A Proof of Correctness for Egalitarian Paxos

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

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

CMU-PDL-12-109
September 2012

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

Abstract

This paper presents a proof of correctness for Egalitarian Paxos (EPaxos), a new distributed consensus algorithm based on Paxos. EPaxos achieves three 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; (2) uniform load balancing across all replicas—no replicas experience higher load because they have special roles; and (3) optimal commit latency in the wide-area when tolerating one and two failures, under realistic conditions. Egalitarian Paxos is to our knowledge the first distributed consensus protocol to achieve all 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.*

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 [1], Boxwood [5], or ZooKeeper[4] 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 [2]. 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 [6]. 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.

2 Preliminaries

We begin by stating assumptions, definitions, and introducing our notation.

Messages exchanged by processes (clients and replicas) are asynchronous. Failures are non-Byzantine (a machine can fail by stopping to respond for an indefinite amount of time). The replicated state machine comprises N 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.

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.

3 Interfering Commands

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

Informally, two commands that interfere must be executed in the same order by all replicas.

For the sake of clarity, we give a formal definition by restricting the type of commands that clients can propose. This, however, is not fundamental: the notion of interference can be applied similarly to many more types of commands.

Let $\mathcal{O}$ be the finite set of all objects that comprise the replicated state of the replicated state machine, and $\mathcal{V}$ the set of all possible values for the objects in $\mathcal{O}$. At any given time, the replicated state is therefore represented by a well defined function $val : \mathcal{O} \to \mathcal{V}$.

A *command* is a tuple $(\mathcal{O}', w, \mathcal{R})$, where $\mathcal{O}' \subseteq \mathcal{O}$, $\mathcal{R} \subseteq \mathcal{O}$, and $w : \mathcal{O}' \to \mathcal{V}$ is a well-defined function. Together, $\mathcal{O}'$ and w represent the *write* part of the command, while $\mathcal{R}$ represents the *read* part. Commands are executed (or *applied*) atomically and sequentially.

Applying a command $c = (\mathcal{O}_c, w_c, \mathcal{R}_c)$ in a state $val : \mathcal{O} \to \mathcal{V}$ has two effects:

1. It atomically modifies the current state to $val' : \mathcal{O} \to \mathcal{V}$ such that $val'(o) = w(o)$ if $o \in \mathcal{O}_c$, otherwise $val'(o) = val(o)$;

2. It returns the set $\{(o, v) \mid o \in \mathcal{R}_c \text{ and } val'(o) = v\}$ to the client.

We are now ready to formally define the notion of interference:

Definition 1 (Interference)**.** Two commands $\gamma = (\mathcal{O}_\gamma, w_\gamma, \mathcal{R}_\gamma)$ and $\delta = (\mathcal{O}_\delta, w_\delta, \mathcal{R}_\delta)$ are said to *interfere* (we write $\gamma \sim \delta$) if either $\mathcal{O}_\gamma \cap \mathcal{O}_\delta \neq \varnothing$, or $\mathcal{O}_\gamma \cap \mathcal{R}_\delta \neq \varnothing$, or $\mathcal{R}_\gamma \cap \mathcal{O}_\delta \neq \varnothing$.

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

4 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 their recipient times out[1].

5 Simplified Egalitarian Paxos

In this section we describe the basic form of the Egalitarian Paxos protocol. In Section 6 we will show how to modify this protocol to reduce the quorum size.

[1]These are the same liveness guarantees provided by Paxos. By FLP [3], it is impossible to provide stronger guarantees for distributed consensus.

5.1 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.

We present a pseudocode description of the Egalitarian Paxos protocol for choosing commands below. 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:

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* numbers 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 5.3 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 tells 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 $0.R$—a replica R will use only ballot numbers $i.R$ (where the number before the dot takes precedence when ordering ballots), and each replica is the default (i.e., initial) leader of its own instances. Whenever a command leader receives a NACK for one of its messages, indicating that some other replica has used a higher ballot in the same instance, that command leader will fallback on executing Explicit Prepare.

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: $seq_\gamma \leftarrow \max\left(\{0\} \cup \{\text{seq. attribute of every command recorded in } commands \text{ that interferes w/ } \gamma\}\right) + 1$
```

```
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

Replica Q for instance $L.i$ of a potentially failed replica L

```
30: increment ballot number $(b+1).Q$, (where $b.L$ was the default 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 a $(\gamma, seq_\gamma, deps_\gamma, Accepted)$ 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 $L.i$, avoiding the fast path
42: else
43:     start Phase 1 for a no-op at instance $L.i$, avoiding fast path
```

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
```

5.2 The Execution Algorithm

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

1. Wait for $R.i$ to be committed (or run 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.

5.3 Fast Egalitarian Paxos

We can use the Fast Paxos optimization in EPaxos to decrease the commit latency by one message delay by letting clients broadcast commands to all replicas. We do not explore this because of two main drawbacks: (1) the fast-path quorum size will be $\lceil 3N/4 \rceil$ (as in Generalized Paxos), which is by at least one replica larger than that in Optimized Egalitarian Paxos (which we describe in Section 6); 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.

5.4 Recovering from Failures

A replica may need to find out the decision for an instance because it has commands to execute 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.

5.5 Joining/Rejoining the Replica Set

New replicas (or replicas that recover after losing the contents of their memory) must be associated previously unseen replica ids by a reliable configuration service (possibly Paxos-replicated itself). The first action of a replica after joining is to send a *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 dead or alive replicas R) for which the old replica has received messages. The new replica will only 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.

5.6 Proof of Properties

We prove that together, the commit protocol and execution algorithm guarantee the properties stated in Section 4.

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

Proof. For any command that reaches the Commit phase, a replica must have executed Phase 1. Phase 1 is only executed for commands proposed by clients. □

For proving stability and consistency, we first prove a stronger property.

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

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

Proof.

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

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

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

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

PROOF:

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

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

1.2.3 *Q.E.D.*

Cases 1.2.1 and 1.2.2 are exhaustive.

1.3 *Q.E.D.*

Immediately from 1.1 and 1.2.

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

PROOF:

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

PROOF: A replica starts an instance only in Phase 1 of the algorithm and it increments the instance number atomically every time it executes Phase 1. The instance number never decreases. If a replica loses the content of its memory (e.g., after a crash), it will be assigned a previously unused replica id by a safe external configuration service—so the same instance can never be started twice.

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

PROOF:

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

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

2.2.3 *Q.E.D.*

Immediately from 2.2.1 and 2.2.2.

2.3 *Q.E.D*

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

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

PROOF:

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

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

PROOF:

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

3.2 Induction step: if tuple $(\gamma, seq_\gamma, deps_\gamma)$ has been committed in ballot b_1, then the next higher successful ballot $b > b_1$ will commit the same tuple.

PROOF:

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

3.2.1 *Case:* Ballot b_1 is committed directly after Phase 2.

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

If the leader of b_1 is among the replicas that reply to the *Prepare* of ballot b_2, then it must have replied after the end of Phase 2 (otherwise it couldn't have completed the smaller ballot b_1), so it will have committed tuple $(\gamma, seq_\gamma, deps_\gamma)$ by then. The leader of b_2 will then know it is safe to commit the same tuple.

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

3.2.1.1 *Subcase:* $N > 3$

The $\lfloor N/2 \rfloor$ replies with tuple $(\gamma, seq_\gamma, deps_\gamma)$ constitute a majority among the first $\lfloor N/2 \rfloor + 1$ *PrepareReply*'s. The leader of ballot b_2, will therefore be able to identify tuple $(\gamma, seq_\gamma, deps_\gamma)$ as potentially committed, and use it in a Multi-Paxos Phase.

3.2.1.2 *Subcase:* $N = 3$

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

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

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

The tuple $(\gamma, seq_\gamma, deps_\gamma)$ is safe by the guarantees of classic Paxos.

3.2.3 *Q.E.D.*

Cases 2.2.1 and 2.2.2 are exhaustive.

3.3 *Q.E.D.*

The induction is complete.

4 *Q.E.D.*
Immediately from 3.

□

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

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

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

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

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

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

Proof.

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

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

PROOF:

1.1.1 *Case:* c is committed immediately after Phase 2.
$N - 1$ replicas have input their attributes for c.

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

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

1.1.2.2 *Subcase:* The Multi-Paxos phase starts after $\lfloor N/2 \rfloor$ *PrepareReply*'s in the recovery phase, none of which is from the initial command leader.
Then $\lfloor N/2 \rfloor$ replicas, plus the initial command leader ($\lfloor N/2 \rfloor + 1$ replicas in total), have contributed to the set of attributes used for the subsequent Multi-Paxos phase.

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

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

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

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

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

2 *Q.E.D.*
By 1, the final dependency *graphs* of γ and δ are in one of three cases:

2.1 *Case:* γ and δ are both in each other's dependency graph.
Then, by the execution algorithm, their dependency graphs are identical, and, moreover, they are in the same strongly connected component. By the execution algorithm, whenever one command is executed, the other is also executed. Since the execution algorithm is deterministic, and since, by Lemma 1, every replica builds the same dependency graphs for γ and δ, every replica will execute the commands in the same order.

2.2 *Case:* γ is in δ's dependency graph, but δ is not in γ's dependency graph.
The commands are in different strongly connected components in δ's graph, and δ's component is ordered after γ's component in reversed topological order.
We show that γ is executed before δ by every replica:

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

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

2.3 *Case:* δ is in γ's dependency graph, but γ is not in δ's dependency graph.
Just like the previous case, with γ and δ interchanged.

2.4 *Q.E.D.*
The above three cases are exhaustive. In all cases, the commands are executed in the same order by every replica.

□

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

Proof.

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

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

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

PROOF:

2.1.1 *Case*: γ is committed directly after Phase 2.
Then $N-1$ replicas have logged the same sequence number for γ, and this is the sequence number with which γ is committed.

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

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

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

3 *Q.E.D.*
At any replica R, there are two possible cases:

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

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

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

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

3.2.1 *Subcase*: δ is in γ's dependency graph.

Then, by 1, δ and γ are in the same strongly connected component. By the execution algorithm and by 2, γ will be executed before δ.

3.2.2 *Subcase*: δ is not in γ's dependency graph.

Then, by 1, γ is in a different strongly connected component than δ, and γ's component is first in reversed topological order. By the execution algorithm, γ is executed before δ.

3.3 *Q.E.D.*

The above cases are exhaustive. In all cases γ is always executed before δ.

$\square$

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

6 Optimized Egalitarian Paxos

We now describe how Egalitarian Paxos can be enhanced by reducing the fast-path quorum size for increasing its performance: higher throughput and lower latency—including optimal commit latency in the wide area for setups with 3 and 5 replicas.

6.1 Preferred Fast-Path Quorums

We modify the way a command leader behaves in Phase 2 of the algorithm: instead of sending *PreAccept* messages to every replica, it sends *PreAccept*s to only those replicas in a fast-path quorum that includes itself. We call this mode of operation *thrifty*. The fast-path quorum can be static per command leader, or it can change for every new command—depending on inter-replica communication latency and dynamic load assessment.

Using this optimization has the immediate benefit of decreasing the overall number of messages processed by the system for each command, thus increasing the system throughput.

Another, less obvious consequence is that we can decrease the fast-path quorum size from $2F$ to $F + \lfloor \frac{F+1}{2} \rfloor$, where F is the maximum number of failures the system can tolerate (the total number of replicas is therefore $N = 2F + 1$). To achieve this, we make two additional modifications to simplified EPaxos:

1. We modify the fast path condition in Phase 2: the command leader commits a command on the fast path if both of the following conditions are fulfilled:

    - The command leader receives $F + \lfloor \frac{F+1}{2} \rfloor - 1$ *PreAcceptReply*'s with identical *deps* and *seq* attributes, and

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

    The last condition is necessary for ensuring that the *seq* attribute for every command in *deps* is final (it will not change), and will aid in recovering from failures, as explained in the next subsesction.

2. We modify the recovery procedure (i.e., the Explicit Prepare Phase), which we describe in the next subsection.

The rest of the algorithm remains the same as described in the previous section.

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

Another consequence is that for 3 replicas, there is no chance of conflicts, even when all commands interfere. This is because the only reply that the command leader waits for the sole *PreAccept* it send does not have another *PreAcceptReply* to conflict with. As long as there are no failures and replicas reply timely, a 3-replica thrifty EPaxos state machine will commit every command after just one round of communication.

6.2 Failure Recovery in Optimized Egalitarian Paxos

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

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

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

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

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

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

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

4. **If** at least one replica has accepted command $(\gamma, deps_\gamma, seq_\gamma)$ in $Q.i$, R exits recovery and starts a Multi-Paxos Phase for this tuple at $Q.i$.

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

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

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

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

   - $\gamma \sim \delta$, and
   - $\delta \notin deps_\gamma$ or $\delta \in deps_\gamma$ but $seq_\delta \geq seq_\gamma$, and
   - $\gamma \notin deps_\delta$.

Otherwise, if such a command δ exists, the receiver of the *TentativePreAccept* replies with NACK and the identity of the command leader that has sent δ (just one of them, if there are multiple such commands), and the status of δ (pre-accepted, accepted or committed).

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

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

   Else if a *TentativePreAccept* NACK returns a command leader that must have been part of γ's fast quorum for γ to have been committed on the fast path, R exits recovery and starts the process of choosing γ at $Q.i$, on the slow path.

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

This decision process is depicted in Figure 1.

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

Lemma 2. *The recovery procedure for Thrifty Egalitarian Paxos cannot deadlock.*

Proof.

For the recovery procedure to deadlock, there must be a command γ for which the recovery procedure always defers.

Assume this is the case. Then there exists a command δ for which the recovery procedure defers the recovery of γ, $\gamma \sim \delta$, and γ has attributes incompatible with those of δ at at least one replica, and the recovery of δ also defers to γ or some other command (otherwise δ would be decided, and so, eventually, would be γ).

Let R be a replica trying to recover γ. Then R must believe that γ may have been committed on the fast path. Eventually, R will defer γ and try to decide δ, and, by our assumption, it must believe that δ too may have been committed on the fast path. Then R must be aware of the following sets and their properties:

1. $RESP_\gamma$, the set of all the replicas in γ's quorum ($QUOR_\gamma$) that have responded to R's prepare messages;
2. $RESP_\delta$, the set of all the replicas in δ's quorum ($QUOR_\delta$) that have responded to R's prepare messages;
3. $|RESP_\gamma| \geq \lfloor \frac{F+1}{2} \rfloor$;
4. $|RESP_\delta| \geq \lfloor \frac{F+1}{2} \rfloor$;
5. $RESP_\gamma \cap RESP_\delta = \varnothing$ (because a replica cannot pre-accept both commands with conflicting attributes);
6. R must know that the possibly failed command leader of δ, $L_\delta \notin QUOR_\gamma$—otherwise it would infer that γ could not have been committed on the fast path, since L_δ would not have pre-accepted it.
7. Since there are at most F replicas that do not reply to R, and L_γ (the possibly failed command leader for γ) must be one of them (otherwise R could decide γ), there are at most $F-1$ replicas that may be part of $QUOR_\delta$ (we denote this superset by $\overline{QUOR_\delta}$) and that R does not receive replies from. Then, for R to believe δ may have been committed on the fast path, it must be the case that $|RESP_\delta| \geq \lfloor \frac{F+1}{2} \rfloor + 1$

By 5 and 6, R must infer that the following sets are disjoint: $\overline{QUOR_\gamma}$ (i.e., the set of replicas that may be part of $QUOR_\gamma$), $RESP_\delta$, and $\{L_\delta\}$. By 7 and our fast-path quorum requirement, the cardinality of the union of these sets must be at least $F + \lfloor \frac{F+1}{2} \rfloor + \lfloor \frac{F+1}{2} \rfloor + 1 + 1 > 2F+1$. But this is impossible, because

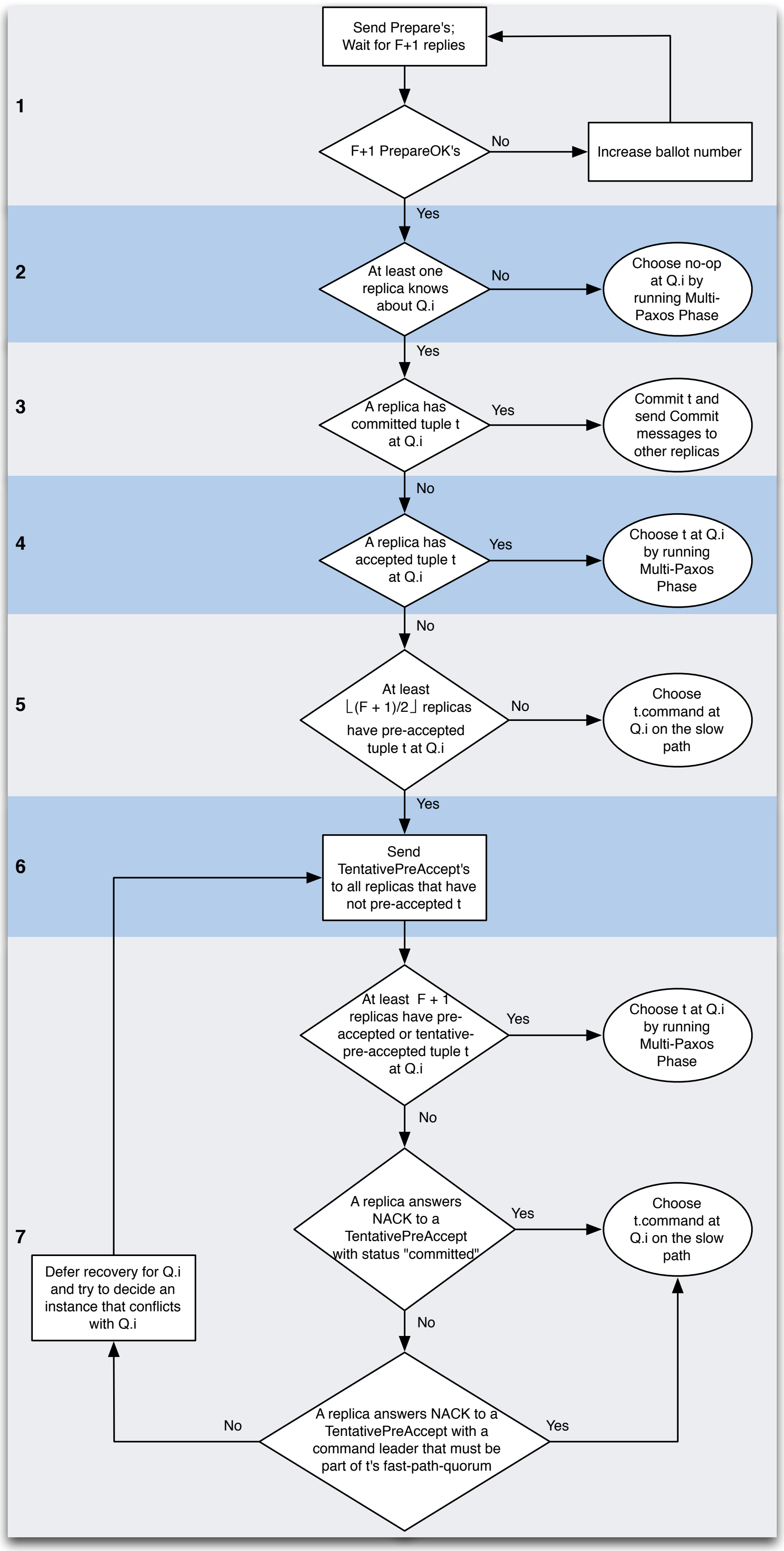
Send Prepare's; Wait for F+1 replies
1
F+1 PrepareOK's
No
Increase ballot number
Yes
2
At least one replica knows about Q.i
No
Choose no-op at Q.i by running Multi-Paxos Phase
Yes
3
A replica has committed tuple t at Q.i
Yes
Commit t and send Commit messages to other replicas
No
4
A replica has accepted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

Figure 1: Decision process for recovery in optimized EPaxos.

this union must be a subset of the replica set, and its cardinality is $2F+1$. Therefore, some of these sets overlap, so R cannot be simultaneously uncertain about γ and δ. Our assumption that the recovery procedure could deadlock is false.

□

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

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

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

1 *Case:* No tuple is committed at instance $Q.i$ before the recovery procedure commits a tuple at $Q.i$.
In all cases, the recovery procedure ends by choosing a tuple on the slow path, by running classic Paxos. The tuple is thus safe by the classic Paxos guarantees.

2 *Case:* A tuple $(\gamma, deps_\gamma, seq_\gamma)$ is committed at $Q.i$ before the recovery procedure terminates.

2.1 *Subcase:* $(\gamma, deps_\gamma, seq_\gamma)$ has previously been committed on the slow path.

Then there must be at least $F+1$ replicas that have accepted $(\gamma, deps_\gamma, seq_\gamma)$. Since the recovery procedure terminates by running classic Paxos in all cases, it will use the same tuple in a Multi-Paxos Phase. By the guarantees of the classic Paxos algorithm, only this tuple can ever be committed at $Q.i$.

2.2 *Subcase:* $(\gamma, deps_\gamma, seq_\gamma)$ has previously been committed on the fast path.

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

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

2.2.2 No interfering command $\delta \sim \gamma$, $\delta \in deps_\gamma$, can be committed such that $\gamma \notin deps_\delta$ and $seq_\delta \geq seq_\gamma$.
PROOF:
We prove this by generalized induction. The relation that we run the induction on is $a \prec b \equiv$ "command a has been committed (in a particular instance) by the recovery procedure for the first time before command b has been committed (in a particular instance) by the recovery procedure for the first time".

2.2.2.1 *Base case:* Let γ_0 be the first command initially committed on the fast path and then committed again as a result of the recovery procedure (or one of the first, if multiple such commands are committed at the exact same time).
Assume there existed $\delta \sim \gamma_0$, $\delta \in deps_{\gamma_0}$, committed such that $\gamma_0 \notin deps_\delta$ and $seq_\delta \geq seq_{\gamma_0}$ at the time of γ_0's recovery. Since γ_0 had been committed on the fast path, then by the *additional condition for the fast-path in optimized EPaxos*, all its dependencies, including

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

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

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

2.2.3 *Q.E.D*
By the recovery procedure, 2.2.1, 2.2.2 and Lemma 2 the recovery procedure will be successful in getting F replicas to pre-accept tuple $(\gamma, deps_\gamma, seq_\gamma)$ (not counting the implicit pre-accept of the initial command leader), and it will start the Multi-Paxos Phase for this tuple.

2.3 *Q.E.D*
Subcases 2.1 and 2.2 are exhaustive and safety is preserved in both.

3 *Q.E.D.*
Cases 1 and 2 are exhaustive and safety is preserved in both.

$\square$

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

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

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

1 *Case:* Both γ and δ have first been committed by their respective command leaders, without running the recovery procedure.
This is no different from simplified EPaxos: the different fast-path condition influences only the recovery path. By Theorem 6 and Theorem 4, γ and δ will be executed in the same order by every replica.

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

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

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

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

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

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

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

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

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

In conclusion $\gamma \in deps_\delta$

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

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

3 *Case:* δ is first committed as a result of the recovery procedure, while γ is first committed by its initial command leader without running the recovery procedure.
Just like case 2, with γ and δ interchanged.

4 *Case:* Both γ and δ are first committed after the recovery procedure.
If at least one of the commands is committed before step 7 in the recovery procedure, or afte exiting step 7 on one of the Else branches, the situation is reducible to one of the previous cases.
The only remaining subcase is that when both commands are committed after exiting step 7 on the If branch. Assume no command has the other in its dependency list when exiting step 7 of the recovery procedure. But each command has been pre-accepted by a majority or replicas (either as a result of *PreAccept*s or *TentativePreAccept*s). Then there must be at least one replica R that pre-accepts both commands, and whose replies are taken into account when establishing each command's commit attributes. If R pre-accepts γ before δ, then, by the conditions in step 6 of the recovery procedure, R will not acknowledge δ without a dependency for γ (and vice-versa). This contradicts our assumption.

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

5 *Q.E.D*
Cases 1, 2, 3 and 4 are exhaustive.

□

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

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

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

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

1 *Case: R* receives a *PreAccept* for δ.
Then R will put γ in $deps_\delta$ and it will increment seq_δ to be larger than seq_γ. Since R's reply is considered when deciding δ's final attributes, δ's dependency list will include γ and its sequence number will be larger than γ's at commit time. By the execution algorithm, γ will always be executed before δ.

2 *Case: R* receives a *TentaticePreAccept* for δ.
Since R will ACK (otherwise δ would not be committed), and $\delta \notin deps_\gamma$ (since δ was proposed after γ was committed), by the conditions in step 6 of the recovery procedure, it must hold that $\gamma \in deps_\delta$ and $seq_\gamma < seq_\delta$. By the execution algorithm, γ will always be executed before δ.

3 *Q.E.D*
Cases 1 and 2 are exhaustive.

□

7 Conclusion

We have presented a proof of correctness for Egalitarian Paxos, a new state machine replication protocol based on Paxos. EPaxos's decentralized and uncoordinated design, as well as its small fast-path quorum size, have important benefits for the availability, performance and performance stability of both local and wide area replication.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.