Egalitarian Paxos

Iulian Moraru[1], David G. Andersen[1], Michael Kaminsky[2]

[1] Carnegie Mellon University, [2] Intel Labs

CMU-PDL-12-108
July 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

This paper describes the design and implementation of Egalitarian Paxos (EPaxos), a new distributed consensus algorithm based on Paxos. EPaxos achieves two goals: (1) availability without interruption *as long as a simple majority of replicas are reachable—its availability is not interrupted when replicas crash or fail to respond; and (2) uniform load balancing across all replicas—no replicas experience higher load because they have special roles. Egalitarian Paxos is to our knowledge the first distributed consensus protocol to achieve both of these goals efficiently: 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.*

Acknowledgements: This work was supported by Intel Science & Technology Center for Cloud Computing, Google, and the National Science Foundation under award CCF-0964474. We would like to thank the members and companies of the PDL Consortium (including Actifio, American Power Conversion, EMC Corporation, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies Co., Intel Corporation, Microsoft Research, NEC Laboratories, NetApp, Inc., Oracle Corporation, Panasas, Riverbed, Samsung Information Systems America, Seagate Technology, STEC, Inc., Symantec Corporation, VMware, Inc., and Western Digital) for their interest, insights, feedback, and support.

Keywords: Distributed Consensus, Paxos

1 Introduction

Today's clusters use fault-tolerant, highly available coordination engines like Chubby [2], Boxwood [13], or ZooKeeper[7] for activities such as operation sequencing, coordination, leader election, and resource discovery. An important limitation in these systems is that during efficient, normal operation, all the 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 [4]. 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 disproportionally high load on the master, which must process more messages than the other replicas [14]. Second, it can harm availability: if the master fails, the system cannot service requests until a new master is elected. Finally, 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. 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 will be 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 (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, substantially reducing both the median and tail commit latency.

To explain Egalitarian Paxos, we begin by reviewing briefly the core Paxos algorithm in Section 3, and several common variants that are designed to provide lower overhead or lower commit latency. In this description, we focus particularly on two relevant derivatives: Mencius [14] and Generalized Paxos [11]. 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 6 confirm that Mencius is effective, but only when the nodes are homogenous. 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.

2 Related Work

The Paxos algorithm for replicated state machines [9, 10] makes efficient forward progress by relying on a stable leader replica who brokers communication with the 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, and practical implementations of Paxos observe this problem [2]. Furthermore, when the leader fails, the state machine becomes temporarily unavailable until a new leader is elected. This problem does not have a simple solution because aggressive leader re-election can lead to false suspicions and stalls if multiple replicas believe they are the new leader. Chubby [2] and Boxwood [13] use Paxos at their core, while ZooKeeper [7] relies on a stable leader protocol similar to Paxos.

Fast Paxos [12] reduces the number of message delays until commands are committed by having clients send commands directly to all replicas. It does not, however, balance load: Some replicas must still act as coordinator and learner nodes, and handle $\Theta(N)$ messages for every command. Like Paxos, Fast Paxos also relies on a stable leader to initiate rounds and arbitrate conflicts (i.e., situations when acceptors order client commands differently, as a consequence of receiving those commands in different orders).

Mencius [14] shares one of our goals: distributing the load evenly across all replicas. It does so by rotating the Paxos leader for every command. The instance space is pre-partitioned among all replicas: replica with id R_{id} is in charge of 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 either reply 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 become even less available than classic 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. In contrast, EPaxos can run uninterrupted as long as more than half the replicas are still alive, and even on the fast commit path, a replica does not need to hear back from all other replicas to commit a command. Finally, Mencius requires FIFO communication channels between replicas, while our algorithm works even when messages can be reordered.

Generalized Paxos [11] previously observed that one can commit 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.[1] Generalized Paxos, however, still relies on a stable leader to order commands that interfere, and learners handle $\Theta(N)$ messages for every command[2]. Multicoordinated Paxos [3] 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. Multicoordinated Paxos still relies on a stable leader to ensure consistent ordering if interfering client commands arrive at coordinators in different orders.

MDCC [8] uses Generalized Paxos to improve commit latency in the wide area. Here too, EPaxos has three advantages over Generalized Paxos: (1) Resolving a conflict (two interfering commands arriving at different acceptors in different orders) requires only one additional round trip in EPaxos, but will take up to two additional round trips in MDCC if the proposal did not originate at the leader's site. (2) For three-site replication the fast path quorum size required to commit commands after only one round trip is 2 vs. 3 for Generalized Paxos. The fast path latency will therefore correspond to a round trip to the site *closest* to the proposer's site, instead of a round trip to the site farthest away. With more than three replicas, the fast path quorum sizes are the same (Section 4.8). (3) Finally, as we explain in Section 4.7, EPaxos can always commit commands after one round trip to the replica closest to the proposer's site for three-site replication, even if all commands conflict. We present the empirical results of this comparison in Section 6.5.

The problem of consistent ordering of broadcast messages is equivalent to state machine replication. In particular, our algorithm has similarities to generic broadcast algorithms [1, 15, 17], where a consistent message delivery order is required only for messages that conflict. Thrifty generic broadcast [1] has the same liveness condition as classic Paxos and EPaxos, but requires $\Theta(N^2)$ messages for every broadcast message, and relies on atomic broadcast [16] to deliver conflicting messages. Other generic broadcast algorithms make stronger assumptions about machine failures: $\mathcal{GB}$, $\mathcal{GB}+$ [15], and optimistic generic broadcast [17] require that more than two thirds of the nodes remain alive. They are also less efficient when handling conflicting commands: $\mathcal{GB}$ and $\mathcal{GB}+$ use sequential instances of Consensus [5] (as many as there are conflicts), while optimistic generic broadcast uses both atomic broadcast (which has a latency of four message delays) and one Consensus instance for every pair of conflicting messages. In contrast, EPaxos requires only two additional message delays (one round trip) to commit commands that interfere,

[1] Egalitarian Paxos can be extended to do the same—see Section 4.8.

[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 round, to balance load if there are no conflicts. Even so, Generalized Paxos would still depend on the leader for availability.

the communication is performed in parallel for all interfering commands, and EPaxos does not rely on a stable leader for deciding the ordering. Moreover, thrifty generic broadcast, $\mathcal{GB}$, and $\mathcal{GB}+$ may see conflicts even if the conflicting messages arrive in the same order at each node.

3 Overview

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

3.1 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. Since 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 the non-faulty replicas will eventually agree[3] 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.

In steady state, clients direct commands to the one replica that has the special *leader* role. For each command, the leader chooses the next available instance, and sends *Accept* messages to at least a simple majority of replicas specifying the instance number and the command; if the *Accept*s are acknowledged by a majority, the leader considers the command committed, notifies the client, and notifies the other replicas asynchronously.

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 committed. To do so, it sends *Prepare* messages to at least a simple majority of replicas. A reply to a *Prepare* contains the command that the replying replica believes may have been chosen in the corresponding instance, and also constitutes a promise not to acknowledge older messages from previous leaders. The Multi-Paxos optimization consists in performing the prepare phase for multiple (possibly infinitely many) instances at the same time. Not doing a separate prepare phase for every instance saves messages and time, reduces the incidence of stalls caused by competing replicas trying to take ownership of the same instances, and results in the same replica being the stable leader over many instances.

We discuss several Paxos variants in Section 2 that improve upon several aspects of this basic protocol.

3.2 Egalitarian Paxos: Intuition

EPaxos is based on two ideas. The first, which has been proposed by generic broadcast algorithms and Generalized Paxos, is that most commands do not interfere with each other, so it is not necessary to enforce a consistent ordering for their execution. The second idea is what distinguishes EPaxos from other state

[3]In fact, termination cannot be strictly guaranteed, given the asynchronous nature of real communication networks and the FLP [6] impossibility result. However, termination can be guaranteed with high probability under realistic assumptions by using timeouts and randomization.

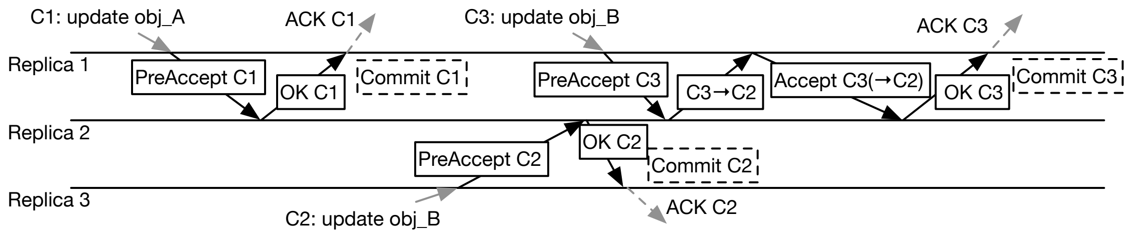

Figure 1: Message flow in Egalitarian Paxos. Commands C2 and C3 interfere (they update the same object) so C3 requires an additional round of communication. C3 $\rightarrow$ C2 signifies that C3 has acquired a dependency on C2. For clarity, we did not illustrate the asynchronous commit messages.

machine replication algorithms: instead of deciding which commands are chosen in which instance of a pre-ordered instance space (as is the case in Multi-Paxos and Mencius), EPaxos determines the ordering of the instances in the process of choosing commands. EPaxos does so by attaching attributes to each command. After a command is committed, all non-faulty replicas will have a consistent view of the attributes for that command, and, based on them, it will execute the command in the same order relative to other interfering commands.

Figure 1 represents a simplified example of how Egalitarian Paxos works. Commands can be proposed by clients at any replica—we call this replica the *command leader* for that command (not to be confused with the stable leader used by Paxos). As long as concurrent proposals do not interfere (the common case for practical workloads), they will be committed after only one round of communication between the command leader and a fast-path quorum of peers. For three total replicas, a fast-path quorum is any two replicas—we explain the concept in detail in Section 4. When commands interfere, they *acquire dependencies* on each other—attributes that commands are committed with, used by replicas to determine the correct order in which to execute the commands (the commit and the execution orders are not necessarily the same). To make sure 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 may be required (as in Figure 1 for command C2).

Using command attributes has two important benefits: (1) all replicas can act as command leaders simultaneously without stalling each other, and (2) replicas that fail to respond do not prevent other replicas from committing commands, as long as a simple majority of replicas are still responsive.

4 Design

In this section we describe Egalitarian Paxos in detail, state its properties and prove them. We begin by stating assumptions, definitions, and 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 replicas. For every replica R there is an unbounded sequence of numbered instances $R.1$, $R.2$, $R.3$, ... that that replica is said to *own*. At most one command will be adopted 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. A client of an EPaxos-based system will interact with the system through an interface of the following form:

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.

To read (a part of) the state, clients send *Read*(*objectID*) messages and wait for *ReadReply*. A *Read* is itself a special no-op command that interferes with updates to the object it is reading—see Section 4.11.

A client that receives a *RequestReply* for a command knows only that the command has been committed, but 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 those commands to be executed.

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

Definition 1. Two commands γ and δ *interfere* if, starting from the same state, executing γ before δ produces different results than executing δ before γ: the resulting states differ and/or the results returned to the clients (if either command is a read) are different.

4.2 Protocol Guarantees

The formal guarantees that Egalitarian Paxos 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 For any two commands γ and δ that interfere, if both γ and δ have been committed by any replicas, then γ and δ 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 A proposed command will eventually be committed by every non-faulty replica, as long as fewer than half the replicas are faulty and messages eventually reach their destination before the recipient times out[4].

4.3 The EPaxos Commit Protocol

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

Figure 2 contains a pseudocode description of the Egalitarian Paxos protocol for choosing commands. The state of each replica is represented in the pseudocode by each replica's private *commands* array.

We split the description of the commit protocol into multiple phases. Not all phases are executed for every command: a command committed after the execution of phases 1, 2 and Commit, is said to have been executed on the *fast path*. The *slow path* involves the additional Multi-Paxos phase. The *Explicit Prepare* phase is only executed on failure recovery.

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

[4]These are the same liveness guarantees provided by Paxos. By FLP [6], it is impossible to provide stronger guarantees for distributed consensus.

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* attributes of all commands in *deps*.

The command leader forwards the command and the initial attributes to at least a *fast quorum* of replicas as a *PreAccept* message. For now, we assume that a fast quorum contains $N-1$ replicas, including the command leader. We will show in Section 4.8 that we can reduce the fast quorum size to $\lceil 3N/4 \rceil$ when $N > 3$.

Each replica, upon receiving the *PreAccept*, updates γ's attributes according to the contents of its *commands* log, records γ and the new attributes in *commands*, and replies to the command leader.

If the command leader receives replies from enough replicas to constitute a fast quorum, and all the updated attributes are the same, it commits the command. If it doesn't receive enough replies, or the attributes in some replies have been updated differently than in others, then the command leader updates the attributes based on $\lfloor N/2 \rfloor + 1$ replies (taking the union of all *deps*, and the highest *seq*), and dictates to at least $\lfloor N/2 \rfloor + 1$ replicas to accept these attributes. This can be seen as running Multi-Paxos for choosing 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 will send *Commit* messages asynchronously to all the other replicas.

Like classic Paxos, every message contains a ballot number (not presented explicitly in the pseudocode for phases other than Explicit Prepare). As in classic Paxos, the ballot number ensures message freshness: a replica will disregard any message with a smaller ballot than the largest it has seen for a certain instance. The initial *Prepare* phase is implicit for all instances $R.i$, for an initial ballot number $0.R$—each replica is the default leader of its own instances.

4.4 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 an explicit prepare phase to force it);
2. Build γ's dependency graph by adding γ and all the commands in instances from γ's dependency list as nodes, with directed edges from γ to these nodes, and then 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 decreasing 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 command in increasing sequence number order (if it hasn't already been executed), and mark it as executed.

4.5 Proof of Properties

We prove that together, the commit protocol and execution algorithm guarantee the properties stated in Section 4.2.

Egalitarian Paxos straightforwardly ensures **nontriviality**: Phase 1 is only executed for commands proposed by clients.

For proving stability and consistency, we first prove the following proposition:

Phase 1

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

```
1: increment instance number $L.i \leftarrow L.i + 1$
2: $seq_\gamma \leftarrow$ max $(\{0\} \cup \{$seq. attribute of every
   command recorded in $commands\}) + 1$
3: $deps_\gamma \leftarrow \{(R, j) \mid commands[R][j]$ interferes w/ $\gamma\}$
4: $commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, PreAccepted)$
5: send $PreAccept(\gamma, seq_\gamma, deps_\gamma, L.i)$ to all other
   replicas
```

Replica R, on receiving $PreAccept(\gamma, seq_\gamma, deps_\gamma, L.i)$ from replica L (steps 6 through 10 executed atomically):

```
6: $max\_seq \leftarrow$ max$(\{0\} \cup \{$seq. attribute of every
   command $\delta$ in $commands$, s.t. $\gamma$ and $\delta$ interfere$\})$
7: update $seq_\gamma \leftarrow$ max$(\{seq_\gamma, max\_seq + 1\})$
8: $deps_{local} \leftarrow \{(R, j) \mid commands[R][j]$ interferes
   with $\gamma\}$
9: update $deps_\gamma \leftarrow deps_\gamma \cup deps_{local}$
10: $commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, PreAccepted)$
11: reply $PreAcceptOK(\gamma, seq_\gamma, deps_\gamma, L.i)$ to $L$
```

Phase 2

Replica L (designated leader for command γ), on receiving at least $\lfloor N/2 \rfloor + 1$ $PreAcceptOK$ responses:

```
12: if received at least $N - 2$ $PreAcceptOK$'s with the
    same $seq_\gamma$ and $deps_\gamma$ attributes then
13:     reply $RequestReply(\gamma, L.i)$ to client
14:     run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
15: else
16:     update $deps_\gamma \leftarrow$ Union($deps_\gamma$ from all replies)
17:     update $seq_\gamma \leftarrow$ max$(\{seq_\gamma$ of all replies$\})$
18:     run Multi-Paxos phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
```

Multi-Paxos

Designated leader replica L, for $(\gamma, seq_\gamma, deps_\gamma)$ at instance $L.i$

```
20: send $Accept(\gamma, seq_\gamma, deps_\gamma, L.i)$ to all replicas
21: if received at least $\lfloor N/2 \rfloor + 1$ $AcceptOK$ in
    response then
22:     reply $RequestReply(\gamma, L.i)$ to client
23:     run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
```

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

```
25: $commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, Accepted)$
26: reply $AcceptOK(\gamma, L.i)$ to $L$
```

Commit

Designated leader replica L, for $(\gamma, seq_\gamma, deps_\gamma)$ at instance $L.i$

```
27: $commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, Committed)$
28: send $Commit(\gamma, seq_\gamma, deps_\gamma, L.i)$ to all other replicas
```

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

```
29: $commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, Committed)$
```

Explicit Prepare Phase

New designated replica Q for command γ, for instance $L.i$ of a potentially failed replica L

```
30: increment ballot number $(b+1).Q$, (where $b.L$ was
    the old ballot number for instance $L.i$)
31: send $Prepare((b+1).Q, L.i)$ to all replicas
    (including self)
32: wait for at least $\lfloor N/2 \rfloor + 1$ responses
33: let $\mathcal{R}$ be set of replies w/ the highest ballot number
34: if $\mathcal{R}$ contains a $(\gamma, seq_\gamma, deps_\gamma, Committed)$ then
35:     run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
36: else if $\mathcal{R}$ contains an $(\gamma, seq_\gamma, deps_\gamma, Accepted)$
    then
37:     run Multi-Paxos phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
38: else if $\mathcal{R}$ contains at least $\lfloor N/2 \rfloor$ identical
    $(\gamma, seq_\gamma, deps_\gamma, PreAccepted)$ replies, and none is
    from $L$ then
39:     run Multi-Paxos phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
40: else if $\mathcal{R}$ contains at least one
    $(\gamma, seq_\gamma, deps_\gamma, PreAccepted)$ then
41:     start Phase 1 for $\gamma$ at instance $L.i$
42: else
43:     start Phase 1 for $\gamma$ at a new instance $Q.j$
```

Replica R, on receiving $Prepare(b.Q, L.i)$ from Q

```
45: if $b.Q$ is larger than the most recent ballot number
    $x.Y$ for instance $L.i$ then
46:     reply $PrepareOK(commands[L][i], x.Y, L.i)$
47: else
48:     reply NACK
```

Figure 2: The Egalitarian Paxos Protocol for Choosing Commands

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 two different commands at the same instance $Q.i$, because (1) Q increments its instance number for every new command, and (2) if Q fails and comes back, it will be assigned a new, previously unseen identifier (Section 4.10). □

Proposition 1 implies **consistency**. Since commands can only be forgotten if a replica crashes, this also implies **stability** if *commands* is maintained in persistent memory. Execution consistency also requires stability and consistency for the command attributes.

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.

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 Multi-Paxos phase, or (2) directly after Phase 2.

Case 1: A tuple is committed after executing the Multi-Paxos phase if more than half of the replicas have logged the tuple as *Accepted* (line 21 in the pseudocode). The tuple is safe via the classic Paxos algorithm guarantees.

Case 2a ($N > 3$): A tuple is committed directly after Phase 2 only if its command leader receives identical responses from at least $N-2$ other replicas (line 12). 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 $(\gamma, seq_\gamma, deps_\gamma)$ which will constitute a majority among the replies, so the new leader will identify this tuple as potentially committed and will use it in the Multi-Paxos phase.

Case 2b ($N = 3$): Compared to the previous case, $\lfloor N/2 \rfloor = 1$ is no longer a majority among the minimum number of replying replicas when a new leader takes over from a potentially failed leader, so, the new leader may not recognize which tuple was committed by the old leader at the end of Phase 2. To solve this problem, we add the constraint that, if $N = 3$, a command is committed on the fast-path only if a replica replies to a *PreAccept* with an unmodified set of attributes. That replica also records that its reply matched the attributes proposed by the leader. Therefore, the new leader will learn that the old leader and another replica have agreed on $(\gamma, seq_\gamma, deps_\gamma)$.

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 have shown that the attributes of a committed command are stable and consistent across all replicas. We now show that they are sufficient to guarantee that all interfering commands are executed in the same order on every replica:

Lemma 1 (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 sketch. If two commands interfere, at least one of them will have the other in its dependency set by the time they are committed: Phase 1 ends after the command has been *PreAccepted* 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 38 in the pseudocode, since all dependencies are based on those set initially by the possibly failed leader). Thus, at least one replica R pre-accepts both γ and δ, and its replies to *PreAccept*s 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 order of their sequence numbers (with an arbitrary criterion to break ties). By Proposition 2 the attributes of all committed commands are stable and consistent across replicas, so all replicas will 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 reversed topological order. By the execution algorithm, γ will always 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, since γ doesn't depend on δ).

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

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. Since 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 set of dependencies. Therefore, when executing δ, δ's graph will 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 that of δ in reversed 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 at least a simple majority to accept it.

4.6 Keeping the Dependency List Small

It is infeasible to include all interfering commands in dependency lists. Instead we include only N dependencies in each list: the instance number $R.i$ with the highest i seen by the current replica. This can be interpreted in two ways: If the communication protocol does not allow messages to be reordered, and the interfering relations are transitive (which is usually the case in practice) the most recent interfering command is sufficient, because its dependency graph will contain all the commands committed in $R.j$ instances, with $j < i$. If these conditions do not hold, every replica will have to assume that any commands in previous $R.j$ ($j < i$) instances are possible dependencies and will have to check all of those that have not been executed—this will not be a long search when commands are executed soon after being committed.

4.7 The Middle Path

As described thus far, Egalitarian Paxos has two main drawbacks: (1) when many commands interfere, the protocol will often fall back to the slow path, which requires one additional round trip time to commit, and increases the total numer of messages processed per command, reducing throughput; and (2) the fast path quorum for $N > 3$ is larger than the classic quorum, so the commit latency may increase—particularly in wide-area deployments where replicas are far apart from each other—because the command leader will have to wait for more replicas to reply.

We can enhance Egalitarian Paxos with an additional mode of operation suitable for situations where these drawbacks manifest. We call this the *middle path*:

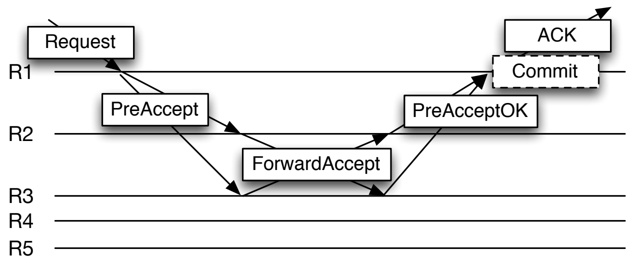

Figure 3: Messages exchanged on the middle path in a five-replica setup. The asynchronous *Commit* messages are not depicted.

1. After receiving a client request, the command leader sends *PreAccept* messages to **only** the replicas in a classic quorum (itself plus $\lfloor N/2 \rfloor$ other replicas);
2. Instead of replying immediately to the *PreAccept*, each non-leader replica in the quorum forwards the *PreAccept* with updated attributes (we will write *ForwardAccept* to denote this message), to every other replica in the quorum except for the command leader;
3. After receiving $\lfloor N/2 \rfloor - 1$ *ForwardAccept*s, each replica will update the current command attributes based on the attributes in these messages and reply to the command leader.

Figure 3 illustrates the messages exchanged on the middle path by a five-replica state machine.

If the replicas in the chosen quorum are responsive and no message is lost, at the end of the three message delays incurred by the middle path every replica in this quorum, including the command leader, will have the same correct attributes for the current command, and the command leader can therefore safely commit—even if all concurrent commands interfere there will be no conflicts, so no need for additional rounds of communication.

If some replicas fail to respond, the command leader will increment the ballot number and revert to the conservative path—i.e., re-send the initial *PreAccept* to everyone, then commit on the fast path if there are no conflicts, or the slow path if there are conflicts.

The middle path has a latency of three message delays for $N > 3$—one more than the fast path, but one less than the slow path—and only two message delays for $N = 3$ (just like the fast path). In the absence of failures, it requires the command leader to contact only the closest replicas in a classic quorum. The middle path uses $\lfloor N/2 \rfloor \times (\lfloor N/2 \rfloor + 1)$ total messages to commit a command, whereas the fast path requires $2(\lceil 3N/4 \rceil - 1)$ messages, and the slow path $2\lfloor N/2 \rfloor$ in addition. The middle path will therefore use just as many messages per command as the fast path for $N \leq 5$, and fewer messages than the slow path for $N \leq 9$.

4.8 Extending Egalitarian Paxos

There are two ways to extend EPaxos, that we do not explore in detail, as they are less practical:

First, we can reduce the fast-path quorum size for $N > 3$ from $N - 1$ to $\lceil 3N/4 \rceil$[5]: the necessary and sufficient condition for a fast-path quorum is that the intersection of a fast-path quorum with a classic quorum represents a majority in the classic quorum. This is so that in case of failures, the recovery process will be able to identify the possible surviving replicas of a fast-path quorum, no matter which classic quorum of replicas reply first. The drawback to implementing this optimization is that recovery becomes more involved: we have to prevent interfering commands from being committed before we commit what we believe may be

[5]Unsurprisingly, this coincides with the fast quorum condition in Generalized Paxos.

a command committed on the fast-path. This optimization only makes a difference for $N > 7$, which is less practical.

Second, we can make EPaxos more like Fast and Generalized Paxos (to decrease the commit latency by one message delay) by letting clients broadcast commands to all replicas. There are two drawbacks: (1) the fast-path quorum sizes for $N = 3$ increases from 2 to 3 (as in Generalized Paxos); and (2) when building $deps$, we can no longer identify commands by their instance numbers—we must use unique identifiers set by the clients instead.

4.9 Recovering from Failures

A replica may need to find out the decision for an instance because it has to execute commands that depend on it. If the replica times out waiting for the commit for that instance, it will try to take ownership of it by running an *Explicit Prepare* phase, at the end of which it will either learn what command was being proposed in the problem instance (in which case it will finalize committing that command), or it will not learn any command (because no other replica has seen it), in which case it will commit a special no-op command to finalize the instance

Another failure-related situation is that where a client timed out waiting for a replica to reply and re-issues its command to a different replica. As a result, the same command can be proposed in two different instances, so every replica must be able to recognize duplicates, and only execute the command once. This situation is not specific to Egalitarian Paxos—it affects any replication protocol. An alternative solution is to make the application tolerant of re-executed commands.

4.10 Joining/Rejoining the Replica Set

New replicas (or replicas that recover after losing the contents of their memory) must be associated a previously unseen replica id. The first action after joining is to send a special *Join* message to at least $\lfloor N/2 \rfloor + 1$ old replicas (replicas that are not themselves in the process of joining). Every reply to the *Join* contains the highest instance numbers $R.i$ (for all replicas R) for which the old replica has received messages. The new replica will only be able to participate in the voting process after seeing commits for all the instances up to and including these instances.

The new replica must receive *Commit* messages for all the commands already chosen before it became live, before fewer than $\lfloor N/2 \rfloor + 1$ old replicas remain—if those commands had been chosen but not explicitly committed, they must be committed.

4.11 Read Leases

To avoid reading stale data, a *Read* must be committed as a command that interferes with all updates to the objects it is reading. However, Paxos-based systems are often optimized for frequent read scenarios, in one of two ways: assume the clients can handle stale data, and perform reads locally at any replica, as in ZooKeeper [7]; or use read leases [4]. Read leases work well for classic Paxos, because the leader is already the only replica through which updates are committed (unless the leader fails). In EPaxos, however, read leases may reduce the load-balancing benefits.

4.12 Avoiding Execution Livelock

With a fast stream of conflicting conflicts, EPaxos could experience execution livelock: any command γ will acquire dependencies on newer commands proposed between sending and receiving the $PreAccept(\gamma)$ message. 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 commit and execution throughputs: the difference between executing n independent commands and executing a batch of n interdependent commands is only an $O(\log n)$ factor: finding the strongly connected components has linear time complexity (the number of dependencies for each command is constant—Section 4.6), and sorting the commands by their sequence attribute will only add an $O(\log n)$ factor.

5 Implementation

We have implemented EPaxos and three other Paxos variants (classic Paxos, Mencius, and Generalized Paxos) in the Go programming language, version go1. We chose Go because it allows for fast prototyping and provides primitives suitable for event-based programming.

Go presented two challenges: its garbage collector affects performance, and, more importantly, its RPC system is slow. We solved the second problem by implementing our own RPC stub generator. We have not yet mitigated the GC penalty, but, because EPaxos uses larger messages (to include the attributes), it is more affected than the other protocols, so our results are fair to the other variants.

The Thrifty Optimization In implementing EPaxos and classic Paxos, 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 classic Paxos) sends *Accept* and *PreAccept* messages to only a quorum of replicas, including itself, not the full set. This reduces the message traffic for each command, and improves throughput. The downside of this optimization is that if an acceptor fails to reply quickly, there is no quick fallback on another reply. To mitigate this, *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 use the *thrifty* optimization because the replies to *Accept* messages contain information about the status of previous instances (whether they were skipped or not) necessary to commit the current instance[6].

6 Evaluation

We evaluate Egalitarian Paxos on Amazon EC2, using large instances[7] for both state machine replicas and clients, running Ubuntu Linux 11.10.

We implemented EPaxos, classic Paxos, Mencius, and Generalized Paxos. Because Generalized Paxos was not designed for high throughput (learners handle $\Theta(N)$ messages for each command) and its availability is tied to that of the leader, as for classic Paxos, we only evaluate Generalized Paxos in the wide area, for which its design is well suited, as observed by MDCC [8].

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 generally treated the same way (though reads can be serviced locally in certain situations, as discussed in Section 4.11). Other message semantics do not influence the behavior of the protocols, with one important exception, which applies to EPaxos, Generalized Paxos, and Mencius, and that our testing workload does capture: conflicts (a proposed-but-not-executed command interferes with the current command). One example of such conflicts are those experienced by a locking service,

[6] More precisely, a Mencius replica must receive *Accept* replies from all the owners of instances it has not received messages for. We implemented Mencius-thrifty, in which *Accept*s are sent first to the replicas the current leader has to hear from, and to other replicas only if a quorum has not yet been reached. It did not improve throughput, however, because under medium and high load, only rarely are all previous instances "filled" at the time a command is proposed.

[7] A large EC2 instance comprises 2 64-bit virtual cores with 2 EC2 Compute Units each and 7.5 GB of memory. The typical RTT in an Amazon EC2 cluster is 0.4 ms

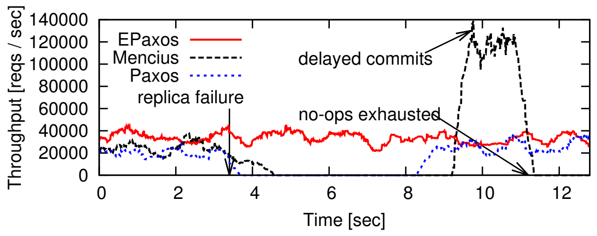

Figure 4: Commit throughput over time for EPaxos, Mencius and classic Paxos when one replica (out of three) fails. For Paxos, the failed replica is the leader.

where conflicts are equivalent to the write-write conflicts triggered by multiple clients updating the same key. A read-heavy workload will be equivalent to updates that rarely target the same key, because reads do not conflict with other reads. Lease renewal traffic, however—which constitutes over 90% of the requests handled by Google's Chubby [2]—generates no conflicts, since only one client can renew a particular lease. For these reasons, although for completeness we present results for 0%, 25% and 100% conflicts (e.g., for the 25% case, $\frac{1}{4}$ of commands target the same key and $\frac{3}{4}$ target different keys), we believe the results corresponding to 0% conflicts to be the most relevant for practical workloads.

6.1 Service Availability under Failures

Figure 4 shows the evolution of the commit throughput in a three-replica setup that experiences the failure of one replica. The client sent requests in an open loop[8] and measured the rate at which it received replies. The throughput variation in all protocols in steady state is caused by garbage collection.

With Multi-Paxos or variant that relies on a stable leader, a leader failure prevents the system from processing client requests until a new leader is elected. While clients could direct their requests to another replica (after they time out), a replica will usually not try to become the new leader immediately, because false suspicions can degrade performance by causing stalls.

The failure of a non-leader replica (a situation not depicted in Figure 4) does not affect the throughput of the system—in fact, it increases slightly, since the leader need no longer send messages to that replica.

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 classic 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 (for 100,000 instances owned by the failed replica, in our experiment, in accordance to the parameters suggested by the authors of Mencius [14]), 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 4. Soon after, the live replicas will reach instances ordered after the last instance for which a no-op has been committed, and will have to repeat the failure recovery process (until the failed replica recovers, or until a reconfiguration).

EPaxos operates uninterrupted by the crash of a minority of replicas. Clients with commands outstanding at the failed replica will time out and retry those requests at another replica. While live replicas will commit commands unhindered, some of these commands may have acquired dependencies on interfering commands the failed replica was processing before failing. Executing them (as opposed to committing them) will therefore be delayed until another replica finalizes committing those commands. 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]In practice, a client needing linearizability must wait for acknowledgements before issuing more commands; the open loop mimics an unbounded number of different clients to measure maximum throughput.

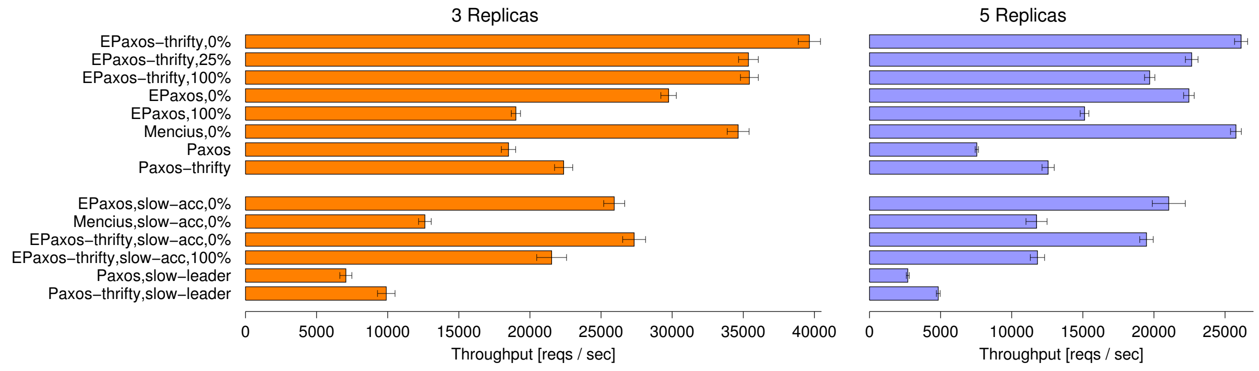

Figure 5: Throughput for small (16 B) commands (the error bars represent 95% confidence intervals). The percentages next to the protocol name represent the percentage of conflicting commands.

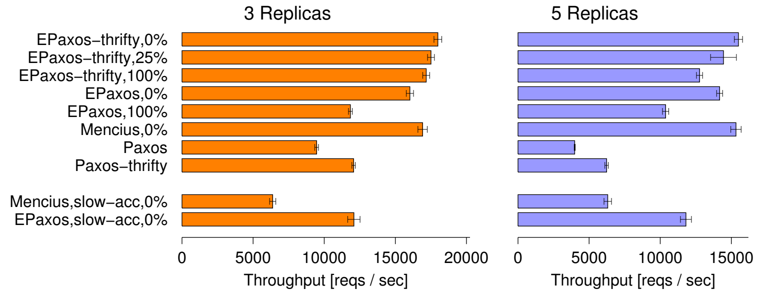

Figure 6: Throughput for large (1 KB) commands (with 95% confidence intervals).

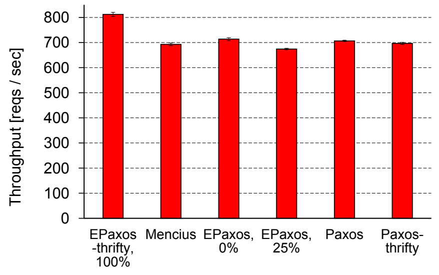

Figure 7: Tput for 3 replicas, 16 B cmds, synch logging to disk (95% c.i.)

6.2 Throughput

We compare the throughput achieved by Egalitarian Paxos, classic (multi) Paxos, and Mencius. A client on a separate EC2 instance sends batched requests in an open loop (only the client requests are batched; messages exchanged by 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. In practice, it is often sufficient to acknowledge after the command has been committed, but because EPaxos has a more complex execution component, we wanted to assess its impact on performance.

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 achieves better throughput than Paxos because the Paxos leader becomes CPU-bottlenecked. Thrifty EPaxos (Section 5) processes fewer messages than Mencius, so its throughput is generally higher—with the exception of conflict situations for more than 3 replicas, when EPaxos executes an extra round per command. Non-thrifty EPaxos processes slightly fewer messages per command than Mencius, because Mencius must sometimes send Skips that cannot be piggybacked on other messages. Because EPaxos messages are slightly larger because of the attributes they carry, our implementation incurs more garbage collection overhead. We plan to address this implementation artifact to mitigate this effect.

Five-replica clusters have lower throughput than three-replica clusters for all protocols: The instance leader must send twice as many *Accept* (and/or *PreAccept*) messages in Mencius, Paxos and non-thrifty

EPaxos, and three times as many *PreAccept*s in thrifty EPaxos[9].

Processing large commands narrows the performance gap between all protocols because all replicas spend more time sending and receiving the command (either from the client or from the leader), but Mencius and EPaxos retain their roughly 2x throughput advantage over Paxos.

Figures 5 and 6 also show the throughput achieved by each protocol when a node is slow (for Paxos that node is the leader—otherwise its throughout does not degrade significantly). In these experiments we created high contention for the CPU on one replica (by running four infinite loop programs). EPaxos is better than both Mencius and Paxos at handling a slow replica because replicas commit commands independently, and, even on the fast commit path, an EPaxos replica only needs to talk to $N-2$ other replicas (where N is the total number of replicas). Mencius, by contrast, runs at the speed of the slowest replica because the instance space is 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.

6.3 Logging Messages to Disk

To immediately resume operation 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). Preservation implies logging every state change to persistent memory before acting upon or replying to any message. The preceeding experiments did not include this overhead, because it is avoidable in some circumstances: if complete power failure of all replicas is not a threat, replicas can recover from failures as presented in Section 4.10; 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 Paxos or Mencius. Figure 7 accordingly presents the throughput achieved when every replica synchronously logs its state changes to disk before replying to protocol messages[10].

Here, all protocols are I/O bound. They perform similarly because all replicas, leaders or not, write to disk at least once for every command to log *Commit* and *Accept* (or *PreAccept*) messages. EPaxos-thrifty outperforms Mencius because it uses fewer messages per command. Since in the thrifty setup some acceptors perform only one write to disk per command (when they receive the Commit for that command), those acceptors have lower load than the leader. By sharing the leader role between all replicas, EPaxos-thrifty runs faster than Paxos-thrifty.

6.4 Execution Algorithm Effect on Latency

In this section we evaluate the impact of interfering commands on execution latency. Figure 8 shows how the median (bottom graph) and 99% latencies vary when increasing the throughput in EPaxos, Mencius and Paxos. In this set of experiments we vary the throughput by varying the number of concurrent clients sending commands in a closed loop (each client sends a command, and then waits for the confirmation that the command has been executed before moving to the next) between 8 and 300. The maximum throughput is lower than in the throughput experiments because replicas no longer handle batched requests on a single TCP connection. Instead, each replica handles hundreds of TCP connections simultaneously, leaving less CPU time for the protocol itself.

While interfering commands increase latency, the strategy that we adopt to avoid livelock in the execution algorithm (as explained in Section 4.12) is effective, as, with three replicas, EPaxos has lower latency than Paxos and Mencius, regardless of the percentage of interfering commands.

For latency too, a slow replica has a higher impact on Mencius than on EPaxos, as evidenced in Figure 9.

[9]The difference is less steep when switching from 5 to 7 replicas—5 *PreAccept*s instead of 3—although 7 replicas is a less practical setup.

[10]We ran these experiments on a private, local cluster because of poor EC2 I/O performance.

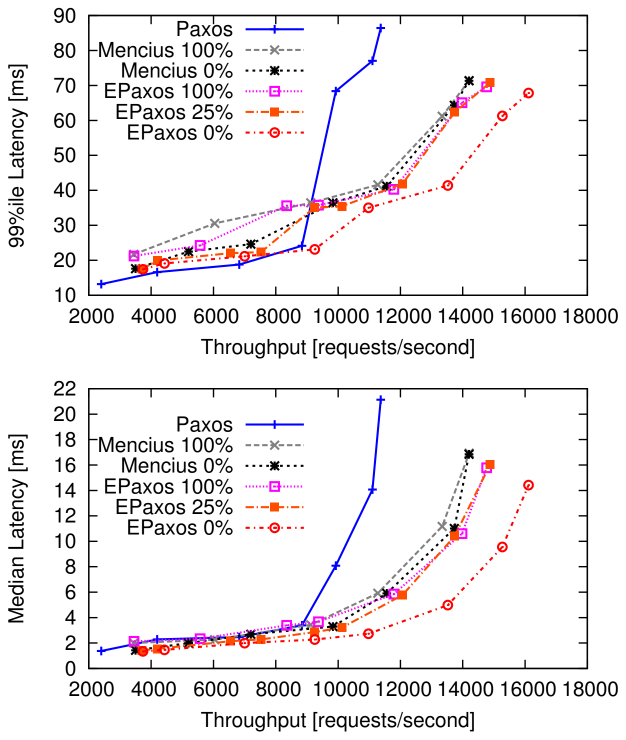

Figure 8: Latency vs. throughput for three replicas. Percentages in the legend refer to the percentage of interfering commands.

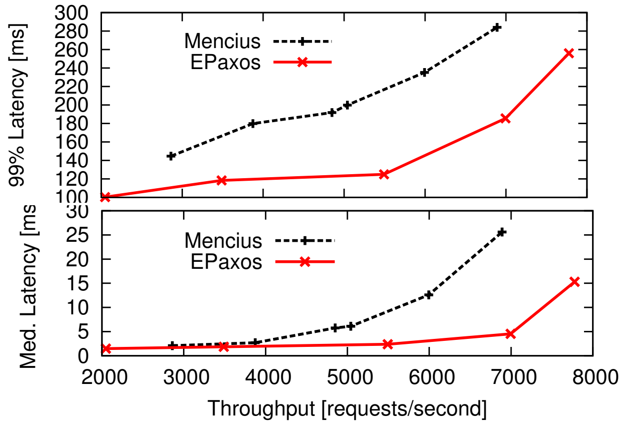

Figure 9: Latency vs. throughput when one replica out of three is slow (i.e., it experiences high CPU contention).

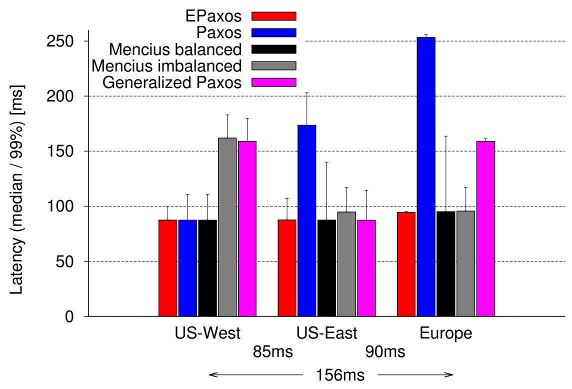

Figure 10: Median commit latency, with 99 percentile error bars, at each of the three wide-area replica sites. For Paxos, the leader is located on the West coast. *Mencius imbalanced* shows the latency when the Europe site generates commands at half the speed of the other sites (no other protocol is affected by this imbalance). The bottom of the graph indicates the inter-site RTTs.

6.5 Latency In Wide Area Replication

To evaluate the benefits of Egalitarian Paxos for wide area replication, we run a three replica experiment with each replica at a different Amazon EC2 datacenter: California, Virginia, and Ireland. At each location there is a client sending requests to its local replica. The clients generate requests simultaneously, and we report the median and 99% commit latencies experienced by each client when using EPaxos, clasic Paxos, Mencius and Generalized Paxos. The results are shown in Figure 10.

In EPaxos, a replica will always be able to commit a command after a round trip to its nearest peer (due to the optimization presented in Section 4.7), even if all commands interfere. With more than three replicas, EPaxos will at least be able to avoid the replica farthest away. Unlike with only three replicas, though, interfering commands will cause an extra round trip to the closest $\lfloor N/2 \rfloor$ replicas. In contrast, the fast quorum size for Generalized Paxos when $N = 3$ is 3, which means that the latency for Generalized Paxos is determined by a round-trip to the farthest replica. Furthermore, conflicts will cause up to two additional round trips for any N, even for just three replicas (one to the leader, and one to the $\lfloor N/2 \rfloor$ closest replicas to the leader). Thus, for the setup in our experiment, EPaxos is not affected by conflicts, but Generalized Paxos experiences median latencies of between 173 ms and 251 ms with 100% conflicts, depending on the location of the leader relative to that of the proposer.

Mencius performs well if the command streams at all locations are matched, because a replica learns about the previous instances it does not own by the time it needs to commit one of its own instances. Imbalances force Mencius to wait for more replies to *Accept* messages. In the worst case, with no proposers at any of the other locations, Mencius will experience latency corresponding to the round trip time to the replica that is farthest away from the proposer.

Classic Paxos has high latency because the local replica cannot initiate the process of choosing a command, having instead to forward it to the leader.

The results in this section refer to commit latency. When only replying to clients after executing commands, EPaxos may be delayed by interering commands being proposed by remote clients: with 100% conflicts EPaxos experiences a median latency of 139 ms at the Europe site, 125 ms at the US-East site, and 130 ms at the US-West site. This, however, is an unrealistically pessimistic scenario because (1) an update (a write) does not have to be executed for the client that generated it to be able to safely move on to other commands, and (2) while, in general, read commands have to be executed before replying to the client, reads generate few conflicts because reads do not conflict with each other.

7 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 benefits for the availability, performance and performance stability of both local and wide area replication.

References

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

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

[3] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. Multicoordinated paxos. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 316–317, New York, NY, USA, 2007. ACM.

[4] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

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

[6] 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.

[7] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association.

[8] Tim Kraska, Gene Pang, Michael J. Franklin, and Samuel Madden. MDCC: Multi-data center consistency. http://arxiv.org/abs/1203.6049, 2012.

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

[10] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), December 2001.

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

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

[13] 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.

[14] 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.

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

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

[17] Piotr Zieliński. Optimistic generic broadcast. In *Proceedings of the 19th International Symposium on Distributed Computing*, pages 369–383, Kraków, Poland, September 2005.