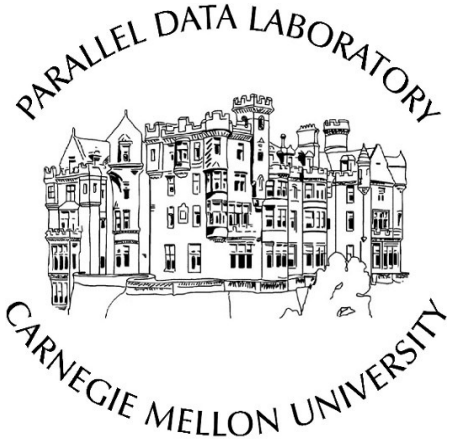


# A Proof of Correctness for Egalitarian Paxos

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

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

CMU-PDL-13-111
August 2013

**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 [7], or ZooKeeper[5] 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 [8]. 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.

**Definition 1** (Interference)**.** Two commands $\gamma$ and $\delta$ *interfere* if there exists a sequence of commands $\Sigma$ such that the serial execution $\Sigma, \gamma, \delta$ is not compatible (i.e., it produces different results than) the serial execution $\Sigma, \delta, \gamma$.

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

Note that the interference relation is symmetric, 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 $\gamma$ committed at some instance $Q.i$, then $R$ will have $\gamma$ 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 $\gamma$ and $\delta$ that interfere, if both $\gamma$ and $\delta$ have been committed by any replicas, then $\gamma$ and $\delta$ will be executed in the same order by every replica.

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

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

# 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.

## 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 Init, Phase 1, and Commit, is said to have

| | |
|---|---|
| $L$, $R$, $Q$ | Replicas (the command leader is usually denoted by $L$) |
| $\gamma$, $\delta$ | Commands |
| $R.i$, with $i = 1, 2, ...$ | Instances belonging to replica $R$ |
| $epoch.i.R$, with $epoch, i = 0, 1, 2, ...$ | Ballot numbers generated by replica $R$ ($epoch.0.R$ is the initial ballot for any instance $R.i$) |
| $deps_\gamma$ | The list of dependencies for command $\gamma$ |
| $seq_\gamma$ | Approximate sequence number for command $\gamma$ |
| $Q.commands[R][i]$ | The state of replica $Q$ at instance $R.i$ |

**Figure 1: Summary of notation.**

been executed on the *fast path*. The *slow path* involves the additional Paxos-Accept phase. The *Explicit Prepare* phase is only executed on failure recovery.

*Phase 1* starts when a replica $L$ receives a request (for a command $\gamma$) and becomes a command leader. $L$ begins the process of choosing $\gamma$ 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 $\gamma$; we say that $\gamma$ *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 5.7 that we can reduce the fast quorum size to $\lceil 3N/4 \rceil$ when $N > 3$.

Each replica, upon receiving the $PreAccept$, updates $\gamma$'s attributes according to the contents of its $commands$ log, records $\gamma$ 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 Paxos-Accept for choosing the triplet $(\gamma, deps_\gamma, seq_\gamma)$ in $\gamma$'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. Ballot numbers used by a replica $R$ have the form $epoch.i.R$, where *epoch* is the natural number id of the current configuration and $i$ is a natural number (*epoch* and $i$ take precedence when ordering ballots, while the replica id $R$ ensures different replicas cannot generate the same ballot number). The initial $Prepare$ phase is implicit for all instances $R.i$, for an initial ballot number $epoch.0.R$ 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.

*Init*

**Replica** $L$ **on receiving** $Request(\gamma)$ **from a client:**

1: start Phase 1 for $\gamma$ at previously unused instance $L.i$

*Phase 1*

**Replica $L$ designated as leader for command $\gamma$ (steps 2, 3 and 4 executed atomically):**

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

**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 $R :: commands$, s.t. $\gamma$ and $\delta$ interfere$\})$

7: update $seq_\gamma \leftarrow$ max$(\{seq_\gamma, max_seq + 1\})$
8: $deps_{local} \leftarrow \{(Q, j) \mid R :: commands[Q][j]$ interferes with $\gamma\}$
9: update $deps_\gamma \leftarrow deps_\gamma \cup deps_{local}$
10: $R :: commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, PreAccepted)$
11: reply $PreAcceptOK(\gamma, seq_\gamma, deps_\gamma, L.i)$ to $L$

**Replica $L$ (designated leader for command $\gamma$), 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 Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$

*Paxos-Accept*

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

19: $L :: commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, Accepted)$
20: send $Accept(\gamma, seq_\gamma, deps_\gamma, L.i)$ to all other replicas in a slow quorum that includes $L$

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

21: $R :: commands[L][i] \leftarrow (\gamma, seq_\gamma, deps_\gamma, Accepted)$
22: reply $AcceptOK(\gamma, L.i)$ to $L$

**Designated leader replica $L$, on receiving at least $\lfloor N/2 \rfloor$ $AcceptOK$ messages for instance $L.i$:**

23: reply $RequestReply(\gamma, L.i)$ to client
24: run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$

*Commit*

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

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

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

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

*Explicit Prepare Phase*

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

28: increment ballot number to $epoch.(b+1).Q$, (where $epoch.b.L$ was the default ballot number for instance $L.i$)
29: send $Prepare(epoch.(b+1).Q, L.i)$ to all replicas (including self)
30: wait for at least $\lfloor N/2 \rfloor + 1$ responses
31: let $\mathcal{R}$ be set of replies w/ the highest ballot number
32: **if** $\mathcal{R}$ contains a $(\gamma, seq_\gamma, deps_\gamma, Committed)$ **then**
33: run Commit phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
34: **else if** $\mathcal{R}$ contains a $(\gamma, seq_\gamma, deps_\gamma, Accepted)$ **then**
35: run Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
36: **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**
37: run Paxos-Accept phase for $(\gamma, seq_\gamma, deps_\gamma)$ at $L.i$
38: **else if** $\mathcal{R}$ contains at least one $(\gamma, seq_\gamma, deps_\gamma, PreAccepted)$ **then**
39: start Phase 1 for $\gamma$ at instance $L.i$, avoiding the fast path
40: **else**
41: start Phase 1 for a *no-op* at instance $L.i$, avoiding fast path

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

42: **if** $epoch.b.Q$ is larger than the most recent ballot number $epoch.x.Y$ for instance $L.i$ **then**
43: reply $PrepareOK(R :: commands[L][i], epoch.x.Y, L.i)$
44: **else**
45: reply NACK

## 5.2 The Execution Algorithm

To execute command $\gamma$ 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 $\gamma$'s dependency graph by adding $\gamma$ and all the commands in instances from $\gamma$'s dependency list as nodes, with directed edges from $\gamma$ to these nodes, and then repeating this process recursively for all of $\gamma$'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 Keeping the Dependency List Small

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

### 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

We take an approach similar to a Vertical Paxos [6] system with majority read quorums: A new replica, or one that recovers without its memory, must receive a new ID and a new (higher) *epoch* number, e.g., from a configuration service or a human. It then sends $Join$ messages to at least $F + 1$ live replicas (that are not themselves in the process of joining). Upon receiving a *Join*, a replica will first update its membership information and then the *epoch* part of each ballot number it uses or expects to receive for new instances—it will thus no longer acknowledge messages for instances initiated in older epochs, that it is not already aware of. It will then send the joining replica the list of committed or ongoing instances that the live replica is aware of. The joining replica becomes live only after receiving commits for all instances included in the replies to at least $F + 1$ *Join* messages. This strategy ensures that a joining replica and a replica that has been excluded from the new configuration cannot participate in voting commands at the same time—thus preserving the property that any two quorums overlap.

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

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

### 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 the Init phase. Init is only executed for commands proposed by clients. □

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

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

*Proof.*

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

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

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

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

PROOF:

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

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

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

1.3 *Q.E.D.*

Immediately from 1.1 and 1.2.

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

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

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

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

PROOF:

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

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

2.2.3 *Q.E.D.*

Immediately from 2.2.1 and 2.2.2.

2.3 *Q.E.D*

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

3 Let $b_{smallest}$ be the smallest ballot number for which a tuple $(\gamma, seq_\gamma, deps_\gamma)$ 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 Paxos-Accept Phase.

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

Since $b_1$ is successful after Phase 1, then a fast quorum ($N - 1$ replicas) have recorded the same tuple $(\gamma, seq_\gamma, deps_\gamma)$ 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 1 (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 Paxos-Accept Phase.

3.2.1.2 *Subcase:* $N = 3$

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

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

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

The tuple $(\gamma, seq_\gamma, deps_\gamma)$ 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 $\gamma$ committed at some instance $Q.i$, then $R$ will have $\gamma$ 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 $\gamma$ and $\delta$ are successfully committed (not necessarily by the same replica), they will be executed in the same order by every replica.*

*Proof.*

1 If $\gamma$ and $\delta$ are successfully committed and $\gamma \sim \delta$, then either $\gamma$ has $\delta$ in its dependency list when $\gamma$ is committed (more precisely, $\gamma$ has $\delta$'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 $\delta$ has $\gamma$ in its dependency list when $\delta$ is committed.
PROOF:

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

PROOF:

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

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

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

1.1.2.2 *Subcase:* The Paxos-Accept phase starts after $\lfloor N/2 \rfloor$ $PrepareReply$'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 Paxos-Accept phase.

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

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

1.1.3 *Case:* $\gamma$ is committed after the current replica receives a $Commit$ for $\gamma$ 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 $\gamma$'s and $\delta$'s final attributes. Because $R$ records every command that it sees in its command log, and because $\gamma \sim \delta$, $R$ will include the command it sees first in the dependency list of the command it sees second.

2 *Q.E.D.*
By 1, the final dependency *graphs* of $\gamma$ and $\delta$ are in one of three cases:

2.1 *Case:* $\gamma$ and $\delta$ are both in each other's dependency graph.

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

2.2 *Case:* $\gamma$ is in $\delta$'s dependency graph, but $\delta$ is not in $\gamma$'s dependency graph.

The commands are in different strongly connected components in $\delta$'s graph, and $\delta$'s component is ordered after $\gamma$'s component in reversed topological order.

We show that $\gamma$ is executed before $\delta$ by every replica:

2.2.1 *Subcase:* A replica tries to execute $\gamma$ first.

The replica will execute $\gamma$ without having executed $\delta$.

2.2.2 *Subcase:* A replica tries to execute $\delta$ first.

By the execution algorithm, the replica will build $\delta$'s dependency graph, which also contains $\gamma$ in a strongly connected component that is ordered before $\delta$'s component in reversed topological order. Then $\gamma$ is executed before $\delta$ is executed.

2.3 *Case:* $\delta$ is in $\gamma$'s dependency graph, but $\gamma$ is not in $\delta$'s dependency graph.

Just like the previous case, with $\gamma$ and $\delta$ 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 $\gamma$ and $\delta$ are serialized by clients (i.e., $\delta$ is proposed only after $\gamma$ is committed by any replica), then every replica will execute $\gamma$ before $\delta$.*

*Proof.*

1 $\gamma$ will be in $\delta$'s dependency graph.
PROOF:
By the time $\delta$ is proposed, $\gamma$ will have been pre-accepted by at least $\lfloor N/2 \rfloor + 1$ replicas. For $\delta$ 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 $\delta$'s dependency list pre-accepts $\delta$ after it has pre-accepted $\gamma$. Since $\gamma \sim \delta$, $R$ will put $\gamma$ in $\delta$'s dependency list.

2 The sequence number with which $\delta$ is committed will be higher than that with which $\gamma$ is committed.
PROOF:

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

PROOF:

2.1.1 *Case*: $\gamma$ is committed directly after Phase 1.

Then $N - 1$ replicas have logged the same sequence number for $\gamma$, and this is the sequence number with which $\gamma$ is committed.

2.1.2 *Case*: $\gamma$ is committed after the Paxos-Accept Phase.

Then at least $\lfloor N/2 \rfloor + 1$ replicas have logged $\gamma$ 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 $\delta$, whose $PreAcceptReply$ is taken into account when establishing $\delta$'s final attributes, will update $\delta$'s sequence number to be higher than $\gamma$'s final sequence number.

3 *Q.E.D.*

At any replica $R$, there are two possible cases:

3.1 *Case*: $R$ tries to execute $\gamma$ before it tries to execute $\delta$.

3.1.1 *Subcase*: $\delta$ is in $\gamma$'s dependency graph.

Then, by 1, $\delta$ and $\gamma$ are in the same strongly connected component. By the execution algorithm and by 2, $\gamma$ will be executed before $\delta$.

3.1.2 *Subcase*: $\delta$ is not in $\gamma$'s dependency graph.

Then, by the execution algorithm, $\gamma$ will be executed (at a moment when $\delta$ won't have been executed).

3.2 *Case*: $R$ tries to execute $\delta$ before it tries to execute $\gamma$.

3.2.1 *Subcase*: $\delta$ is in $\gamma$'s dependency graph.

Then, by 1, $\delta$ and $\gamma$ are in the same strongly connected component. By the execution algorithm and by 2, $\gamma$ will be executed before $\delta$.

3.2.2 *Subcase*: $\delta$ is not in $\gamma$'s dependency graph.

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

3.3 *Q.E.D.*

The above cases are exhaustive. In all cases $\gamma$ is always executed before $\delta$.

□

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.

### 5.7 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.

## 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

Instead of sending $PreAccept$ messages to every replica, a command leader 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:

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

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

   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.

   Alternatively, for up to seven total replicas, we can eliminate the second condition if we make the following modification to the protocol:

   **Accept-Deps:** Every sender of an Accept or AcceptReply message will attach a dependency list updated right before the Accept / AcceptReply was sent; every receiver of an Accept or AcceptReply will store the message in its log permanently.

   The updated dependency information in these recorded *Accept* and *AcceptReply* messages is only used during the recovery procedure, and has no role in the execution algorithm. Using this alternative protocol modification has the important benefit that it does not affect the chance of committing on the fast-path.

   When using the Accept-Deps variant of the protocol, it is important for replicas that have not received a command to not set dependencies on its corresponding instance. This may happen in implementations that use the optimization described in Section 5.3 (which involves setting implicit dependencies on all instances with IDs smaller than a given instance ID) and may result in the approximate sequence number not being updated correctly. There are two ways an implementation can avoid this problem: (a) acceptors attach the ranges of instances that they have not seen to *PreAcceptReplies* or (b) acceptors do not reply to *PreAccepts* before receiving messages for all instances that they would set implicit dependencies on.

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 Paxos-Accept Phase.

3. **If** at least one replica has committed command $\gamma$ in $Q.i$ (there is at most one such command), with attributes $deps_\gamma$ and $seq_\gamma$, $R$ commits $\gamma$ 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 Paxos-Accept Phase for this tuple at $Q.i$ (it will choose the one accepted with the highest ballot number, if there are multiple different accepted tuples at $Q.i$).

5. **If** at least $\lfloor \frac{F+1}{2} \rfloor$ replicas have pre-accepted $\gamma$ *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 $\gamma$ at $Q.i$, on the slow path (i.e., Phase 1, Phase 2, Paxos-Accept, Commit).

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

   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 $\delta$ such that:

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

   ii. $\gamma \notin deps_\delta$, and

   iii. (a) $\delta \notin deps_\gamma$, or

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

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

7. (a) **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 Paxos-Accept Phase for this tuple at Q.i.

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

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

(d) **Else if** There exists command $\gamma_0$ such that $R$ has deferred the recovery of $\gamma_0$ because of a conflict with $\gamma$, and $\gamma_0$'s initial command leader must have been part of $\gamma$'s fast quorum for $\gamma$ to have been committed on the fast path, then $R$ exits the recovery of $\gamma$ and starts the process of choosing $\gamma$ at $Q.i$, on the slow path.

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

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

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

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

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

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

(e) **Else if** There exists command $\gamma_0$ such that $R$ has deferred the recovery of $\gamma_0$ because of a conflict with $\gamma$, and $\gamma_0$'s initial command leader must have been part of $\gamma$'s fast quorum for $\gamma$ to have been committed on the fast path, then $R$ exits the recovery of $\gamma$ and starts the process of choosing $\gamma$ at $Q.i$, on the slow path.

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

This decision process is depicted in Figure 2.

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

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

*Proof.*

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

I **Case 1:** The chain contains two consecutive commands $\gamma$ and $\delta$ such that $\gamma \notin deps_\delta$ and $\delta \notin deps_\gamma$.
Let $R$ be a replica trying to recover $\gamma$. $R$ must believe that $\gamma$ may have been committed on the fast path. Eventually, $R$ will defer $\gamma$ and try to decide $\delta$, and, by our initial assumption, it must believe that $\delta$ too may have been committed on the fast path.
$R$ must be aware of the following sets and their properties:

1. $RESP_\gamma$, the set of all the replicas in $\gamma$'s fast quorum ($QUOR_\gamma$) that have responded to $R$'s prepare messages, does not include $L_\gamma$, the initial command leader for $\gamma$ (otherwise $\gamma$ could be decided);
2. $RESP_\delta$, the set of all the replicas in $\delta$'s quorum ($QUOR_\delta$) that have responded to $R$'s prepare messages, does not include $L_\delta$, the initial command leader for $\delta$;
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. $R$ infers that $L_\gamma \notin QUOR_\delta$—otherwise $\delta$ could not have been committed on the fast path, since $L_\gamma$ has set conflicting attributes for $\gamma$.
7. $R$ infers that $L_\delta \notin QUOR_\gamma$—otherwise $\gamma$ could not have been committed on the fast path, since $L_\delta$ has set conflicting attributes for $\delta$.
8. Since there are at most $F$ replicas that do not reply to $R$, and $L_\gamma$ (the possibly failed command leader for $\gamma$) must be one of them (otherwise $R$ could decide $\gamma$), by 6, 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 $\delta$ 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 2, 5 and 7, $R$ 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 8 and our fast-path quorum requirement, the cardinality of the union of these sets must be at least $F + \lfloor \frac{F+1}{2} \rfloor + \lfloor \frac{F+1}{2} \rfloor + 1 + 1 > 2F + 1$. But this is impossible, because this union must be a subset of the replica set, and its cardinality is $2F + 1$. Therefore, some of these sets overlap, so $R$ cannot be simultaneously uncertain about $\gamma$ and $\delta$. Our assumption that the recovery procedure could deadlock is false.

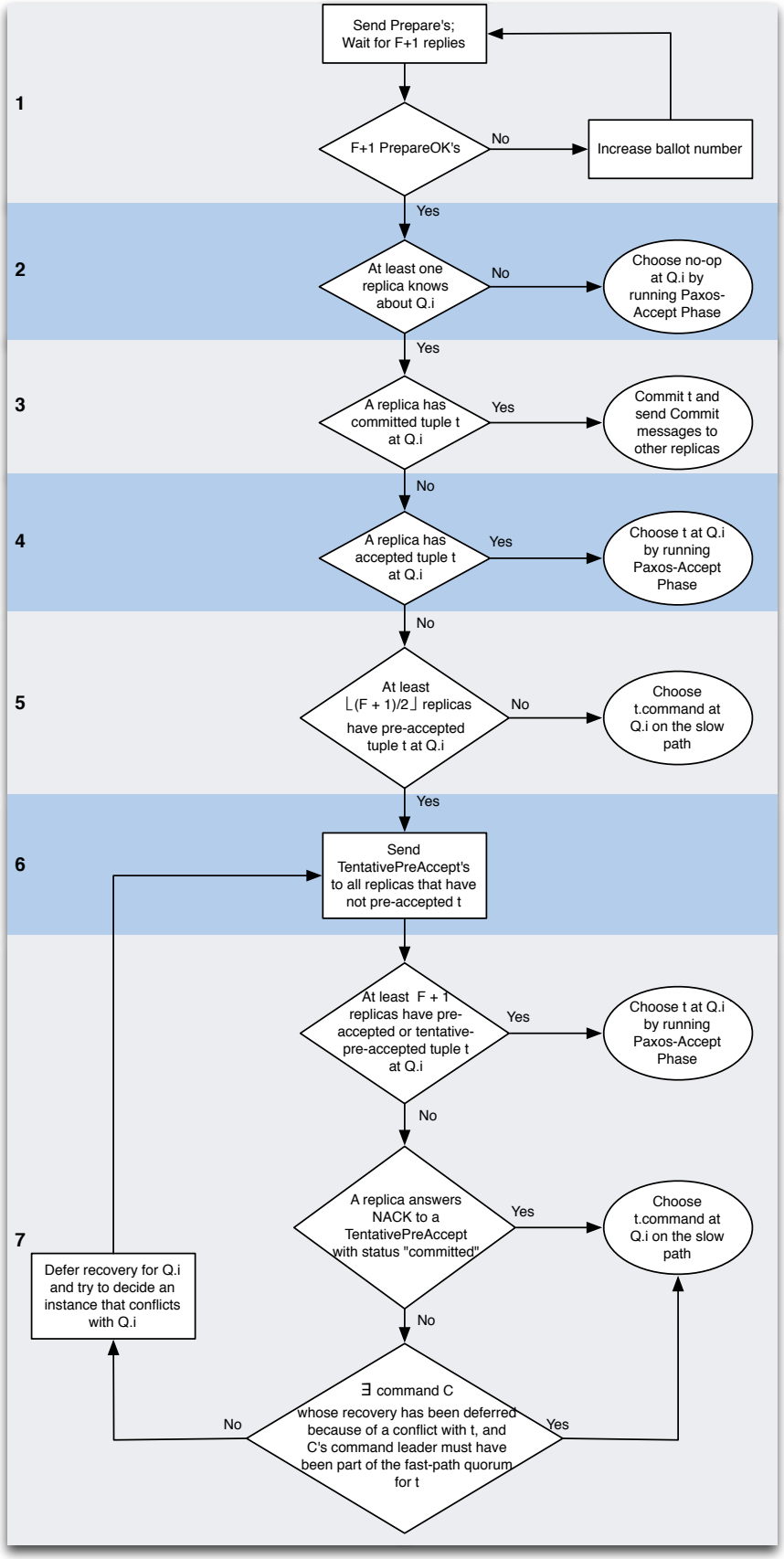


**Figure 2: Decision process for recovery in optimized EPaxos.**

II **Case 2:** $c_{i+1} \in deps_{c_i}$, for all $i = 1..n$ (with $c_{n+1} \equiv c_1$).
For recovery to defer, it must be the case that $c_i \sim c_{i+1}$, $c_i \notin deps_{c_{i+1}}$, $c_{i+1} \in deps_{c_i}$, and $seq_{c_i} \geq seq_{c_{i+1}}$ for any $i$ (and $c_{n+1} \equiv c_1$). Then $seq_{c_1} = seq_{c_2} = ... = seq_{c_n}$. Note also that this is only possible for $n \geq 3$.

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

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

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

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

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

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

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

The subcases enumerated above are exhaustive.

□

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

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

*Proof.*

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

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

2 *Case:* A tuple $(\gamma, deps_\gamma, seq_\gamma)$ has been 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 Paxos-Accept Phase. By the guarantees of the classic Paxos algorithm, only this tuple can ever be committed at $Q.i$.

2.2 *Subcase:* $(\gamma, deps_\gamma, seq_\gamma)$ 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: $\delta$ must be pre-accepted by a majority of replicas, and that majority will intersect $\gamma$'s quorum (itself a majority) in at least one replica, which will ensure that at least one command will be in the other's $deps$ set.

2.2.2 If the protocol does not implement **Accept-Deps**, but does implement **FP-deps-committed**, then no interfering command $\delta \sim \gamma$, $\delta \in deps_\gamma$, 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 $\gamma_0$ 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 $\gamma_0$'s recovery. Since $\gamma_0$ had been committed on the fast path, then by the *additional condition for the fast-path in optimized EPaxos* (**FP-deps-committed**), all its dependencies, including $\delta$ must have been committed before $seq_{\gamma_0}$ had been computed. Then, $\delta$ 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 $\delta$ had been committed incorrectly by the recovery procedure (before $\gamma_0$), 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 $\gamma$ 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 $\gamma$ has been committed on the fast path, then, by the additional condition for the fast-path in optimized EPaxos, all its dependencies, including $\delta$ must have been committed

before $seq_\gamma$ had been computed. Then, $\delta$ 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 $\delta$ 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 $\gamma$ 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 $\delta$'s recovery by correctly committing its initial fast-path attributes. Then $seq_\delta$ cannot be larger or equal to $seq_\gamma$, since $seq_\gamma$ has been updated to be larger than $seq_\delta$ at $\delta$'s initial commit time.

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

2.2.3 If the protocol does not implement **FP-deps-committed**, but does implement **Accept-Deps**, the recovery procedure will not exit on the first Else case of step 7'.

PROOF: For the recovery procedure to exit on the first Else case of step 7', there must exist a committed tuple $(\delta, deps_\delta, seq_\delta)$, with $\delta \sim \gamma$, $\delta \in deps_\gamma$, $\gamma \notin deps_\delta$ and $seq_\delta \geq seq_\gamma$, and there must exist an *Accept* or *AcceptReply* message for this tuple sent by a replica $R'$ in $\gamma's$ fast-path quorum such that the additional dependencies for this message do not include $\gamma$. Then $R'$ must have accepted $(\delta, deps_\delta, seq_\delta)$ before pre-accepting $\gamma$. This is impossible, since then $R'$ would not have pre-accepted $\gamma$ with $seq_\gamma \leq seq_\delta$, as we know it must have for $\gamma$ to be committed on the fast-path.

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

2.2.5 *Q.E.D*
By the recovery procedure, 2.2.1, 2.2.2, 2.2.3, 2.2.4 and Lemma 2 the recovery procedure will be successful in getting $F$ replicas to pre-accept tuple $(\gamma, deps_\gamma, seq_\gamma)$ (not counting the implicit pre-accept of the initial command leader), and it will start the Paxos-Accept Phase for this tuple.

2.3 *Q.E.D*

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

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

□

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

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

*Proof.*

Let $\gamma$ and $\delta$ be two commands that interfere and have been committed. We show that all replicas execute $\gamma$ and $\delta$ in the same order.

1 *Case:* Both $\gamma$ and $\delta$ 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, $\gamma$ and $\delta$ will be executed in the same order by every replica.

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

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

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

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

We show that either $\gamma$ has $\delta$ as a dependency or $\delta$ has $\gamma$ as a dependency:

2.2.1 *Sub-subcase:* $\gamma$ had been pre-accepted with $\delta \in deps_\gamma$.

$\gamma$'s pre-accepted attributes as received in the recovery procedure at step 7 do not change, so $\gamma$ will be committed with $\delta$ as a dependency.

2.2.2 *Sub-subcase:* $\gamma$ 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 $\gamma$'s original command leader have pre-accepted $\gamma$ as a result of a $PreAccept$ or a $TentativePreAccept$. $\delta$ will also have been pre-accepted by a majority of replicas, so there is at least one replica that has pre-accepted both $\delta$ and $\gamma$, and whose replies are taken into account both when establishing $\delta$'s commit attributes and in the recovery procedure for $\gamma$. Let this replica be $R$:

2.2.2.1 *Sub-sub-subcase:* $R$ pre-accepts $\gamma$ as a result of receiving a $PreAccept$ from $\gamma$'s initial command leader.

Then $R$ must have learned about $\gamma$ before receiving a $PreAccept$ for $\delta$, so $\gamma \in deps_\delta$.

2.2.2.2 *Sub-sub-subcase:* $R$ pre-accepts $\gamma$ 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 $\delta$ such that $\gamma \in deps_\delta$, or $\delta$ reaches $R$ after the $TentativePreAccept$ for $\gamma$. In either case, $\gamma \in deps_\delta$ when $\delta$ 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:* $\delta$ is first committed as a result of the recovery procedure, while $\gamma$ is first committed by its initial command leader without running the recovery procedure.
Just like case 2, with $\gamma$ and $\delta$ interchanged.

4 *Case:* Both $\gamma$ and $\delta$ are first committed after the recovery procedure.
If at least one of the commands is committed before step 7 in the recovery procedure, or after exiting step 7 on one of the Else branches, the situation is reducible to one of the previous cases.
The only remaining subcase is that when both commands are committed after exiting step 7 on the If branch. Assume no command has the other in its dependency list when exiting step 7 of the recovery procedure. But each command has been pre-accepted by a majority or replicas (either as a result of $PreAccept$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 $\gamma$ before $\delta$, then, by the conditions in step 6 of the recovery procedure, $R$ will not acknowledge $\delta$ without a dependency for $\gamma$ (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 $\gamma$ and $\delta$ be two interfering commands serialized by clients: $\delta$ is proposed only after a replica has committed $\gamma$. We show that $\gamma$ will always be executed before $\delta$

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

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

2 *Case:* $R$ receives a $TentativePreAccept$ for $\delta$ at some point other than step 7' in the recovery procedure.
Since $R$ will ACK (otherwise $\delta$ would not be committed), and $\delta \notin deps_\gamma$ (because $\delta$ was proposed after $\gamma$ 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, $\gamma$ will always be executed before $\delta$.

3 *Case:* $R$ receives a $TentativePreAccept$ for $\delta$ at step 7' in the recovery procedure.
In this case, there must exist a command $\gamma'$ such that $\gamma' \sim \delta$, $\gamma' \in deps_\delta$, $\delta \notin deps_{\gamma'}$ and $seq_\delta \leq seq_{\gamma'}$, and $R$ is instructed to ignore the conflict between the attributes of $\delta$ and those of $\gamma'$. We show that $\gamma \neq \gamma'$.
Assume $\gamma' = \gamma$. Then $\gamma$ must have first been committed on the slow-path, because otherwise $\delta$ couldn't have acquired a stale dependency on $\gamma$ (i.e., a dependency where $seq_\delta$ hasn't been updated to be larger than the committed $seq_\gamma$)—this remains true for implicit dependencies, as per the discussion in Section 6.1.
Then there must exist a replica $R'$ that has participated in the Paxos-Accept phase for $\gamma$, and that was also supposed to be part of $\delta$'s fast-path quorum. We first note that $L_\delta \neq R'$, where $L_\delta$ is the initial command leader for $\delta$, because it wouldn't have set the conflicting sequence number for $\delta$ after having accepted the commit-time attributes for $\gamma$.

3.1 *Subcase* $F = 2$.

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

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

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

3.2 *Subcase* $F = 3$.

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

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

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

□

# 7 Linearizability and Serializability in Egalitarian Paxos

Egalitarian Paxos guarantees that interfering commands are executed in a linearizable order. We now show what the overall consistency properties of a system using Egalitarian Paxos are.

If the interference relation is transitive (which is equivalent with a setting where each command refers to the state of a single object) then execution linearizability implies the whole system is linearizable, as defined by Herlihy and Wing: linearizability is a *local* property, meaning that *"a system is linearizable if each individual object is linearizable"* [4].

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

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

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

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

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

client 2: *read A and B*

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

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

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

*Proof.*

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

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

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

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

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

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

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

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

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

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

**Table 1: Consistency guarantees in EPaxos.**

□

Table 1 summarizes the consistency guarantees of EPaxos.

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

# 8 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 *Proc. 26th ACM SOSP*, 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] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), July 1990.

[5] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proc. USENIX ATC*, USENIXATC'10, Berkeley, CA, USA, 2010. USENIX Association.

[6] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. Vertical Paxos and primary-backup replication. Technical report, Microsoft Research, 2009.

[7] 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.

[8] 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.

## APPENDIX

TLA+ specification for the Egalitarian Paxos commit protocol:

MODULE *EgalitarianPaxos*

EXTENDS *Naturals*, *FiniteSets*

$Max(S) \triangleq$ IF $S = \{\}$ THEN $0$ ELSE CHOOSE $i \in S : \forall j \in S : j \leq i$

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

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

ASSUME *IsFiniteSet*(*Replicas*)

Quorum conditions: (simplified)

ASSUME $\forall r \in Replicas :$
$\quad \land SlowQuorums(r) \subseteq$ SUBSET $Replicas$
$\quad \land \forall SQ \in SlowQuorums(r) :$
$\quad\quad \land r \in SQ$
$\quad\quad \land Cardinality(SQ) = (Cardinality(Replicas) \div 2) + 1$

ASSUME $\forall r \in Replicas :$
$\quad \land FastQuorums(r) \subseteq$ SUBSET $Replicas$
$\quad \land \forall FQ \in FastQuorums(r) :$
$\quad\quad \land r \in FQ$
$\quad\quad \land Cardinality(FQ) = (Cardinality(Replicas) \div 2) +$
$\quad\quad\quad\quad ((Cardinality(Replicas) \div 2) + 1) \div 2$

Special none command

$none \triangleq$ CHOOSE $c : c \notin Commands$

The instance space

$Instances \triangleq Replicas \times (1 \,..\, Cardinality(Commands))$

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

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

All possible protocol messages:

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

Variables:

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

VARIABLES *cmdLog*, *proposed*, *executed*, *sentMsg*, *crtInst*, *leaderOfInst*,

$committed,\ ballots,\ preparing$

$$
\begin{array}{l}
TypeOK \triangleq \\
\quad \land \quad cmdLog \in [Replicas \to \text{SUBSET } [inst : Instances, \\
\qquad\qquad\qquad\qquad\qquad\qquad status : Status, \\
\qquad\qquad\qquad\qquad\qquad\qquad ballot\ : Nat \times Replicas, \\
\qquad\qquad\qquad\qquad\qquad\qquad cmd : Commands \cup \{none\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad deps : \text{SUBSET } Instances, \\
\qquad\qquad\qquad\qquad\qquad\qquad seq : Nat]] \\
\quad \land \quad proposed \in \text{SUBSET } Commands \\
\quad \land \quad executed\ \in [Replicas \to \text{SUBSET } (Nat \times Commands)] \\
\quad \land \quad sentMsg \in \text{SUBSET } Message \\
\quad \land \quad crtInst\ \ \in [Replicas \to Nat] \\
\quad \land \quad leaderOfInst \in [Replicas \to \text{SUBSET } Instances] \\
\quad \land \quad committed \in [Instances \to \text{SUBSET } ((Commands \cup \{none\}) \times \\
\qquad\qquad\qquad\qquad\qquad\qquad (\text{SUBSET } Instances) \times \\
\qquad\qquad\qquad\qquad\qquad\qquad Nat)] \\
\quad \land \quad ballots \in Nat \\
\quad \land \quad preparing \in [Replicas \to \text{SUBSET } Instances]
\end{array}
$$

$$
\begin{array}{l}
vars \triangleq \langle cmdLog,\ proposed,\ executed,\ sentMsg,\ crtInst,\ leaderOfInst, \\
\qquad committed,\ ballots,\ preparing \rangle
\end{array}
$$

Initial state predicate

$$
\begin{array}{l}
Init \triangleq \\
\quad \land\ sentMsg = \{\} \\
\quad \land\ cmdLog = [r \in Replicas \mapsto \{\}] \\
\quad \land\ proposed = \{\} \\
\quad \land\ executed\ = [r \in Replicas \mapsto \{\}] \\
\quad \land\ crtInst = [r\ \ \in Replicas \mapsto 1] \\
\quad \land\ leaderOfInst = [r \in Replicas \mapsto \{\}] \\
\quad \land\ committed = [i \in Instances \mapsto \{\}] \\
\quad \land\ ballots = 1 \\
\quad \land\ preparing = [r \in Replicas \mapsto \{\}]
\end{array}
$$

Actions

$$
\begin{array}{l}
StartPhase1(C,\ cleader,\ Q,\ inst,\ ballot,\ oldMsg) \triangleq \\
\quad \text{LET } newDeps \triangleq \{rec.inst : rec \in cmdLog[cleader]\} \\
\qquad newSeq \triangleq 1 + Max(\{t.seq : t \in cmdLog[cleader]\}) \\
\qquad oldRecs \triangleq \{rec \in cmdLog[cleader] : rec.inst = inst\} \text{IN} \\
\qquad \land\ cmdLog' = [cmdLog \text{ EXCEPT } ![cleader] = (@ \setminus oldRecs) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst\ \ \mapsto inst, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto \text{“pre-accepted”},
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\quad ballot \mapsto ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad cmd \quad \mapsto C,\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad deps \quad \mapsto newDeps,\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad seq \quad\ \mapsto newSeq]\}]\\
\qquad\qquad \land\ leaderOfInst' = [leaderOfInst \text{ EXCEPT } ![cleader] = @ \cup \{inst\}]\\
\qquad\qquad \land\ sentMsg' = (sentMsg \setminus oldMsg) \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad [type \ : \{\text{“pre-accept”}\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ src \quad : \{cleader\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ dst \quad : Q \setminus \{cleader\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ inst \quad : \{inst\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ballot : \{ballot\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ cmd \quad : \{C\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ deps \quad : \{newDeps\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ seq \quad : \{newSeq\}]\\
\\
Propose(C,\ cleader) \triangleq\\
\quad \text{LET } newInst \triangleq \langle cleader,\ crtInst[cleader]\rangle\\
\qquad\ \ newBallot \triangleq \langle 0,\ cleader\rangle\\
\quad \text{IN} \quad \land\ proposed' = proposed \cup \{C\}\\
\qquad\ \ \land\ (\exists\, Q \in FastQuorums(cleader) :\\
\qquad\qquad\qquad StartPhase1(C,\ cleader,\ Q,\ newInst,\ newBallot,\ \{\}))\\
\qquad\ \ \land\ crtInst' = [crtInst \text{ EXCEPT } ![cleader] = @ + 1]\\
\qquad\ \ \land \text{UNCHANGED } \langle executed,\ committed,\ ballots,\ preparing\rangle\\
\\
Phase1Reply(replica) \triangleq\\
\quad \exists\, msg \in sentMsg :\\
\qquad \land\ msg.type = \text{“pre-accept”}\\
\qquad \land\ msg.dst = replica\\
\qquad \land\ \text{LET } oldRec \triangleq \{rec \in cmdLog[replica] : rec.inst = msg.inst\} \text{IN}\\
\qquad\quad \land\ (\forall\, rec \in oldRec :\\
\qquad\qquad\ (rec.ballot = msg.ballot \lor rec.ballot[1] < msg.ballot[1]))\\
\qquad\quad \land\ \text{LET } newDeps \triangleq msg.deps \cup\\
\qquad\qquad\qquad\qquad\qquad (\{t.inst : t \in cmdLog[replica]\} \setminus \{msg.inst\})\\
\qquad\qquad\qquad newSeq \triangleq Max(\{msg.seq,\\
\qquad\qquad\qquad\qquad\qquad\qquad 1 + Max(\{t.seq : t \in cmdLog[replica]\})\})\\
\qquad\qquad\qquad instCom \triangleq \{t.inst : t \in \{tt \in cmdLog[replica] :\\
\qquad\qquad\qquad\qquad\qquad\ tt.status \in \{\text{“committed”},\ \text{“executed”}\}\}\} \text{IN}\\
\qquad\qquad\ \land\ cmdLog' = [cmdLog \text{ EXCEPT } ![replica] = (@ \setminus oldRec) \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \quad \mapsto msg.inst,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto \text{“pre-accepted”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ballot \mapsto msg.ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ cmd \quad \mapsto msg.cmd,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ deps \quad \mapsto newDeps,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ seq \quad \mapsto newSeq]\}]\\
\qquad\qquad\ \land\ sentMsg' = (sentMsg \setminus \{msg\}) \cup
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{[type \;\mapsto \text{“pre-accept-reply”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; src \;\mapsto replica,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; dst \;\mapsto msg.src,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; inst \;\mapsto msg.inst,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; ballot \mapsto msg.ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; deps \;\mapsto newDeps,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; seq \;\mapsto newSeq,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\; committed \mapsto instCom]\}\\
\qquad\qquad\qquad\qquad\qquad\qquad\wedge \;\textsc{unchanged}\ \langle proposed,\ crtInst,\ executed,\ leaderOfInst,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad committed,\ ballots,\ preparing\rangle
\end{array}
$$

$$
\begin{array}{l}
Phase1Fast(cleader,\ i,\ Q) \triangleq\\
\quad \wedge i \ \in leaderOfInst[cleader]\\
\quad \wedge Q \in FastQuorums(cleader)\\
\quad \wedge \exists\, record \in cmdLog[cleader] :\\
\qquad \wedge record.inst = i\\
\qquad \wedge record.status = \text{“pre-accepted”}\\
\qquad \wedge record.ballot[1] = 0\\
\qquad \wedge \textsc{let}\ replies \triangleq \{msg \in sentMsg :\\
\qquad\qquad\qquad\qquad \wedge msg.inst = i\\
\qquad\qquad\qquad\qquad \wedge msg.type = \text{“pre-accept-reply”}\\
\qquad\qquad\qquad\qquad \wedge msg.dst = cleader\\
\qquad\qquad\qquad\qquad \wedge msg.src \in Q\\
\qquad\qquad\qquad\qquad \wedge msg.ballot = record.ballot\}\textsc{in}\\
\qquad\quad \wedge \ (\forall\, replica \in (Q \setminus \{cleader\}) :\\
\qquad\qquad \exists\, msg \in replies : msg.src = replica)\\
\qquad\quad \wedge \ (\forall\, r1,\ r2 \in replies :\\
\qquad\qquad \wedge r1.deps = r2.deps\\
\qquad\qquad \wedge r1.seq = r2.seq)\\
\qquad\quad \wedge \ \textsc{let}\ r \triangleq \textsc{choose}\ r \in replies : \textsc{true}\textsc{in}\\
\qquad\qquad \wedge \ \textsc{let}\ localCom \triangleq \{t.inst :\\
\qquad\qquad\qquad\qquad t \in \{tt \in cmdLog[cleader] :\\
\qquad\qquad\qquad\qquad\qquad tt.status \in \{\text{“committed”},\ \text{“executed”}\}\}\}\\
\qquad\qquad\qquad\quad extCom \triangleq \textsc{union}\ \{msg.committed : msg \in replies\}\textsc{in}\\
\qquad\qquad\qquad\quad (r.deps \subseteq (localCom \cup extCom))\\
\qquad\qquad \wedge \ cmdLog' = [cmdLog\ \textsc{except}\ ![cleader] = (@ \setminus \{record\}) \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \;\mapsto i,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; status \mapsto \text{“committed”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; ballot \mapsto record.ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; cmd \;\mapsto record.cmd,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; deps \;\mapsto r.deps,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\;\; seq \;\mapsto r.seq]\}]\\
\qquad\qquad \wedge \ sentMsg' = (sentMsg \setminus replies) \cup\\
\qquad\qquad\qquad\qquad\quad \{[type \;\mapsto \text{“commit”},\\
\qquad\qquad\qquad\qquad\quad inst \;\mapsto i,
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad ballot \;\mapsto record.ballot,\\
\qquad\qquad\qquad\qquad\quad cmd \;\;\mapsto record.cmd,\\
\qquad\qquad\qquad\qquad\quad deps \;\;\mapsto r.deps,\\
\qquad\qquad\qquad\qquad\quad seq \;\;\mapsto r.seq]\}\\
\qquad\qquad\wedge\;\; leaderOfInst' = [leaderOfInst \text{ EXCEPT } ![cleader] = @ \setminus \{i\}]\\
\qquad\qquad\wedge\;\; committed' = [committed \text{ EXCEPT } ![i] =\\
\qquad\qquad\qquad\qquad\qquad\qquad @ \cup \{\langle record.cmd,\, r.deps,\, r.seq\rangle\}]\\
\qquad\qquad\wedge\;\; \text{UNCHANGED } \langle proposed,\, executed,\, crtInst,\, ballots,\, preparing\rangle
\end{array}
$$

$$
\begin{array}{l}
Phase1Slow(cleader,\, i,\, Q) \;\triangleq\\
\quad\wedge i \;\in leaderOfInst[cleader]\\
\quad\wedge Q \in SlowQuorums(cleader)\\
\quad\wedge \exists\, record \in cmdLog[cleader] :\\
\qquad\wedge record.inst = i\\
\qquad\wedge record.status = \text{“pre-accepted”}\\
\qquad\wedge \text{LET } replies \triangleq \{msg \in sentMsg :\\
\qquad\qquad\qquad\qquad\wedge msg.inst = i\\
\qquad\qquad\qquad\qquad\wedge msg.type = \text{“pre-accept-reply”}\\
\qquad\qquad\qquad\qquad\wedge msg.dst = cleader\\
\qquad\qquad\qquad\qquad\wedge msg.src \in Q\\
\qquad\qquad\qquad\qquad\wedge msg.ballot = record.ballot\}\text{IN}\\
\qquad\wedge\;\; (\forall\, replica \in (Q \setminus \{cleader\}) : \exists\, msg \in replies : msg.src = replica)\\
\qquad\wedge\;\; \text{LET } finalDeps \triangleq \text{UNION } \{msg.deps : msg \in replies\}\\
\qquad\qquad\quad finalSeq \triangleq Max(\{msg.seq : msg \in replies\})\text{IN}\\
\qquad\qquad\wedge\;\; cmdLog' = [cmdLog \text{ EXCEPT } ![cleader] = (@ \setminus \{record\}) \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \;\;\mapsto i,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; status \mapsto \text{“accepted”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; ballot \mapsto record.ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; cmd \;\;\mapsto record.cmd,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; deps \;\;\mapsto finalDeps,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; seq \;\;\mapsto finalSeq]\}]\\
\qquad\qquad\wedge\;\; \exists\, SQ \in SlowQuorums(cleader) :\\
\qquad\qquad\quad (sentMsg' = (sentMsg \setminus replies) \cup\\
\qquad\qquad\qquad\qquad [type : \{\text{“accept”}\},\\
\qquad\qquad\qquad\qquad\;\; src : \{cleader\},\\
\qquad\qquad\qquad\qquad\;\; dst : SQ \setminus \{cleader\},\\
\qquad\qquad\qquad\qquad\;\; inst : \{i\},\\
\qquad\qquad\qquad\qquad\;\; ballot : \{record.ballot\},\\
\qquad\qquad\qquad\qquad\;\; cmd : \{record.cmd\},\\
\qquad\qquad\qquad\qquad\;\; deps : \{finalDeps\},\\
\qquad\qquad\qquad\qquad\;\; seq : \{finalSeq\}])\\
\qquad\qquad\wedge\;\; \text{UNCHANGED } \langle proposed,\, executed,\, crtInst,\, leaderOfInst,\\
\qquad\qquad\qquad\qquad\qquad committed,\, ballots,\, preparing\rangle
\end{array}
$$

$$
Phase2Reply(replica) \;\triangleq
$$

$$
\begin{array}{l}
\exists\, msg \in sentMsg : \\
\quad \land\ msg.type = \text{“accept”} \\
\quad \land\ msg.dst = replica \\
\quad \land\ \text{LET}\ oldRec \triangleq \{rec \in cmdLog[replica] : rec.inst = msg.inst\}\ \text{IN} \\
\qquad \land\ (\forall\, rec \in oldRec : (rec.ballot = msg.ballot \lor \\
\qquad\qquad\qquad\qquad rec.ballot[1] < msg.ballot[1])) \\
\qquad \land\ cmdLog' = [cmdLog\ \text{EXCEPT}\ ![replica] = (@ \setminus oldRec) \cup \\
\qquad\qquad\qquad\qquad \{[inst \mapsto msg.inst, \\
\qquad\qquad\qquad\qquad\ status \mapsto \text{“accepted”}, \\
\qquad\qquad\qquad\qquad\ ballot \mapsto msg.ballot, \\
\qquad\qquad\qquad\qquad\ cmd \mapsto msg.cmd, \\
\qquad\qquad\qquad\qquad\ deps \mapsto msg.deps, \\
\qquad\qquad\qquad\qquad\ seq \mapsto msg.seq]\}] \\
\qquad \land\ sentMsg' = (sentMsg \setminus \{msg\}) \cup \\
\qquad\qquad\qquad\qquad \{[type \mapsto \text{“accept-reply”}, \\
\qquad\qquad\qquad\qquad\ src \mapsto replica, \\
\qquad\qquad\qquad\qquad\ dst \mapsto msg.src, \\
\qquad\qquad\qquad\qquad\ inst \mapsto msg.inst, \\
\qquad\qquad\qquad\qquad\ ballot \mapsto msg.ballot]\} \\
\qquad \land\ \text{UNCHANGED}\ \langle proposed, crtInst, executed, leaderOfInst, \\
\qquad\qquad\qquad\qquad committed, ballots, preparing\rangle
\end{array}
$$

$$
\begin{array}{l}
Phase2Finalize(cleader, i, Q) \triangleq \\
\quad \land\ i \in leaderOfInst[cleader] \\
\quad \land\ Q \in SlowQuorums(cleader) \\
\quad \land\ \exists\, record \in cmdLog[cleader] : \\
\qquad \land\ record.inst = i \\
\qquad \land\ record.status = \text{“accepted”} \\
\qquad \land\ \text{LET}\ replies \triangleq \{msg \in sentMsg : \\
\qquad\qquad\qquad\qquad \land\ msg.inst = i \\
\qquad\qquad\qquad\qquad \land\ msg.type = \text{“accept-reply”} \\
\qquad\qquad\qquad\qquad \land\ msg.dst = cleader \\
\qquad\qquad\qquad\qquad \land\ msg.src \in Q \\
\qquad\qquad\qquad\qquad \land\ msg.ballot = record.ballot\}\ \text{IN} \\
\qquad\quad \land\ (\forall\, replica \in (Q \setminus \{cleader\}) : \exists\, msg \in replies : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad msg.src = replica) \\
\qquad\quad \land\ cmdLog' = [cmdLog\ \text{EXCEPT}\ ![cleader] = (@ \setminus \{record\}) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \mapsto i, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ status \mapsto \text{“committed”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ ballot \mapsto record.ballot, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ cmd \mapsto record.cmd, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ deps \mapsto record.deps, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ seq \mapsto record.seq]\}] \\
\qquad\quad \land\ sentMsg' = (sentMsg \setminus replies) \cup
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\quad \{[type \;\mapsto \text{“commit”},\\
\qquad\qquad\qquad\qquad\quad inst \;\mapsto i,\\
\qquad\qquad\qquad\qquad\quad ballot \;\mapsto record.ballot,\\
\qquad\qquad\qquad\qquad\quad cmd \;\mapsto record.cmd,\\
\qquad\qquad\qquad\qquad\quad deps \;\mapsto record.deps,\\
\qquad\qquad\qquad\qquad\quad seq \;\mapsto record.seq]\}\\
\qquad\qquad \land\;\; committed' = [committed \text{ EXCEPT } ![i] = @ \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad \{\langle record.cmd,\, record.deps,\, record.seq\rangle\}]\\
\qquad\qquad \land\;\; leaderOfInst' = [leaderOfInst \text{ EXCEPT } ![cleader] = @ \setminus \{i\}]\\
\qquad\qquad \land\;\; \text{UNCHANGED } \langle proposed,\, executed,\, crtInst,\, ballots,\, preparing\rangle
\end{array}
$$

$$
\begin{array}{l}
Commit(replica,\, cmsg) \;\triangleq\\
\quad \text{LET } oldRec \;\triangleq\; \{rec \in cmdLog[replica] : rec.inst = cmsg.inst\} \text{IN}\\
\qquad \land \forall\, rec \in oldRec : (rec.status \notin \{\text{“committed”},\ \text{“executed”}\} \land\\
\qquad\qquad\qquad\qquad rec.ballot[1] \leq cmsg.ballot[1])\\
\qquad \land\; cmdLog' = [cmdLog \text{ EXCEPT } ![replica] = (@ \setminus oldRec) \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \;\mapsto cmsg.inst,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; status \;\mapsto \text{“committed”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; ballot \;\mapsto cmsg.ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; cmd \;\mapsto cmsg.cmd,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; deps \;\mapsto cmsg.deps,\\
\qquad\qquad\qquad\qquad\qquad\qquad\;\; seq \;\mapsto cmsg.seq]\}]\\
\qquad \land\; committed' = [committed \text{ EXCEPT } ![cmsg.inst] = @ \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad \{\langle cmsg.cmd,\, cmsg.deps,\, cmsg.seq\rangle\}]\\
\qquad \land \text{UNCHANGED } \langle proposed,\, executed,\, crtInst,\, leaderOfInst,\\
\qquad\qquad\qquad\qquad sentMsg,\, ballots,\, preparing\rangle
\end{array}
$$

Recovery actions

$$
\begin{array}{l}
SendPrepare(replica,\, i,\, Q) \;\triangleq\\
\quad \land\; i \notin leaderOfInst[replica]\\
\quad \land\; i \notin preparing[replica]\\
\quad \land\; ballots \leq MaxBallot\\
\quad \land \neg(\exists\, rec \quad \in cmdLog[replica] :\\
\qquad\qquad\qquad\qquad \land\; rec.inst = i\\
\qquad\qquad\qquad\qquad \land\; rec.status \in \{\text{“committed”},\ \text{“executed”}\})\\
\quad \land\; sentMsg' = sentMsg \cup\\
\qquad\qquad\qquad\qquad [type \;: \{\text{“prepare”}\},\\
\qquad\qquad\qquad\qquad\;\; src \;: \{replica\},\\
\qquad\qquad\qquad\qquad\;\; dst \;: Q,\\
\qquad\qquad\qquad\qquad\;\; inst \;: \{i\},\\
\qquad\qquad\qquad\qquad\;\; ballot \;: \{\langle ballots,\, replica\rangle\}]\\
\quad \land\; ballots' \;\; = ballots + 1\\
\quad \land\; preparing' = [preparing \text{ EXCEPT } ![replica] = @ \cup \{i\}]\\
\quad \land \text{UNCHANGED } \langle cmdLog,\, proposed,\, executed,\, crtInst,
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad leaderOfInst,\ committed\rangle \\
\\
ReplyPrepare(replica) \triangleq \\
\quad \exists\, msg \in sentMsg : \\
\qquad \land\ msg.type = \text{“prepare”} \\
\qquad \land\ msg.dst = replica \\
\qquad \land\ \ \lor \exists\, rec \in cmdLog[replica] : \\
\qquad\qquad\qquad \land\, rec.inst = msg.inst \\
\qquad\qquad\qquad \land\, msg.ballot[1] > rec.ballot[1] \\
\qquad\qquad\qquad \land\, sentMsg' = (sentMsg \setminus \{msg\}) \cup \\
\qquad\qquad\qquad\qquad\qquad \{[type \ \ \mapsto \text{“prepare-reply”}, \\
\qquad\qquad\qquad\qquad\qquad\ \ src \ \ \mapsto replica, \\
\qquad\qquad\qquad\qquad\qquad\ \ dst \ \ \mapsto msg.src, \\
\qquad\qquad\qquad\qquad\qquad\ \ inst \ \ \mapsto rec.inst, \\
\qquad\qquad\qquad\qquad\qquad\ \ ballot \mapsto msg.ballot, \\
\qquad\qquad\qquad\qquad\qquad\ \ prev\_ballot \mapsto rec.ballot, \\
\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto rec.status, \\
\qquad\qquad\qquad\qquad\qquad\ \ cmd \ \ \mapsto rec.cmd, \\
\qquad\qquad\qquad\qquad\qquad\ \ deps \ \ \mapsto rec.deps, \\
\qquad\qquad\qquad\qquad\qquad\ \ seq \ \ \mapsto rec.seq]\} \\
\qquad\qquad\qquad \land\, cmdLog' = [cmdLog \ \text{EXCEPT}\ ![replica] = (@ \setminus \{rec\}) \cup \\
\qquad\qquad\qquad\qquad\qquad \{[inst \ \ \mapsto rec.inst, \\
\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto rec.status, \\
\qquad\qquad\qquad\qquad\qquad\ \ ballot \ \mapsto msg.ballot, \\
\qquad\qquad\qquad\qquad\qquad\ \ cmd \ \ \mapsto rec.cmd, \\
\qquad\qquad\qquad\qquad\qquad\ \ deps \ \ \mapsto rec.deps, \\
\qquad\qquad\qquad\qquad\qquad\ \ seq \ \ \mapsto rec.seq]\}] \\
\qquad\qquad\qquad \land \text{IF}\ rec.inst \in leaderOfInst[replica]\ \text{THEN} \\
\qquad\qquad\qquad\qquad\qquad \land\, leaderOfInst' = [leaderOfInst \ \text{EXCEPT}\ ![replica] = \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad @ \setminus \{rec.inst\}] \\
\qquad\qquad\qquad\qquad\qquad \land\, \text{UNCHANGED}\ \langle proposed,\ executed,\ committed, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad crtInst,\ ballots,\ preparing\rangle \\
\qquad\qquad\qquad\qquad\ \ \text{ELSE}\ \ \text{UNCHANGED}\ \langle proposed,\ executed,\ committed,\ crtInst, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ballots,\ preparing,\ leaderOfInst\rangle \\
\\
\qquad\quad \lor\ \land \neg(\exists\, rec \quad \in cmdLog[replica] : rec.inst = msg.inst) \\
\qquad\qquad\ \ \land\, sentMsg' = (sentMsg \setminus \{msg\}) \cup \\
\qquad\qquad\qquad\qquad\qquad \{[type \ \ \mapsto \text{“prepare-reply”}, \\
\qquad\qquad\qquad\qquad\qquad\ \ src \ \ \mapsto replica, \\
\qquad\qquad\qquad\qquad\qquad\ \ dst \ \ \mapsto msg.src, \\
\qquad\qquad\qquad\qquad\qquad\ \ inst \ \ \mapsto msg.inst, \\
\qquad\qquad\qquad\qquad\qquad\ \ ballot \mapsto msg.ballot, \\
\qquad\qquad\qquad\qquad\qquad\ \ prev\_ballot \mapsto \langle 0,\ replica\rangle, \\
\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto \text{“not-seen”}, \\
\qquad\qquad\qquad\qquad\qquad\ \ cmd \ \ \mapsto none,
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\quad deps \;\mapsto \{\},\\
\qquad\qquad\qquad\quad seq \;\mapsto 0]\}\\
\qquad\qquad \wedge\, cmdLog' = [cmdLog \text{ EXCEPT } ![replica] = @ \cup\\
\qquad\qquad\qquad\quad \{[inst \;\mapsto msg.inst,\\
\qquad\qquad\qquad\quad\; status \mapsto \text{“not-seen”},\\
\qquad\qquad\qquad\quad\; ballot \mapsto msg.ballot,\\
\qquad\qquad\qquad\quad\; cmd \;\mapsto none,\\
\qquad\qquad\qquad\quad\; deps \;\mapsto \{\},\\
\qquad\qquad\qquad\quad\; seq \;\mapsto 0]\}]\\
\qquad\qquad \wedge \text{UNCHANGED } \langle proposed,\, executed,\, committed,\, crtInst,\, ballots,\\
\qquad\qquad\qquad\quad leaderOfInst,\, preparing\rangle
\end{array}
$$

$$
\begin{array}{l}
PrepareFinalize(replica,\, i,\, Q) \;\triangleq\\
\quad \wedge\, i \in preparing[replica]\\
\quad \wedge\, \exists\, rec \in cmdLog[replica] :\\
\qquad \wedge\, rec.inst = i\\
\qquad \wedge\, rec.status \notin \{\text{“committed”},\, \text{“executed”}\}\\
\qquad \wedge \text{LET } replies \triangleq \{msg \in sentMsg :\\
\qquad\qquad\qquad\quad \wedge\, msg.inst = i\\
\qquad\qquad\qquad\quad \wedge\, msg.type = \text{“prepare-reply”}\\
\qquad\qquad\qquad\quad \wedge\, msg.dst = replica\\
\qquad\qquad\qquad\quad \wedge\, msg.ballot = rec.ballot\} \text{IN}\\
\qquad\quad \wedge\, (\forall\, rep \in Q : \exists\, msg \quad\; \in replies : msg.src = rep)\\
\qquad\quad \wedge \quad \vee\, \exists\, com \in replies :\\
\qquad\qquad\qquad\quad \wedge\, (com.status \in \{\text{“committed”},\, \text{“executed”}\})\\
\qquad\qquad\qquad\quad \wedge\, preparing' = [preparing \text{ EXCEPT } ![replica] = @ \setminus \{i\}]\\
\qquad\qquad\qquad\quad \wedge\, sentMsg' = sentMsg \setminus replies\\
\qquad\qquad\qquad\quad \wedge \text{UNCHANGED } \langle cmdLog,\, proposed,\, executed,\, crtInst,\, leaderOfInst,\\
\qquad\qquad\qquad\qquad\qquad committed,\, ballots\rangle\\
\qquad\qquad\quad \vee\, \wedge\, \neg(\exists\, msg \in replies : msg.status \in \{\text{“committed”},\, \text{“executed”}\})\\
\qquad\qquad\qquad\quad \wedge\, \exists\, acc \in replies :\\
\qquad\qquad\qquad\qquad\quad \wedge\, acc.status = \text{“accepted”}\\
\qquad\qquad\qquad\qquad\quad \wedge\, (\forall\, msg \in (replies \setminus \{acc\}) :\\
\qquad\qquad\qquad\qquad\qquad\quad (msg.prev\_ballot[1] \leq acc.prev\_ballot[1] \vee\\
\qquad\qquad\qquad\qquad\qquad\quad\; msg.status \neq \text{“accepted”}))\\
\qquad\qquad\qquad\qquad\quad \wedge\, sentMsg' = (sentMsg \setminus replies) \cup\\
\qquad\qquad\qquad\qquad\qquad\quad [type \;: \{\text{“accept”}\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; src \;: \{replica\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; dst \;: Q \setminus \{replica\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; inst \;: \{i\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; ballot : \{rec.ballot\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; cmd \;: \{acc.cmd\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; deps \;: \{acc.deps\},\\
\qquad\qquad\qquad\qquad\qquad\quad\; seq \;: \{acc.seq\}]\\
\qquad\qquad\qquad\qquad\quad \wedge\, cmdLog' = [cmdLog \text{ EXCEPT } ![replica] = (@ \setminus \{rec\}) \cup
\end{array}
$$

$$
\begin{array}{l}
\quad \{[inst \quad \mapsto i, \\
\quad\ \ status \mapsto \text{“accepted”}, \\
\quad\ \ ballot \ \mapsto rec.ballot, \\
\quad\ \ cmd \quad \mapsto acc.cmd, \\
\quad\ \ deps \quad \mapsto acc.deps, \\
\quad\ \ seq \quad\ \mapsto acc.seq]\} \\
\quad \land preparing' = [preparing \ \text{EXCEPT} \ ![replica] = @ \setminus \{i\}] \\
\quad \land leaderOfInst' = [leaderOfInst \ \text{EXCEPT} \ ![replica] = @ \cup \{i\}] \\
\quad \land \text{UNCHANGED} \ \langle proposed, \ executed, \ crtInst, \ committed, \ ballots \rangle \\
\lor \ \land \neg(\exists \, msg \in replies : \\
\quad\ \ msg.status \in \{\text{“accepted”}, \ \text{“committed”}, \ \text{“executed”}\}) \\
\quad \land \text{LET} \ preaccepts \triangleq \{msg \in replies : msg.status = \text{“pre-accepted”}\} \ \text{IN} \\
\quad\ \ (\lor \ \land \forall \, p1, \ p2 \in preaccepts : \\
\qquad\qquad\ \ p1.cmd = p2.cmd \land p1.deps = p2.deps \land p1.seq = p2.seq \\
\qquad\ \ \land \neg(\exists \, pl \in preaccepts : pl.src \quad\ \ = i[1]) \\
\qquad\ \ \land Cardinality(preaccepts) \geq Cardinality(Q) - 1 \\
\qquad\ \ \land \text{LET} \ pac \triangleq \text{CHOOSE} \ pac \in preaccepts : \text{TRUE IN} \\
\qquad\qquad \land \ \ sentMsg' = (sentMsg \setminus replies) \cup \\
\qquad\qquad\qquad\ \ [type \ : \{\text{“accept”}\}, \\
\qquad\qquad\qquad\ \ src \ : \{replica\}, \\
\qquad\qquad\qquad\ \ dst \ : Q \setminus \{replica\}, \\
\qquad\qquad\qquad\ \ inst \ : \{i\}, \\
\qquad\qquad\qquad\ \ ballot : \{rec.ballot\}, \\
\qquad\qquad\qquad\ \ cmd \ : \{pac.cmd\}, \\
\qquad\qquad\qquad\ \ deps \ : \{pac.deps\}, \\
\qquad\qquad\qquad\ \ seq \ : \{pac.seq\}] \\
\qquad\qquad \land \ \ cmdLog' = [cmdLog \ \text{EXCEPT} \ ![replica] = (@ \setminus \{rec\}) \cup \\
\qquad\qquad\qquad\ \ \{[inst \quad \mapsto i, \\
\qquad\qquad\qquad\qquad status \mapsto \text{“accepted”}, \\
\qquad\qquad\qquad\qquad ballot \ \mapsto rec.ballot, \\
\qquad\qquad\qquad\qquad cmd \quad \mapsto pac.cmd, \\
\qquad\qquad\qquad\qquad deps \quad \mapsto pac.deps, \\
\qquad\qquad\qquad\qquad seq \quad\ \mapsto pac.seq]\}] \\
\qquad\qquad\ \ \land preparing' = [preparing \ \text{EXCEPT} \ ![replica] = @ \setminus \{i\}] \\
\qquad\qquad\ \ \land leaderOfInst' = [leaderOfInst \ \text{EXCEPT} \ ![replica] = @ \cup \{i\}] \\
\qquad\qquad\ \ \land \text{UNCHANGED} \ \langle proposed, \ executed, \ crtInst, \ committed, \ ballots \rangle \\
\qquad\ \ \lor \ \land \forall \, p1, \ p2 \in preaccepts : p1.cmd = p2.cmd \land \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ p1.deps = p2.deps \land \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ p1.seq = p2.seq \\
\qquad\qquad\ \ \land \neg(\exists \, pl \in preaccepts : pl.src = i[1]) \\
\qquad\qquad\ \ \land Cardinality(preaccepts) < Cardinality(Q) - 1 \\
\qquad\qquad\ \ \land Cardinality(preaccepts) \geq Cardinality(Q) \div 2 \\
\qquad\qquad\ \ \land \text{LET} \ pac \triangleq \text{CHOOSE} \ pac \in preaccepts : \text{TRUE IN} \\
\qquad\qquad\qquad \land \ \ sentMsg' = (sentMsg \setminus replies) \cup \\
\qquad\qquad\qquad\qquad\ \ [type \ : \{\text{“try-pre-accept”}\},
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\quad src \quad : \{replica\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad dst \quad : Q,\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad inst \quad : \{i\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad ballot : \{rec.ballot\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad cmd \quad : \{pac.cmd\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad deps \quad : \{pac.deps\},\\
\qquad\qquad\qquad\qquad\qquad\qquad\quad seq \quad : \{pac.seq\}]\\
\qquad\qquad\qquad\qquad\qquad\quad \land\ preparing' = [preparing \text{ EXCEPT } ![replica] = @ \setminus \{i\}]\\
\qquad\qquad\qquad\qquad\qquad\quad \land\ leaderOfInst' = [leaderOfInst \text{ EXCEPT } ![replica] = @ \cup \{i\}]\\
\qquad\qquad\qquad\qquad\qquad\quad \land\ \text{UNCHANGED } \langle cmdLog,\ proposed,\ executed,\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad crtInst,\ committed,\ ballots\rangle\\
\qquad\qquad\qquad\lor\ \land\ \lor \exists\, p1,\ p2 \in preaccepts : p1.cmd \neq p2.cmd \lor\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p1.deps \neq p2.deps \lor\\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad p1.seq \neq p2.seq\\
\qquad\qquad\qquad\qquad\lor \exists\, pl \in preaccepts : pl.src = i[1]\\
\qquad\qquad\qquad\qquad\lor Cardinality(preaccepts) < Cardinality(Q) \div 2\\
\qquad\qquad\qquad\land\ preaccepts \neq \{\}\\
\qquad\qquad\qquad\land\ \text{LET } pac \triangleq \text{ CHOOSE } pac \in preaccepts : pac.cmd \neq none \text{IN}\\
\qquad\qquad\qquad\qquad\land\ StartPhase1(pac.cmd,\ replica,\ Q,\ i,\ rec.ballot,\ replies)\\
\qquad\qquad\qquad\qquad\land\ preparing' = [preparing \text{ EXCEPT } ![replica] = @ \setminus \{i\}]\\
\qquad\qquad\qquad\qquad\land\ \text{UNCHANGED } \langle proposed,\ executed,\ crtInst,\ committed,\ ballots\rangle)\\
\qquad\qquad\lor\ \land \forall\, msg \in replies : msg.status = \text{“not-seen”}\\
\qquad\qquad\qquad\land\ StartPhase1(none,\ replica,\ Q,\ i,\ rec.ballot,\ replies)\\
\qquad\qquad\qquad\land\ preparing' = [preparing \text{ EXCEPT } ![replica] = @ \setminus \{i\}]\\
\qquad\qquad\qquad\land\ \text{UNCHANGED } \langle proposed,\ executed,\ crtInst,\ committed,\ ballots\rangle
\end{array}
$$

$$
\begin{array}{l}
ReplyTryPreaccept(replica) \triangleq\\
\quad \exists\, tpa \in sentMsg :\\
\qquad \land\ tpa.type = \text{“try-pre-accept”}\\
\qquad \land\ tpa.dst = replica\\
\qquad \land\ \text{LET } oldRec \triangleq \{rec \in cmdLog[replica] : rec.inst = tpa.inst\} \text{IN}\\
\qquad\qquad \land\ \forall\, rec \in oldRec : rec.ballot[1] \leq tpa.ballot[1] \land\\
\qquad\qquad\qquad\qquad rec.status \notin \{\text{“accepted”},\ \text{“committed”},\ \text{“executed”}\}\\
\qquad\qquad \land\ \lor (\exists\, rec \in cmdLog[replica] \setminus oldRec :\\
\qquad\qquad\qquad\qquad \land\ tpa.inst \notin rec.deps\\
\qquad\qquad\qquad\qquad \land\ \lor rec.inst \notin tpa.deps\\
\qquad\qquad\qquad\qquad\quad\ \lor rec.seq \geq tpa.seq\\
\qquad\qquad\qquad\qquad \land\ sentMsg' = (sentMsg \setminus \{tpa\}) \cup\\
\qquad\qquad\qquad\qquad\qquad\qquad \{[type \ \mapsto \text{“try-pre-accept-reply”},\\
\qquad\qquad\qquad\qquad\qquad\qquad\ src \ \mapsto replica,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ dst \ \mapsto tpa.src,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ inst \ \mapsto tpa.inst,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ ballot \mapsto tpa.ballot,\\
\qquad\qquad\qquad\qquad\qquad\qquad\ status \mapsto rec.status]\})\\
\qquad\qquad\qquad\qquad \land\ \text{UNCHANGED } \langle cmdLog,\ proposed,\ executed,\ committed,\ crtInst,
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ballots,\ leaderOfInst,\ preparing\rangle \\
\lor\ \land\ (\forall\, rec \in cmdLog[replica] \setminus oldRec : \\
\qquad\qquad tpa.inst \in rec.deps \lor (rec.inst \in tpa.deps \land \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad rec.seq < tpa.seq)) \\
\qquad \land\ sentMsg' = (sentMsg \setminus \{tpa\}) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \{[type \ \mapsto \text{“try-pre-accept-reply”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ src \ \mapsto replica, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ dst \ \mapsto tpa.src, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ inst \ \mapsto tpa.inst, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ballot \mapsto tpa.ballot, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto \text{“OK”}]\} \\
\qquad \land\ cmdLog' = [cmdLog \ \textsc{except}\ ![replica] = (@ \setminus oldRec) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \ \mapsto tpa.inst, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto \text{“pre-accepted”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ballot \mapsto tpa.ballot, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ cmd \ \mapsto tpa.cmd, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ deps \ \mapsto tpa.deps, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ seq \ \mapsto tpa.seq]\}] \\
\qquad \land\ \textsc{unchanged}\ \langle proposed,\ executed,\ committed,\ crtInst,\ ballots, \\
\qquad\qquad\qquad\qquad\qquad\qquad leaderOfInst,\ preparing\rangle
\end{array}
$$

$$
\begin{array}{l}
FinalizeTryPreAccept(cleader,\ i,\ Q) \ \triangleq \\
\quad \exists\, rec \in cmdLog[cleader] : \\
\qquad \land\ rec.inst = i \\
\qquad \land\ \textsc{let}\ tprs \ \triangleq\ \{msg \in sentMsg : msg.type = \text{“try-pre-accept-reply”} \land \\
\qquad\qquad\qquad\qquad\qquad\qquad msg.dst = cleader \land msg.inst = i \land \\
\qquad\qquad\qquad\qquad\qquad\qquad msg.ballot = rec.ballot\}\textsc{in} \\
\qquad\quad \land\ \ \forall\, r \in Q : \exists\, tpr \in tprs : tpr.src = r \\
\qquad\quad \land\ \ \lor \land \forall\, tpr \in tprs : tpr.status = \text{“OK”} \\
\qquad\qquad\qquad \land\ sentMsg' = (sentMsg \setminus tprs) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad [type \ : \{\text{“accept”}\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ src \ \ : \{cleader\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ dst \ \ : Q \setminus \{cleader\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ inst \ \ : \{i\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ballot : \{rec.ballot\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ cmd \ \ : \{rec.cmd\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ deps \ \ : \{rec.deps\}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ seq \ \ : \{rec.seq\}] \\
\qquad\qquad\qquad \land\ cmdLog' = [cmdLog \ \textsc{except}\ ![cleader] = (@ \setminus \{rec\}) \cup \\
\qquad\qquad\qquad\qquad\qquad\qquad \{[inst \ \mapsto i, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ status \mapsto \text{“accepted”}, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ ballot \mapsto rec.ballot, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ cmd \ \mapsto rec.cmd, \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \ deps \ \mapsto rec.deps,
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad seq \quad \mapsto rec.seq]\}] \\
\quad \land \text{UNCHANGED } \langle proposed,\ executed,\ committed,\ crtInst,\ ballots, \\
\qquad\qquad\qquad\qquad leaderOfInst,\ preparing\rangle \\
\lor \ \land \exists\, tpr \in tprs : tpr.status \in \{\text{“accepted”},\ \text{“committed”},\ \text{“executed”}\} \\
\quad \land StartPhase1(rec.cmd,\ cleader,\ Q,\ i,\ rec.ballot,\ tprs) \\
\quad \land \text{UNCHANGED } \langle proposed,\ executed,\ committed,\ crtInst,\ ballots, \\
\qquad\qquad\qquad\qquad leaderOfInst,\ preparing\rangle \\
\lor \ \land \exists\, tpr \in tprs : tpr.status = \text{“pre-accepted”} \\
\quad \land \forall\, tpr \in tprs : tpr.status \in \{\text{“OK”},\ \text{“pre-accepted”}\} \\
\quad \land sentMsg' = sentMsg \setminus tprs \\
\quad \land leaderOfInst' = [leaderOfInst \text{ EXCEPT } ![cleader] = @ \setminus \{i\}] \\
\quad \land \text{UNCHANGED } \langle cmdLog,\ proposed,\ executed,\ committed,\ crtInst, \\
\qquad\qquad\qquad\qquad ballots,\ preparing\rangle
\end{array}
$$

Action groups

$$
\begin{array}{l}
CommandLeaderAction \ \triangleq \\
\quad \lor (\exists\, C \in (Commands \setminus proposed) : \\
\qquad \exists\, cleader \in Replicas : Propose(C,\ cleader)) \\
\quad \lor (\exists\, cleader \in Replicas : \exists\, inst \in leaderOfInst[cleader] : \\
\qquad \lor (\exists\, Q \in FastQuorums(cleader) : Phase1Fast(cleader,\ inst,\ Q)) \\
\qquad \lor (\exists\, Q \in SlowQuorums(cleader) : Phase1Slow(cleader,\ inst,\ Q)) \\
\qquad \lor (\exists\, Q \in SlowQuorums(cleader) : Phase2Finalize(cleader,\ inst,\ Q)) \\
\qquad \lor (\exists\, Q \in SlowQuorums(cleader) : FinalizeTryPreAccept(cleader,\ inst,\ Q)))
\end{array}
$$

$$
\begin{array}{l}
ReplicaAction \ \triangleq \\
\quad \exists\, replica \in Replicas : \\
\quad\ ( \lor Phase1Reply(replica) \\
\quad\ \lor Phase2Reply(replica) \\
\quad\ \lor \exists\, cmsg \in sentMsg : (cmsg.type = \text{“commit”} \land Commit(replica,\ cmsg)) \\
\quad\ \lor \exists\, i \in Instances : \\
\qquad \land crtInst[i[1]] > i[2] \quad \text{This condition states that the instance has} \\
\qquad\qquad\qquad\qquad\qquad\quad \text{been started by its original owner} \\
\qquad \land \exists\, Q \in SlowQuorums(replica) : SendPrepare(replica,\ i,\ Q) \\
\quad\ \lor ReplyPrepare(replica) \\
\quad\ \lor \exists\, i \ \in preparing[replica] : \\
\qquad \exists\, Q \in SlowQuorums(replica) : PrepareFinalize(replica,\ i,\ Q) \\
\quad\ \lor ReplyTryPreaccept(replica))
\end{array}
$$

Next action

$$
\begin{array}{l}
Next \ \triangleq \\
\quad \lor CommandLeaderAction \\
\quad \lor ReplicaAction
\end{array}
$$

The complete definition of the algorithm

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

Theorems

$$\begin{array}{l}
Nontriviality \triangleq \\
\quad \forall\, i \in Instances : \\
\qquad \Box(\forall\, C \in committed[i] : C \in proposed \lor C = none)
\end{array}$$

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

$$\begin{array}{l}
Consistency \triangleq \\
\quad \forall\, i \in Instances : \\
\qquad \Box(Cardinality(committed[i]) \leq 1)
\end{array}$$

THEOREM $Spec \Rightarrow (\Box\, TypeOK) \land Nontriviality \land Stability \land Consistency$