There Is More Consensus in Egalitarian Parliaments

Iulian Moraru, David G. Andersen, Michael Kaminsky
Carnegie Mellon University and Intel Labs

Abstract

This paper describes the design and implementation of Egalitarian Paxos (EPaxos), a new distributed consensus algorithm based on Paxos. EPaxos achieves three goals: (1) optimal commit latency in the wide-area when tolerating one and two failures, under realistic conditions; (2) uniform load balancing across all replicas (thus achieving high throughput); and (3) graceful performance degradation when replicas are slow or crash.

Egalitarian Paxos is to our knowledge the first protocol to achieve the previously stated goals efficiently—that is, requiring only a simple majority of replicas to be non-faulty, using a number of messages linear in the number of replicas to choose a command, and committing commands after just one communication round (one round trip) in the common case or after at most two rounds in any case. We prove Egalitarian Paxos's properties theoretically and demonstrate its advantages empirically through an implementation running on Amazon EC2.

1 Introduction

Distributed computing places two main demands on replication protocols: (1) high throughput for replication inside a computing cluster; and (2) low latency for replication across data centers. Today's clusters use fault-tolerant coordination engines such as Chubby [4], Boxwood [22], or ZooKeeper [12] for activities including operation sequencing, coordination, leader election, and resource discovery. Many databases are accessed simultaneously from different continents, requiring geo-replication [2, 8].

An important limitation on these systems is that during efficient, failure-free operation, all clients communicate with a single master (or leader) server at all times.

Copyright is held by the Owner/Author(s).
SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2517350

This optimization—termed "Multi-Paxos" for systems based on the Paxos protocol [16]—is important to achieving high throughput in practical systems [7]. Changing the leader requires invoking additional consensus mechanisms that substantially reduce performance.

This algorithmic limitation has several important consequences. First, it impairs scalability by placing a disproportionately high load on the master, which must process more messages than the other replicas [23]. Second, when performing geo-replication, clients incur additional latency for communicating with a remote master. Third, as we show in this paper, 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 (EPaxos) 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 is committed without interfering with other concurrent commands. This allows the system to evenly distribute the load to all replicas, eliminating the first bottleneck identified above (i.e., having one server that must be on the critical path for all communication). EPaxos's flexible load distribution better handles permanently or transiently slow nodes, as well as the latency heterogeneity caused by geographical distribution of replicas; this substantially reduces both the median and tail commit latency. Finally, the system can provide higher availability and higher performance under failures because there is no transient interruption caused by leader election: there is no leader, and hence, no need for leader election, as long as more than half of the replicas are available.

We begin by reviewing the core Paxos algorithm and the intuition behind Egalitarian Paxos in Section 2. We then describe several Paxos variants that reduce overhead or commit latency in Section 3. Throughout the paper we compare extensively against Multi-Paxos and two recent Paxos derivatives: Mencius [23] and Generalized Paxos [17]. Mencius successfully shares the master load by distributing the master responsibilities round-robin among the replicas. Generalized Paxos introduces the idea that non-conflicting writes can be committed independently in state machine replication to improve commit latency. Our results in Section 7 confirm that Mencius is effective, but only when the nodes are homogeneous: EPaxos achieves higher throughput and better performance stability under a variety of realistic conditions, such as wide-area replication, failures, and nodes that experience performance variability. Generalized Paxos relies on a single master node, and thus suffers from the associated problems.

2 Overview

We begin by briefly describing the classic Paxos algorithm, followed by an overview of Egalitarian Paxos.

2.1 Paxos Background

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 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 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.

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). When a non-leader replica 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.

Section 3 discusses several Paxos variants that improve upon this basic protocol.

2.2 Egalitarian Paxos: 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 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.

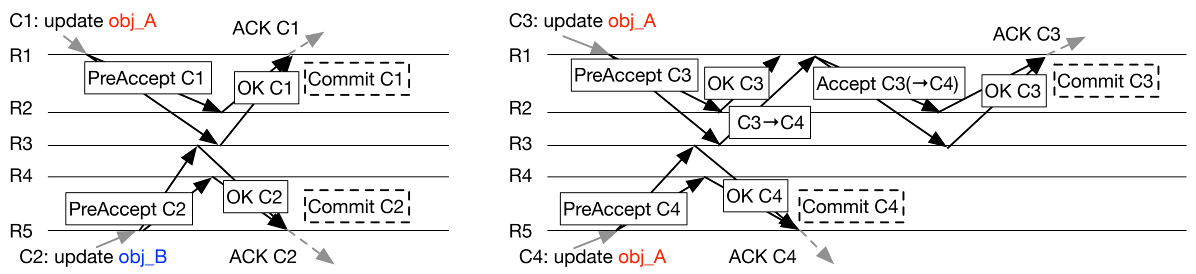


Figure 1: EPaxos message flow. R1, R2, ... R5 are the five replicas. Commands C1 and C2 (left) do not interfere, so both can commit on the fast path. C3 and C4 (right) interfere, so one (C3) will be committed on the slow path. C3 → C4 signifies that C3 acquired a dependency on C4. For clarity, we omit the async commit messages.

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 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 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 1 for command C3). We call this the *slow path*.

3 Comparison with Related Work

Multi-Paxos [15, 16] 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 [4]. 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. Chubby [4] and Boxwood [22] use Multi-Paxos, while ZooKeeper [12] relies on a stable leader protocol similar to Multi-Paxos.

Mencius [23] 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 [18] 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 [17] 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.[1] 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 [5] 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 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 Section 7.2. These advantages make EPaxos a good fit for MDCC [14], 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 datacenter. This distinction allows EPaxos to have smaller fast-path quorums and has the added benefit of not requiring clients to broadcast their proposals to a supermajority of nodes.

In *S-Paxos* [3], 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.

Consistently ordering broadcast messages is equivalent to state machine replication. EPaxos has similarities to generic broadcast algorithms [1, 26, 29], that require a consistent message delivery order only for conflicting messages. Thrifty generic broadcast [1] has the same liveness condition as (E)Paxos, but requires $\Theta(N^2)$ messages for every broadcast message. It relies on atomic broadcast [27] to deliver conflicting messages, which has a latency of four message delays. $\mathcal{GB}$, $\mathcal{GB}$+ [26], and optimistic generic broadcast [29] 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 [6] 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 at most two additional one-way message delays 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.

Eve [13] takes the orthogonal approach of parallelizing command *execution* on multi-core systems.

4 Design

In this section we describe Egalitarian Paxos in detail, state its properties and sketch informal proofs of those properties. Formal proofs and a TLA+ specification of the protocol can be found in a technical report accompanying this paper [25]. We begin by stating assumptions and definitions, and by introducing our notation.

4.1 Preliminaries

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 instances $R.1, R.2, R.3, ...$ that replica R is said to *own*. 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

[1]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.

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 replicated 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 replicated state updated by its 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)*.

Before describing EPaxos in detail, we must define **command 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 equivalent* to Σ, δ, γ (i.e., they result in different machine states and/or different values returned by the reads within these sequences).

4.2 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.

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.[2]

[2]Paxos provides the same liveness guarantees. By FLP [9], it is impossible to provide stronger guarantees for distributed consensus.

4.3 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*[3] is $2F$ (out of the total of $N = 2F + 1$ replicas). The fully optimized EPaxos reduces this quorum to only $F + \lfloor \frac{F+1}{2} \rfloor$ replicas. The slow-path quorum size is always $F + 1$.

4.3.1 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 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) 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:

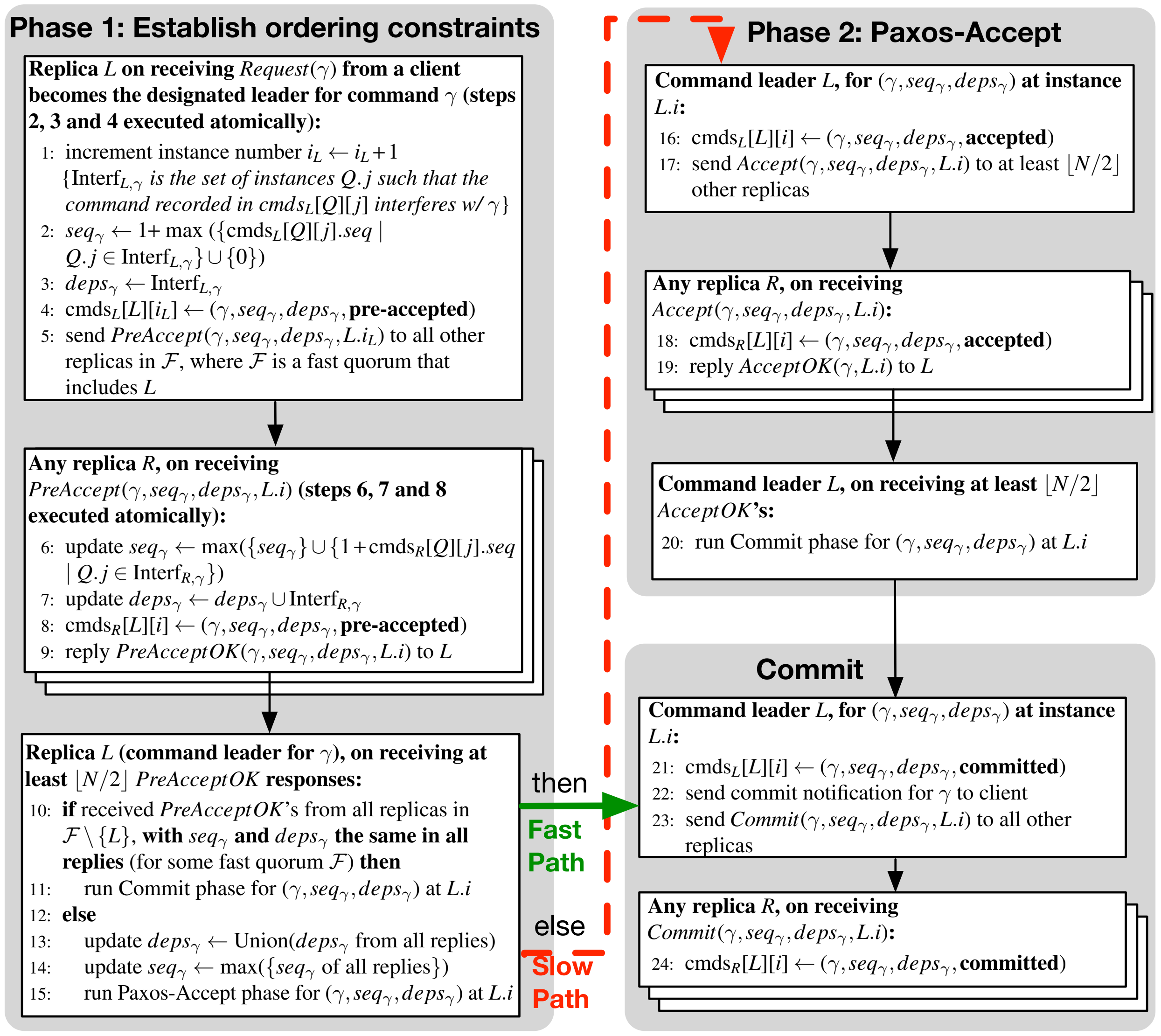
deps is the list of all instances that contain commands (not necessarily committed) that interfere with γ; we say that γ *depends* on those instances and their corresponding commands;

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

[3]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 2: The basic Egalitarian Paxos protocol for choosing commands.

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 $(\gamma, deps_\gamma, seq_\gamma)$ 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). 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 4.7). 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 $R.i$.

4.3.2 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 com-

```
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: The EPaxos simplified recovery procedure.

mands 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.

4.3.3 Informal Proof of Properties

Together, the commit protocol and execution algorithm guarantee the properties stated in Section 4.2. We prove this formally in a technical report [25], 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 4.7). □

Proposition 1 implies **consistency**. Furthermore, because commands can be forgotten only if a replica crashes, this 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_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.

Proposition 2. *Replicas commit only safe tuples.*

Proof sketch. A tuple $(\gamma, seq_\gamma, deps_\gamma)$ 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 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 $(\gamma, seq_\gamma, deps_\gamma)$, 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:

Lemma 1 (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. $\square$

Lemma 2 (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. $\square$

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.

4.4 Optimized Egalitarian Paxos

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 $F + \lfloor \frac{F+1}{2} \rfloor$ replicas, including the command leader. 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).

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 $\lfloor \frac{F+1}{2} \rfloor$ replicas that have pre-accepted a tuple $(\gamma, deps_\gamma, seq_\gamma)$ in the current instance with identical attributes. Upon discovering them, it tries to convince other replicas to pre-accept this tuple by sending *TryPreAccept* messages. A replica receiving a *TryPreAccept* 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 $deps_\gamma$ and does not have γ in its *deps* either, or one that is in $deps_\gamma$ 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 $(\gamma, deps_\gamma, seq_\gamma)$, 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 $deps_\gamma$ 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 associated technical report [25] contains detailed 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 4.2.

Before concluding this subsection, it is important to point out another implication of the new fast-path quorums size. After F failures, there may be as few as $\lfloor \frac{F+1}{2} \rfloor$ 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 $F + \lfloor \frac{F+1}{2} \rfloor - 1$ *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*, 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.

4.5 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.

4.6 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.

4.7 Reconfiguring the Replica Set

Reconfiguring a replicated state machine is an extensive topic [19, 20, 21]. In EPaxos, ordering ballots by their *epoch* prefix enables a solution that resembles Vertical Paxos [19] 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 [7].

4.8 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 [12]; or grant a read lease to the stable leader so that it can respond without committing an operation [7]. 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).

4.9 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 4.5), and sorting the commands by their sequence attribute adds only an $O(\log n)$ factor.

5 Practical Considerations

Command interference. For EPaxos to function efficiently, the implementation must be able to decide whether two commands interfere before executing them (it can, however, conservatively assume interference if uncertain). Although there are many approaches that could work, one that seems likely is to use explicitly-specified dependency keys as in Google's High Replication Datastore [10] and Megastore [2]. Interference can easily be inferred for NoSQL key-value stores where all (or most) operations identify the keys they are targeting. 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 [28]). For other transactions it will be difficult to predict what exact state they will modify, but it is safe to assume they interfere with any other transaction.

Consistency guarantees. EPaxos guarantees per-object linearizability. As shown by Herlihy and Wing [11], linearizability is a local property, meaning that *"a system is linearizable if each individual object is linearizable"*. This only applies to operations that target single objects, or more generally, to operations for which interference is transitive. The equivalent property for multi-object operations is *strict serializability*. If interference is not transitive, EPaxos maintains per-object linearizability, but does not guarantee strict serializability without a simple modification: the commit notification for a command is sent to clients only after every instance in the command's graph of dependencies has been committed (the proof is in the associated tech report [25]). This has the added benefit that it simplifies the protocol: approximate sequence numbers are no longer necessary (commands within the same strongly connected component are sorted by an arbitrary criterion), which makes for a simplified recovery procedure. The drawback of this version of EPaxos is increased perceived latency at the client for operations that conflict with other concurrent operations—for the common case of non-interfering operations the latency remains the same as in canonical EPaxos.

6 Implementation

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

6.1 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 [24].

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.

6.2 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]

7 Evaluation

We evaluated Egalitarian Paxos on Amazon EC2, using large instances[5] for both state machine replicas and clients, running Ubuntu Linux 11.10.

7.1 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 4.8). 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,

[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.

[5]Two 64-bit virtual cores with 2 EC2 Compute Units each and 7.5 GB of memory. The typical RTT in an EC2 cluster is 0.4 ms

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 [4]—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 [4]) 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 [8].

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

7.2 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 datacenters 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 4 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 4 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. 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[6], 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.

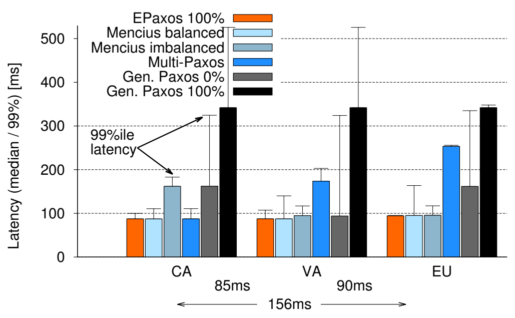
However, if high command interference is common, there is a wide range of techniques that we can use to reduce latency, but we leave for future work: e.g., forwarding PreAccepts among fast quorum members to reduce slow-path commit latency by one message delay, or reverting to a partitioned Multi-Paxos 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).

7.3 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 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.[7]

[6]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).

[7]Learners handle $\Theta(N)$ messages per command and the leader must frequently commit checkpoints—see Section 3.



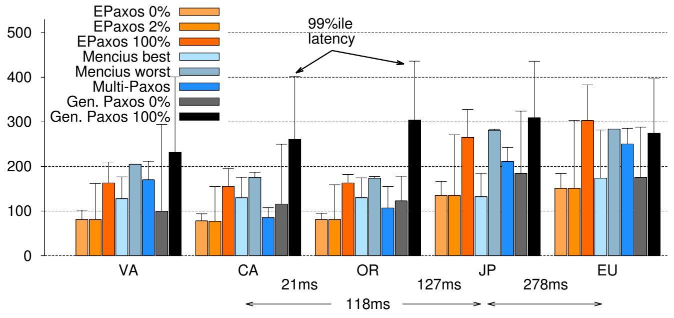


Figure 4: Median commit latency (99%ile indicated by lines atop the bars) at each of 3 (left graph) and 5 (right graph) wide-area replicas. The Multi- and Generalized Paxos leader is in CA. In *Mencius imbalanced*, EU generates commands at half the rate of the other sites (no other protocol is affected by imbalance). In *Mencius worst*, only one site generates commands at a given time. The bottom of the graph shows inter-site RTTs.

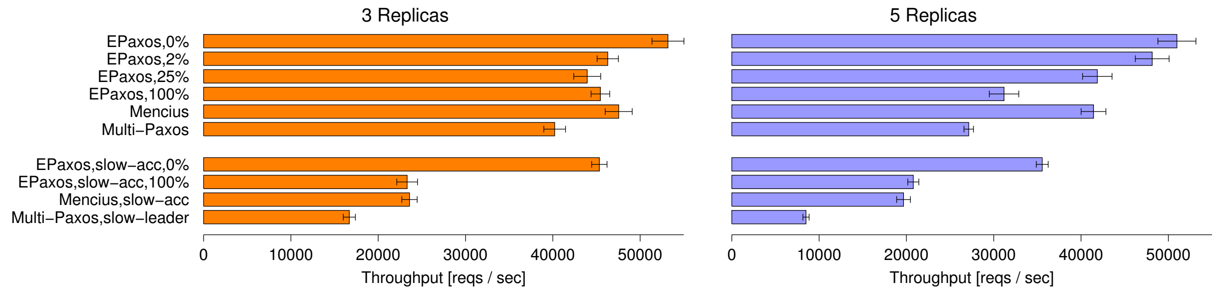


Figure 5: Throughput for small (16 B) commands (error bars show 95% CI).

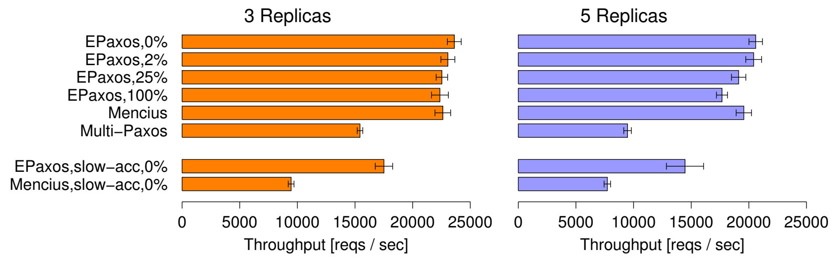


Figure 6: Throughput for large (1 KB) commands (with 95% CI).

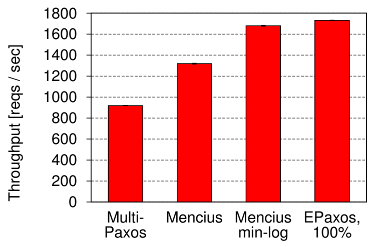


Figure 7: Tput for 3 replicas, 16 B commands, sync. log to Flash (w/ 95% CI).

A client on a separate EC2 instance sends batched requests in an open loop[8] (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 5 shows the throughput achieved by 3 and 5 replicas when the commands are small (16 B). Figure 6 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 6.2), 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.

Figures 5 and 6 also show throughput when one node is slow (for Multi-Paxos that node is the leader—otherwise its throughout is mostly unaffected). In these experi-

[8] 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.

ments, 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.

7.4 Logging Messages Persistently

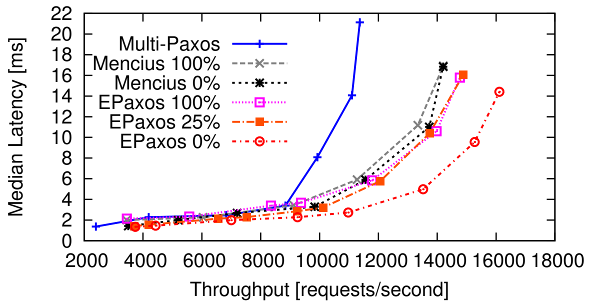
To resume immediately after a crash, a replica must preserve the contents of its memory intact, otherwise it may break safety (for all of the protocols we evaluate). This implies persistently logging every state change before acting upon or replying to any message. The preceding experiments did not include 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 4.7; 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.

Here (Figure 7), 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 7), needs only slightly more synchronous logging than EPaxos (every min-log replica must still log its own skipped instances synchronously).

7.5 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



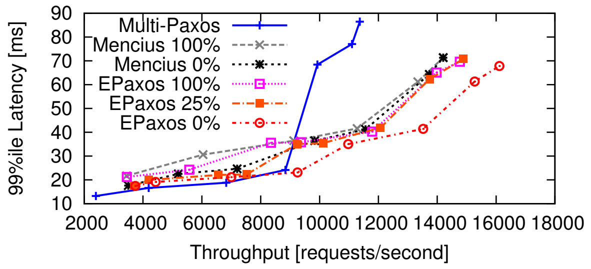


Figure 8: Latency vs. throughput for 3 replicas.

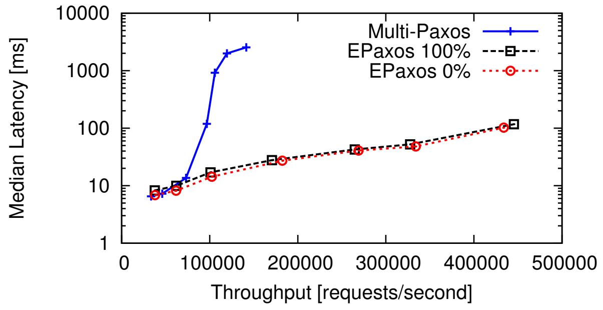


Figure 9: Latency vs. throughput for 5 replicas when batching small (16 B) commands every 5 ms.

EPaxos's execution algorithm (Section 4.9) is effective.

Figure 8 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.

7.6 Batching

Batching increases the maximum throughput of Multi-Paxos by 5x and of EPaxos by 9x (Figure 9). 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

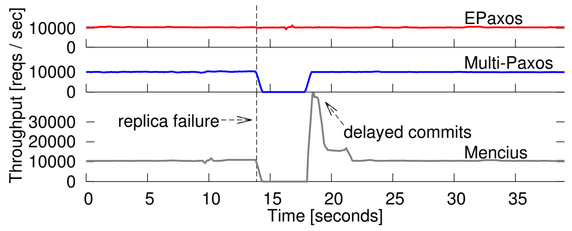


Figure 10: Commit throughput when one of three replicas fails. For Multi-Paxos, the leader fails.

load more evenly across replicas, whereas Multi-Paxos places it all on the stable leader. Under the same client throughput, 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.

7.7 Service Availability under Failures

Figure 10 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 [4, 22]. The failure of a non-leader replica (a situation not depicted in Figure 10) 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 10. 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.

8 Conclusion

We have presented the design and implementation of Egalitarian Paxos, a new state machine replication protocol based on Paxos. We have shown that its decentralized and uncoordinated design has important theoretical and practical benefits for the availability, performance and performance stability of both local and wide area replication.

Acknowledgements: We thank our shepherd Nickolai Zeldovich, the anonymous SOSP and OSDI reviewers; Miguel Castro for his insightful feedback; John Wilkes, Michael Abd-El-Malek and Garth Gibson for helping us understand real-world applications of Paxos; Bin Fan, Hyeontaek Lim, Andy Pavlo, Greg Ganger and Jon Howell for their useful comments. 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 paper.

References

[1] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proc. 14th International Conference on Distributed Computing*, DISC '00, pages 268–282, London, UK, UK, 2000. Springer-Verlag.

[2] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. 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.

[3] 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.

[4] M. Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. 7th USENIX OSDI*, Seattle, WA, Nov. 2006.

[5] L. J. Camargos, R. M. Schmidt, and F. 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.

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

[7] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *Proc. 26th ACM SOSP*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[8] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX OSDI*. USENIX, 2012.

[9] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, Apr. 1985. ISSN 0004-5411.

[10] Google AppEngine. High replication datastore, 2012. `https://developers.google.com/appengine/docs/java/datastore/overview`.

[11] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[12] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association.

[13] 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*, Hollywood, CA, Oct. 2012.

[14] T. Kraska, G. Pang, M. J. Franklin, S. Madden, and A. Fekete. MDCC: Multi-data center consistency. In *Proc. 8th ACM European Conference on Computer Systems (EuroSys)*, Apr. 2013.

[15] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998. ISSN 0734-2071.

[16] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32 (4), Dec. 2001.

[17] L. Lamport. Generalized consensus and Paxos. `http://research.microsoft.com/apps/pubs/default.aspx?id=64631`, 2005.

[18] L. Lamport. Fast Paxos. `http://research.microsoft.com/apps/pubs/default.aspx?id=64624`, 2006.

[19] L. Lamport, D. Malkhi, and L. Zhou. Vertical Paxos and primary-backup replication. Technical report, Microsoft Research, 2009.

[20] L. Lamport, D. Malkhi, and L. Zhou. Reconfiguring a state machine. *SIGACT News*, 41(1), Mar. 2010.

[21] B. Liskov and J. Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT Computer Science and Artificial Intelligence Laboratory, 2012.

[22] J. MacCormick, N. Murphy, M. Najork, C. A. Thekkath, and L. Zhou. Boxwood: abstractions as the foundation for storage infrastructure. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[23] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: building efficient replicated state machines for WANs. In *Proc. 8th USENIX OSDI*, pages 369–384, San Diego, CA, Dec. 2008.

[24] I. Moraru, D. G. Andersen, and M. Kaminsky. Epaxos code base. `https://github.com/efficient/epaxos`, Aug. 2013.

[25] I. Moraru, D. G. Andersen, and M. Kaminsky. A proof of correctness for Egalitarian Paxos. Technical report, Parallel Data Laboratory, Carnegie Mellon University, Aug. 2013. `http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-13-111.pdf`.

[26] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15:97–107, Apr. 2002.

[27] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291: 79–101, Jan. 2003.

[28] tpc-c. TPC benchmark C. `http://www.tpc.org/tpcc/spec/tpcc_current.pdf`, 2010.

[29] P. Zieliński. Optimistic generic broadcast. In *Proc. 19th International Symposium on Distributed Computing (DISC)*, pages 369–383, Kraków, Poland, Sept. 2005.