ r ∈ Replicas :
  ∧ FastQuorums(r) ⊆ SUBSET Replicas
  ∧ ∀ FQ ∈ FastQuorums(r) :
    ∧ r ∈ FQ
    ∧ Cardinality(FQ) = (Cardinality(Replicas) ÷ 2) +
                        ((Cardinality(Replicas) ÷ 2) + 1) ÷ 2
```

Special none command

```
none ≜ CHOOSE c : c ∉ Commands
```

The instance space

```
Instances ≜ Replicas × (1 . . Cardinality(Commands))
```

The possible status of a command in the log of a replica.

$$\text{Status} \triangleq \{\text{“not-seen”}, \text{“pre-accepted”}, \text{“accepted”}, \text{“committed”}\}$$

All possible protocol messages:

$$
\begin{array}{ll}
\multicolumn{2}{l}{\text{Message} \triangleq} \\
 & [\text{type} : \{\text{“pre-accept”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \\
 & \quad \text{cmd} : \text{Commands} \cup \{\text{none}\}, \text{deps} : \textsc{subset}\ \text{Instances}, \text{seq} : \text{Nat}] \\
\cup & [\text{type} : \{\text{“accept”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \\
 & \quad \text{cmd} : \text{Commands} \cup \{\text{none}\}, \text{deps} : \textsc{subset}\ \text{Instances}, \text{seq} : \text{Nat}] \\
\cup & [\text{type} : \{\text{“commit”}\}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \\
 & \quad \text{cmd} : \text{Commands} \cup \{\text{none}\}, \text{deps} : \textsc{subset}\ \text{Instances}, \text{seq} : \text{Nat}] \\
\cup & [\text{type} : \{\text{“prepare”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}] \\
\cup & [\text{type} : \{\text{“pre-accept-reply”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \\
 & \quad \text{deps} : \textsc{subset}\ \text{Instances}, \text{seq} : \text{Nat}, \text{committed} : \textsc{subset}\ \text{Instances}] \\
\cup & [\text{type} : \{\text{“accept-reply”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}] \\
\cup & [\text{type} : \{\text{“prepare-reply”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \text{prev\_ballot} : \text{Nat} \times \text{Replicas}, \\
 & \quad \text{status} : \text{Status}, \\
 & \quad \text{cmd} : \text{Commands} \cup \{\text{none}\}, \text{deps} : \textsc{subset}\ \text{Instances}, \text{seq} : \text{Nat}] \\
\cup & [\text{type} : \{\text{“try-pre-accept”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \\
 & \quad \text{cmd} : \text{Commands} \cup \{\text{none}\}, \text{deps} : \textsc{subset}\ \text{Instances}, \text{seq} : \text{Nat}] \\
\cup & [\text{type} : \{\text{“try-pre-accept-reply”}\}, \text{src} : \text{Replicas}, \text{dst} : \text{Replicas}, \\
 & \quad \text{inst} : \text{Instances}, \text{ballot} : \text{Nat} \times \text{Replicas}, \text{status} : \text{Status} \cup \{\text{“OK”}\}]
\end{array}
$$

Variables:

comdLog = the commands log at each replica
proposed = command that have been proposed
executed = the log of executed commands at each replica
sentMsg = sent (but not yet received) messages
crtInst = the next instance available for a command leader
leaderOfInst = the set of instances each replica has started but not yet finalized
committed = maps commands to set of commit attributs tuples
ballots = largest ballot number used by any replica
preparing = set of instances that each replica is currently preparing (i.e. recovering)

```
VARIABLES cmdLog, proposed, executed, sentMsg, crtInst, leaderOfInst,
           committed, ballots, preparing

TypeOK ≜
    ∧  cmdLog ∈ [Replicas → SUBSET [inst : Instances,
                                      status : Status,
                                      ballot  : Nat × Replicas,
                                      cmd : Commands ∪ {none},
                                      deps : SUBSET Instances,
                                      seq : Nat]]
    ∧  proposed ∈ SUBSET Commands
    ∧  executed  ∈ [Replicas → SUBSET (Nat × Commands)]
    ∧  sentMsg ∈ SUBSET Message
    ∧  crtInst   ∈ [Replicas → Nat]
    ∧  leaderOfInst ∈ [Replicas → SUBSET Instances]
    ∧  committed ∈ [Instances → SUBSET ((Commands ∪ {none}) ×
                                          (SUBSET Instances) ×
                                          Nat)]
    ∧  ballots ∈ Nat
    ∧  preparing ∈ [Replicas → SUBSET Instances]

vars  ≜  ⟨cmdLog, proposed, executed, sentMsg, crtInst, leaderOfInst,
          committed, ballots, preparing⟩
```

Initial state predicate

```
Init  ≜
  ∧ sentMsg = {}
  ∧ cmdLog = [r ∈ Replicas ↦ {}]
  ∧ proposed = {}
  ∧ executed  = [r ∈ Replicas ↦ {}]
  ∧ crtInst = [r  ∈ Replicas ↦ 1]
  ∧ leaderOfInst = [r ∈ Replicas ↦ {}]
  ∧ committed = [i ∈ Instances ↦ {}]
  ∧ ballots = 1
  ∧ preparing = [r ∈ Replicas ↦ {}]
```

Actions

```
StartPhase1(C, cleader, Q, inst, ballot, oldMsg)  ≜
    LET newDeps  ≜  {rec.inst : rec ∈ cmdLog[cleader]}
        newSeq  ≜  1 + Max({t.seq : t ∈ cmdLog[cleader]})
        oldRecs  ≜  {rec ∈ cmdLog[cleader] : rec.inst = inst}IN
         ∧ cmdLog′ = [cmdLog EXCEPT ![cleader] = (@ \ oldRecs) ∪
                                    {[inst     ↦ inst,
                                      status ↦ "pre-accepted",
                                      ballot ↦ ballot,
                                      cmd    ↦ C,
                                      deps   ↦ newDeps,
                                      seq    ↦ newSeq]}]
                   ∧ leaderOfInst′ = [leaderOfInst EXCEPT ![cleader] = @ ∪ {inst}]
                   ∧ sentMsg′ = (sentMsg \ oldMsg) ∪
                                    [type   : {"pre-accept"},
                                     src    : {cleader},
                                     dst    : Q \ {cleader},
                                     inst   : {inst},
                                     ballot : {ballot},
                                     cmd    : {C},
                                     deps   : {newDeps},
                                     seq    : {newSeq}]

Propose(C, cleader) ≜
   LET newInst ≜ ⟨cleader, crtInst[cleader]⟩
       newBallot ≜ ⟨0, cleader⟩
   IN  ∧ proposed′ = proposed ∪ {C}
       ∧ (∃ Q ∈ FastQuorums(cleader) :
              StartPhase1(C, cleader, Q, newInst, newBallot, {}))
       ∧ crtInst′ = [crtInst EXCEPT ![cleader] = @ + 1]
       ∧ UNCHANGED ⟨executed, committed, ballots, preparing⟩


Phase1Reply(replica) ≜
   ∃ msg ∈ sentMsg :
      ∧ msg.type = "pre-accept"
      ∧ msg.dst = replica
      ∧ LET oldRec ≜ {rec ∈ cmdLog[replica] : rec.inst = msg.inst} IN
          ∧ (∀ rec ∈ oldRec :
              (rec.ballot = msg.ballot ∨ rec.ballot[1] < msg.ballot[1]))
          ∧ LET newDeps ≜ msg.deps ∪
                          ({t.inst : t ∈ cmdLog[replica]} \ {msg.inst})
                newSeq ≜ Max({msg.seq,
                              1 + Max({t.seq : t ∈ cmdLog[replica]})})
                instCom ≜ {t.inst : t ∈ {tt ∈ cmdLog[replica] :
                              tt.status ∈ {"committed", "executed"}}} IN
              ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ oldRec) ∪
                                   {[inst   ↦ msg.inst,
                                     status ↦ "pre-accepted",
                                     ballot ↦ msg.ballot,
                                     cmd    ↦ msg.cmd,
                                     deps   ↦ newDeps,
                                     seq    ↦ newSeq]}]
```

```
                    ∧  sentMsg′ = (sentMsg \ {msg}) ∪
                                       {[type    ↦ "pre-accept-reply",
                                         src     ↦ replica,
                                         dst     ↦ msg.src,
                                         inst    ↦ msg.inst,
                                         ballot  ↦ msg.ballot,
                                         deps    ↦ newDeps,
                                         seq     ↦ newSeq,
                                         committed ↦ instCom]}
                    ∧  UNCHANGED ⟨proposed, crtInst, executed, leaderOfInst,
                                  committed, ballots, preparing⟩

Phase1Fast(cleader, i, Q) ≜
    ∧ i  ∈ leaderOfInst[cleader]
    ∧ Q ∈ FastQuorums(cleader)
    ∧ ∃ record ∈ cmdLog[cleader] :
        ∧ record.inst = i
        ∧ record.status = "pre-accepted"
        ∧ record.ballot[1] = 0
        ∧ LET replies ≜ {msg ∈ sentMsg :
                            ∧ msg.inst = i
                            ∧ msg.type = "pre-accept-reply"
                            ∧ msg.dst = cleader
                            ∧ msg.src ∈ Q
                            ∧ msg.ballot = record.ballot}IN
            ∧  (∀ replica ∈ (Q \ {cleader}) :
                    ∃ msg ∈ replies : msg.src = replica)
            ∧  (∀ r1, r2 ∈ replies :
                    ∧ r1.deps = r2.deps
                    ∧ r1.seq = r2.seq)
            ∧  LET r ≜ CHOOSE r ∈ replies : TRUEIN
                ∧  LET localCom ≜ {t.inst :
                              t ∈ {tt ∈ cmdLog[cleader] :
                                  tt.status ∈ {"committed", "executed"}}}
                        extCom ≜ UNION {msg.committed : msg ∈ replies}IN
                        (r.deps ⊆ (localCom ∪ extCom))
                ∧  cmdLog′ = [cmdLog EXCEPT ![cleader] = (@ \ {record}) ∪
                                        {[inst    ↦ i,
                                          status ↦ "committed",
                                          ballot  ↦ record.ballot,
                                          cmd     ↦ record.cmd,
                                          deps    ↦ r.deps,
                                          seq     ↦ r.seq]}]
                ∧  sentMsg′ = (sentMsg \ replies) ∪
                            {[type  ↦ "commit",
                          inst     ↦ i,
                          ballot   ↦ record.ballot,
                          cmd      ↦ record.cmd,
                          deps     ↦ r.deps,
                          seq      ↦ r.seq]}
                ∧  leaderOfInst′ = [leaderOfInst EXCEPT ![cleader] = @ \ {i}]
                ∧  committed′ = [committed EXCEPT ![i] =
                                                   @ ∪ {⟨record.cmd, r.deps, r.seq⟩}]
                ∧  UNCHANGED ⟨proposed, executed, crtInst, ballots, preparing⟩

Phase1Slow(cleader, i, Q) ≜
    ∧ i  ∈ leaderOfInst[cleader]
    ∧ Q ∈ SlowQuorums(cleader)
    ∧ ∃ record ∈ cmdLog[cleader] :
        ∧ record.inst = i
        ∧ record.status = "pre-accepted"
        ∧ LET replies ≜ {msg ∈ sentMsg :
                              ∧ msg.inst = i
                              ∧ msg.type = "pre-accept-reply"
                              ∧ msg.dst = cleader
                              ∧ msg.src ∈ Q
                              ∧ msg.ballot = record.ballot}IN
            ∧  (∀ replica ∈ (Q \ {cleader}) : ∃ msg ∈ replies : msg.src = replica)
            ∧  LET finalDeps ≜ UNION {msg.deps : msg ∈ replies}
                   finalSeq ≜ Max({msg.seq : msg ∈ replies})IN
                ∧  cmdLog′ = [cmdLog EXCEPT ![cleader] = (@ \ {record}) ∪
                                             {[inst    ↦ i,
                                               status ↦ "accepted",
                                               ballot ↦ record.ballot,
                                               cmd    ↦ record.cmd,
                                               deps   ↦ finalDeps,
                                               seq    ↦ finalSeq]}]
                ∧  ∃ SQ ∈ SlowQuorums(cleader) :
                   (sentMsg′ = (sentMsg \ replies) ∪
                             [type : {"accept"},
                             src : {cleader},
                             dst : SQ \ {cleader},
                             inst : {i},
                             ballot : {record.ballot},
                             cmd : {record.cmd},
                             deps : {finalDeps},
                             seq : {finalSeq}])
                ∧  UNCHANGED ⟨proposed, executed, crtInst, leaderOfInst,
                                committed, ballots, preparing⟩
```

```
Phase2Reply(replica) ≜
    ∃ msg ∈ sentMsg :
        ∧ msg.type = "accept"
        ∧ msg.dst = replica
        ∧ LET oldRec ≜ {rec ∈ cmdLog[replica] : rec.inst = msg.inst} IN
            ∧ (∀ rec      ∈ oldRec : (rec.ballot = msg.ballot ∨
                                      rec.ballot[1] < msg.ballot[1]))
            ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ oldRec) ∪
                                {[inst    ↦ msg.inst,
                                  status ↦ "accepted",
                                  ballot ↦ msg.ballot,
                                  cmd    ↦ msg.cmd,
                                  deps   ↦ msg.deps,
                                  seq     ↦ msg.seq]}]
            ∧ sentMsg′ = (sentMsg \ {msg}) ∪
                                {[type   ↦ "accept-reply",
                                  src     ↦ replica,
                                  dst     ↦ msg.src,
                                  inst    ↦ msg.inst,
                                  ballot ↦ msg.ballot]}
            ∧ UNCHANGED ⟨proposed, crtInst, executed, leaderOfInst,
                          committed, ballots, preparing⟩

Phase2Finalize(cleader, i, Q) ≜
    ∧ i   ∈ leaderOfInst[cleader]
    ∧ Q ∈ SlowQuorums(cleader)
    ∧ ∃ record ∈ cmdLog[cleader] :
        ∧ record.inst = i
        ∧ record.status = "accepted"
        ∧ LET replies ≜ {msg ∈ sentMsg :
                                ∧ msg.inst = i
                                ∧ msg.type = "accept-reply"
                                ∧ msg.dst = cleader
                                ∧ msg.src ∈ Q
                                ∧ msg.ballot = record.ballot} IN
            ∧ (∀ replica ∈ (Q \ {cleader}) : ∃ msg ∈ replies :
                                                msg.src = replica)
            ∧ cmdLog′ = [cmdLog EXCEPT ![cleader] = (@ \ {record}) ∪
                                    {[inst    ↦ i,
                                      status ↦ "committed",
                                      ballot ↦ record.ballot,
                                      cmd    ↦ record.cmd,
                                      deps   ↦ record.deps,
                                      seq     ↦ record.seq]}]
        ∧  sentMsg′ = (sentMsg \ replies) ∪
                      {[type   ↦ “commit”,
                        inst   ↦ i,
                        ballot ↦ record.ballot,
                        cmd    ↦ record.cmd,
                        deps   ↦ record.deps,
                        seq    ↦ record.seq]}
        ∧  committed′ = [committed EXCEPT ![i] = @ ∪
                            {⟨record.cmd, record.deps, record.seq⟩}]
        ∧  leaderOfInst′ = [leaderOfInst EXCEPT ![cleader] = @ \ {i}]
        ∧  UNCHANGED ⟨proposed, executed, crtInst, ballots, preparing⟩

Commit(replica, cmsg) ≜
    LET oldRec ≜ {rec ∈ cmdLog[replica] : rec.inst = cmsg.inst} IN
        ∧ ∀ rec ∈ oldRec : (rec.status ∉ {“committed”, “executed”} ∧
                            rec.ballot[1] ≤ cmsg.ballot[1])
        ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ oldRec) ∪
                                        {[inst   ↦ cmsg.inst,
                                          status ↦ “committed”,
                                          ballot ↦ cmsg.ballot,
                                          cmd    ↦ cmsg.cmd,
                                          deps   ↦ cmsg.deps,
                                          seq    ↦ cmsg.seq]}]
        ∧ committed′ = [committed EXCEPT ![cmsg.inst] = @ ∪
                            {⟨cmsg.cmd, cmsg.deps, cmsg.seq⟩}]
        ∧ UNCHANGED ⟨proposed, executed, crtInst, leaderOfInst,
                        sentMsg, ballots, preparing⟩
```

Recovery actions

```
SendPrepare(replica, i, Q) ≜
    ∧ i ∉ leaderOfInst[replica]
    ∧ ballots ≤ MaxBallot
    ∧ ¬(∃ rec ∈ cmdLog[replica] :
                ∧ rec.inst = i
                ∧ rec.status ∈ {“committed”, “executed”})
    ∧ sentMsg′ = sentMsg ∪
                [type   : {“prepare”},
                 src    : {replica},
                 dst    : Q,
                 inst   : {i},
                 ballot : {⟨ballots, replica⟩}]
    ∧ ballots′ = ballots + 1
    ∧ preparing′ = [preparing EXCEPT ![replica] = @ ∪ {i}]
    ∧ UNCHANGED ⟨cmdLog, proposed, executed, crtInst,
                   leaderOfInst, committed⟩

ReplyPrepare(replica) ≜
   ∃ msg ∈ sentMsg :
      ∧ msg.type = "prepare"
      ∧ msg.dst = replica
      ∧  ∨ ∃ rec ∈ cmdLog[replica] :
             ∧ rec.inst = msg.inst
             ∧ msg.ballot[1] > rec.ballot[1]
             ∧ sentMsg′ = (sentMsg \ {msg}) ∪
                         {[type   ↦ "prepare-reply",
                           src    ↦ replica,
                           dst    ↦ msg.src,
                           inst   ↦ rec.inst,
                           ballot ↦ msg.ballot,
                           prev_ballot ↦ rec.ballot,
                           status ↦ rec.status,
                           cmd    ↦ rec.cmd,
                           deps   ↦ rec.deps,
                           seq    ↦ rec.seq]}
              ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ {rec}) ∪
                         {[inst   ↦ rec.inst,
                           status ↦ rec.status,
                           ballot ↦ msg.ballot,
                           cmd    ↦ rec.cmd,
                           deps   ↦ rec.deps,
                           seq    ↦ rec.seq]}]
              ∧ IF rec.inst ∈ leaderOfInst[replica] THEN
                    ∧ leaderOfInst′ = [leaderOfInst EXCEPT ![replica] =
                                                         @ \ {rec.inst}]
                    ∧ UNCHANGED ⟨proposed, executed, committed,
                                 crtInst, ballots, preparing⟩
                ELSE  UNCHANGED ⟨proposed, executed, committed, crtInst,
                                 ballots, preparing, leaderOfInst⟩

         ∨ ∧ ¬(∃ rec   ∈ cmdLog[replica] : rec.inst = msg.inst)
           ∧ sentMsg′ = (sentMsg \ {msg}) ∪
                         {[type   ↦ "prepare-reply",
                           src    ↦ replica,
                           dst    ↦ msg.src,
                           inst   ↦ msg.inst,
                           ballot ↦ msg.ballot,
                           prev_ballot ↦ ⟨0, replica⟩,
                           status ↦ "not-seen",
                           cmd    ↦ none,
                    deps   ↦ {},
                    seq    ↦ 0]}
          ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = @ ∪
                   {[inst   ↦ msg.inst,
                    status ↦ "not-seen",
                    ballot ↦ msg.ballot,
                    cmd    ↦ none,
                    deps   ↦ {},
                    seq    ↦ 0]}]
          ∧ UNCHANGED ⟨proposed, executed, committed, crtInst, ballots,
                       leaderOfInst, preparing⟩

PrepareFinalize(replica, i, Q) ≜
    ∧ i ∈ preparing[replica]
    ∧ ∃ rec ∈ cmdLog[replica] :
        ∧ rec.inst = i
        ∧ rec.status ∉ {"committed", "executed"}
        ∧ LET replies ≜ {msg ∈ sentMsg :
                          ∧ msg.inst = i
                          ∧ msg.type = "prepare-reply"
                          ∧ msg.dst = replica
                          ∧ msg.ballot = rec.ballot}IN
            ∧ (∀ rep ∈ Q : ∃ msg    ∈ replies : msg.src = rep)
            ∧   ∨ ∃ com ∈ replies :
                        ∧ (com.status ∈ {"committed", "executed"})
                        ∧ preparing′ = [preparing EXCEPT ![replica] = @ \ {i}]
                        ∧ sentMsg′ = sentMsg \ replies
                        ∧ UNCHANGED ⟨cmdLog, proposed, executed, crtInst, leaderOfInst,
                                     committed, ballots⟩
                ∨ ∧ ¬(∃ msg ∈ replies : msg.status ∈ {"committed", "executed"})
                  ∧ ∃ acc ∈ replies :
                        ∧ acc.status = "accepted"
                        ∧ (∀ msg ∈ (replies \ {acc}) :
                            (msg.prev_ballot[1] ≤ acc.prev_ballot[1] ∨
                             msg.status ≠ "accepted"))
                        ∧ sentMsg′ = (sentMsg \ replies) ∪
                                 [type   : {"accept"},
                                  src    : {replica},
                                  dst    : Q \ {replica},
                                  inst   : {i},
                                  ballot : {rec.ballot},
                                  cmd    : {acc.cmd},
                                  deps   : {acc.deps},
                                  seq    : {acc.seq}]
                        ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ {rec}) ∪
```

```
                      {[inst   ↦ i,
                        status ↦ "accepted",
                        ballot ↦ rec.ballot,
                        cmd    ↦ acc.cmd,
                        deps   ↦ acc.deps,
                        seq    ↦ acc.seq]}]
                 ∧ preparing′ = [preparing EXCEPT ![replica] = @ \ {i}]
                 ∧ leaderOfInst′ = [leaderOfInst EXCEPT ![replica] = @ ∪ {i}]
                 ∧ UNCHANGED ⟨proposed, executed, crtInst, committed, ballots⟩
    ∨ ∧ ¬(∃ msg ∈ replies :
              msg.status ∈ {"accepted", "committed", "executed"})
      ∧ LET preaccepts ≜ {msg ∈ replies : msg.status = "pre-accepted"}IN
         ( ∨ ∧ ∀ p1, p2 ∈ preaccepts :
                      p1.cmd = p2.cmd ∧ p1.deps = p2.deps ∧ p1.seq = p2.seq
             ∧ ¬(∃ pl ∈ preaccepts : pl.src        = i[1])
             ∧ Cardinality(preaccepts) ≥ Cardinality(Q) − 1
              ∧ LET pac ≜ CHOOSE pac ∈ preaccepts : TRUEIN
                  ∧   sentMsg′ = (sentMsg \ replies) ∪
                               [type   : {"accept"},
                                src    : {replica},
                                dst    : Q \ {replica},
                                inst   : {i},
                                ballot : {rec.ballot},
                                cmd    : {pac.cmd},
                                deps   : {pac.deps},
                                seq    : {pac.seq}]
                  ∧   cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ {rec}) ∪
                              {[inst   ↦ i,
                                status ↦ "accepted",
                                ballot ↦ rec.ballot,
                                cmd    ↦ pac.cmd,
                                deps   ↦ pac.deps,
                                seq    ↦ pac.seq]}]
                   ∧ preparing′ = [preparing EXCEPT ![replica] = @ \ {i}]
                   ∧ leaderOfInst′ = [leaderOfInst EXCEPT ![replica] = @ ∪ {i}]
                   ∧ UNCHANGED ⟨proposed, executed, crtInst, committed, ballots⟩
           ∨ ∧ ∀ p1, p2 ∈ preaccepts : p1.cmd = p2.cmd ∧
                                       p1.deps = p2.deps ∧
                                       p1.seq = p2.seq
             ∧ ¬(∃ pl ∈ preaccepts : pl.src = i[1])
             ∧ Cardinality(preaccepts) < Cardinality(Q) − 1
             ∧ Cardinality(preaccepts) ≥ Cardinality(Q) ÷ 2
              ∧ LET pac ≜ CHOOSE pac ∈ preaccepts : TRUEIN
                  ∧   sentMsg′ = (sentMsg \ replies) ∪
                               [type   : {"try-pre-accept"},
                                src    : {replica},
                                dst    : Q,
                                inst   : {i},
                                ballot : {rec.ballot},
                                cmd    : {pac.cmd},
                                deps   : {pac.deps},
                                seq    : {pac.seq}]
                        ∧  preparing′ = [preparing EXCEPT ![replica] = @ \ {i}]
                        ∧  leaderOfInst′ = [leaderOfInst EXCEPT ![replica] = @ ∪ {i}]
                        ∧  UNCHANGED ⟨cmdLog, proposed, executed,
                                        crtInst, committed, ballots⟩
                   ∨  ∧  ∨ ∃ p1, p2 ∈ preaccepts : p1.cmd ≠ p2.cmd ∨
                                                    p1.deps ≠ p2.deps ∨
                                                    p1.seq ≠ p2.seq
                        ∨ ∃ pl ∈ preaccepts : pl.src = i[1]
                        ∨ Cardinality(preaccepts) < Cardinality(Q) ÷ 2
                      ∧ preaccepts ≠ {}
                      ∧ LET pac ≜ CHOOSE pac ∈ preaccepts : pac.cmd ≠ noneIN
                          ∧  StartPhase1(pac.cmd, replica, Q, i, rec.ballot, replies)
                          ∧  preparing′ = [preparing EXCEPT ![replica] = @ \ {i}]
                          ∧  UNCHANGED ⟨proposed, executed, crtInst, committed, ballots⟩)
             ∨   ∧ ∀ msg ∈ replies : msg.status = "not-seen"
                 ∧ StartPhase1(none, replica, Q, i, rec.ballot, replies)
                 ∧ preparing′ = [preparing EXCEPT ![replica] = @ \ {i}]
                 ∧ UNCHANGED ⟨proposed, executed, crtInst, committed, ballots⟩

ReplyTryPreaccept(replica) ≜
  ∃ tpa ∈ sentMsg :
      ∧ tpa.type = "try-pre-accept"
      ∧ tpa.dst = replica
      ∧ LET oldRec ≜ {rec ∈ cmdLog[replica] : rec.inst = tpa.inst}IN
         ∧  ∀ rec ∈ oldRec : rec.ballot[1] ≤ tpa.ballot[1] ∧
                             rec.status ∉ {"accepted", "committed", "executed"}
         ∧   ∨ (∃ rec ∈ cmdLog[replica] \ oldRec :
                  ∧ tpa.inst ∉ rec.deps
                  ∧ ∨ rec.inst ∉ tpa.deps
                    ∨ rec.seq ≥ tpa.seq
                  ∧ sentMsg′ = (sentMsg \ {tpa}) ∪
                           {[type   ↦ "try-pre-accept-reply",
                             src    ↦ replica,
                             dst    ↦ tpa.src,
                             inst   ↦ tpa.inst,
                             ballot ↦ tpa.ballot,
                             status ↦ rec.status]})
                  ∧ UNCHANGED ⟨cmdLog, proposed, executed, committed, crtInst,
                                   ballots, leaderOfInst, preparing⟩
                   ∨ ∧ (∀ rec ∈ cmdLog[replica] \ oldRec :
                             tpa.inst ∈ rec.deps ∨ (rec.inst ∈ tpa.deps ∧
                                                     rec.seq < tpa.seq))
                     ∧ sentMsg′ = (sentMsg \ {tpa}) ∪
                                   {[type   ↦ "try-pre-accept-reply",
                                     src    ↦ replica,
                                     dst    ↦ tpa.src,
                                     inst   ↦ tpa.inst,
                                     ballot ↦ tpa.ballot,
                                     status ↦ "OK"]}
                     ∧ cmdLog′ = [cmdLog EXCEPT ![replica] = (@ \ oldRec) ∪
                                   {[inst   ↦ tpa.inst,
                                     status ↦ "pre-accepted",
                                     ballot ↦ tpa.ballot,
                                     cmd    ↦ tpa.cmd,
                                     deps   ↦ tpa.deps,
                                     seq    ↦ tpa.seq]}]
                     ∧ UNCHANGED ⟨proposed, executed, committed, crtInst, ballots,
                                  leaderOfInst, preparing⟩

FinalizeTryPreAccept(cleader, i, Q) ≜
    ∃ rec ∈ cmdLog[cleader] :
        ∧ rec.inst = i
        ∧ LET tprs ≜ {msg ∈ sentMsg : msg.type = "try-pre-accept-reply" ∧
                              msg.dst = cleader ∧ msg.inst = i ∧
                              msg.ballot = rec.ballot}IN
            ∧ ∀ r ∈ Q : ∃ tpr ∈ tprs : tpr.src = r
            ∧ ∨ ∧ ∀ tpr ∈ tprs : tpr.status = "OK"
                ∧ sentMsg′ = (sentMsg \ tprs) ∪
                              [type   : {"accept"},
                               src    : {cleader},
                               dst    : Q \ {cleader},
                               inst   : {i},
                               ballot : {rec.ballot},
                               cmd    : {rec.cmd},
                               deps   : {rec.deps},
                               seq    : {rec.seq}]
                ∧ cmdLog′ = [cmdLog EXCEPT ![cleader] = (@ \ {rec}) ∪
                              {[inst   ↦ i,
                                status ↦ "accepted",
                                ballot ↦ rec.ballot,
                                cmd    ↦ rec.cmd,
                                deps   ↦ rec.deps,
                    $\text{seq} \mapsto \text{rec.seq}]\}]$
              $\land$ UNCHANGED $\langle$proposed, executed, committed, crtInst, ballots,
                          leaderOfInst, preparing$\rangle$
           $\lor$ $\land$ $\exists$ tpr $\in$ tprs : tpr.status $\in$ {"accepted", "committed", "executed"}
              $\land$ StartPhase1(rec.cmd, cleader, Q, i, rec.ballot, tprs)
              $\land$ UNCHANGED $\langle$proposed, executed, committed, crtInst, ballots,
                          leaderOfInst, preparing$\rangle$
           $\lor$ $\land$ $\exists$ tpr $\in$ tprs : tpr.status = "pre-accepted"
              $\land$ $\forall$ tpr $\in$ tprs : tpr.status $\in$ {"OK", "pre-accepted"}
              $\land$ sentMsg' = sentMsg \ tprs
              $\land$ leaderOfInst' = [leaderOfInst EXCEPT ![cleader] = @ \ {i}]
              $\land$ UNCHANGED $\langle$cmdLog, proposed, executed, committed, crtInst,
                          ballots, preparing$\rangle$
```

Action groups

```
CommandLeaderAction $\triangleq$
    $\lor$ ($\exists$ C $\in$ (Commands \ proposed) :
         $\exists$ cleader $\in$ Replicas : Propose(C, cleader))
    $\lor$ ($\exists$ cleader $\in$ Replicas : $\exists$ inst $\in$ leaderOfInst[cleader] :
         $\lor$ ($\exists$ Q $\in$ FastQuorums(cleader)  : Phase1Fast(cleader, inst, Q))
         $\lor$ ($\exists$ Q $\in$ SlowQuorums(cleader) : Phase1Slow(cleader, inst, Q))
         $\lor$ ($\exists$ Q $\in$ SlowQuorums(cleader) : Phase2Finalize(cleader, inst, Q))
         $\lor$ ($\exists$ Q $\in$ SlowQuorums(cleader) : FinalizeTryPreAccept(cleader, inst, Q)))

ReplicaAction $\triangleq$
   $\exists$ replica $\in$ Replicas :
      ( $\lor$ Phase1Reply(replica)
        $\lor$ Phase2Reply(replica)
        $\lor$ $\exists$ cmsg $\in$ sentMsg : (cmsg.type = "commit" $\land$ Commit(replica, cmsg))
        $\lor$ $\exists$ i $\in$ Instances :
           $\land$ crtInst[i[1]] > i[2]  This condition states that the instance has
                                  been started by its original owner
           $\land$ $\exists$ Q $\in$ SlowQuorums(replica) : SendPrepare(replica, i, Q)
        $\lor$ ReplyPrepare(replica)
        $\lor$ $\exists$ i   $\in$ preparing[replica] :
           $\exists$ Q $\in$ SlowQuorums(replica) : PrepareFinalize(replica, i, Q)
        $\lor$ ReplyTryPreaccept(replica))
```

Next action

```
Next $\triangleq$
    $\lor$ CommandLeaderAction
    $\lor$ ReplicaAction
```

The complete definition of the algorithm

$$\text{Spec} \triangleq \text{Init} \land \Box[\text{Next}]_{\text{vars}}$$

Theorems

```
$\text{Nontriviality} \triangleq$
    $\forall\, \text{i} \in \text{Instances} :$
        $\Box(\forall\, \text{C} \in \text{committed[i]} : \text{C} \in \text{proposed} \lor \text{C} = \text{none})$

$\text{Stability} \triangleq$
    $\forall\, \text{replica} \in \text{Replicas} :$
        $\forall\, \text{i} \in \text{Instances} :$
            $\forall\, \text{C} \in \text{Commands} :$
                $\Box((\exists\, \text{rec1} \in \text{cmdLog[replica]} :$
                     $\land\ \text{rec1.inst} = \text{i}$
                     $\land\ \text{rec1.cmd} = \text{C}$
                     $\land\ \text{rec1.status} \in \{\text{“committed”}, \text{“executed”}\}) \Rightarrow$
                    $\Box(\exists\, \text{rec2} \in \text{cmdLog[replica]} :$
                        $\land\ \text{rec2.inst} = \text{i}$
                        $\land\ \text{rec2.cmd} = \text{C}$
                        $\land\ \text{rec2.status} \in \{\text{“committed”}, \text{“executed”}\}))$

$\text{Consistency} \triangleq$
    $\forall\, \text{i} \in \text{Instances} :$
        $\Box(\text{Cardinality(committed[i])} \leq 1)$

THEOREM $\text{Spec} \Rightarrow (\Box\text{TypeOK}) \land \text{Nontriviality} \land \text{Stability} \land \text{Consistency}$
```