Making Democracy Work: Fixing and Simplifying Egalitarian Paxos

Fedor Ryabinin
IMDEA Software Institute, Madrid, Spain

Alexey Gotsman
IMDEA Software Institute, Madrid, Spain

Pierre Sutra
Institut Polytechnique de Paris, Palaiseau, France

Abstract

Classical state-machine replication protocols, such as Paxos, rely on a distinguished leader process to order commands. Unfortunately, this approach makes the leader a single point of failure and increases the latency for clients that are not co-located with it. As a response to these drawbacks, *Egalitarian Paxos* [19] introduced an alternative, leaderless approach, that allows replicas to order commands collaboratively. Not relying on a single leader allows the protocol to maintain non-zero throughput with up to f crashes of *any* processes out of a total of $n = 2f + 1$. The protocol furthermore allows any process to execute a command c fast, in 2 message delays, provided no more than $e = \lceil \frac{f+1}{2} \rceil$ other processes fail, and all concurrently submitted commands commute with c; the latter condition is often satisfied in practical systems.

Egalitarian Paxos has served as a foundation for many other replication protocols. But unfortunately, the protocol is very complex, ambiguously specified and suffers from nontrivial bugs. In this paper, we present EPaxos* – a simpler and correct variant of Egalitarian Paxos. Our key technical contribution is a simpler failure-recovery algorithm, which we have rigorously proved correct. Our protocol also generalizes Egalitarian Paxos to cover the whole spectrum of failure thresholds f and e such that $n \geq \max\{2e + f - 1, 2f + 1\}$ – the number of processes that we show to be optimal.

2012 ACM Subject Classification Theory of computation → Distributed computing models

Keywords and phrases Consensus, state-machine replication, fault tolerance

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2025.22

Related Version *Extended Version*: `http://arxiv.org/abs/2511.02743` [27]

Funding This work was partially supported by the projects BYZANTIUM, funded by MCIN/AEI and NextGenerationEU/PRTR; DECO and María de Maeztu, funded by MICIU/AEI; and REDONDA, funded by the EU CHIST-ERA network.

Acknowledgements We thank Benedict Elliott Smith, whose work on leaderless consensus in Apache Cassandra inspired this paper.

1 Introduction

State-machine replication (SMR) is a classic approach to implementing fault-tolerant services [11, 28]. In this approach, the service is defined by a deterministic state machine, and each process maintains its own local replica of the machine. An SMR protocol coordinates the execution of client commands at the processes to ensure that the system is *linearizable* [10], i.e., behaves as if commands are executed sequentially by a single process. Classic protocols for implementing SMR under partial synchrony, such as Paxos [12] and Raft [22], rely on a distinguished leader process that determines the order of command execution. Unfortunately, this approach makes the leader a single point of failure and increases the latency for clients that are not co-located with it.

29th International Conference on Principles of Distributed Systems (OPODIS 2025).
Editors: Andrei Arusoaie, Emanuel Onica, Michael Spear, and Sara Tucci-Piergiovanni; Article No. 22; pp. 22:1–22:19
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

As a response to these drawbacks, *Egalitarian Paxos* (EPaxos) [19] introduced a leaderless approach that allows processes to order commands collaboratively. This more democratic ordering mechanism brings several benefits, the first of which is faster command execution. In more detail, as is standard, EPaxos tolerates up to f processes crashing out of a total of $n = 2f + 1$. But when there are no more than $\lceil \frac{f+1}{2} \rceil$ crashes and the system is synchronous, the protocol additionally allows any correct process to execute a command c in only 2 message delays, provided c commutes with all concurrently submitted commands; the latter condition is often satisfied in practice. In this case the protocol takes a *fast path*, which contacts *a fast quorum* of $n - \lceil \frac{f+1}{2} \rceil = f + \lfloor \frac{f+1}{2} \rfloor$ processes. In contrast, in Paxos, a non-leader process requires at least 3 messages delays to execute a command, as it has to route the command through the leader. Furthermore, a leader failure prevents Paxos from executing any commands, no matter quickly or slowly, until a new leader is elected. EPaxos can maintain non-zero throughput with up to f failures of *any* processes – a practically important feature for highly available systems.

Interestingly, in comparison to existing SMR protocols with fast paths [13, 23], EPaxos can provide fast decisions while tolerating an extra process failure. For example, Generalized Paxos [13] decides fast when the number of failures does not exceed $\lfloor \frac{f}{2} \rfloor < \lfloor \frac{f}{2} \rfloor + 1 = \lceil \frac{f+1}{2} \rceil$. Thus, for $n = 3$ and $n = 5$, EPaxos provides fast decisions while tolerating any minority of failures, with its fast path accessing the remaining majority – same as for slow decisions in the usual Paxos. Providing fast decisions in these cases does not require contacting larger quorums. The above advantages are no small change in practical systems, especially those deployed over wide-area networks. In this setting, the number of replicas is typically small (e.g., $n \in \{3, 5\}$ [6]), while the latencies are high. Thus, waiting for an extra message or contacting an additional far-away replica may cost hundreds of milliseconds per command.

Due to these considerations, EPaxos has been influential: it has served as a foundation for alternative leaderless SMR protocols [3, 8, 9], protocols for distributed transactions and storage [4, 20, 21], and performance-evaluation studies [31]. EPaxos has also motivated the study of theoretical frameworks for leaderless consensus [2] and SMR [16, 25]. More recently, it inspired a new protocol for strongly consistent transactions in Apache Cassandra, a popular NoSQL datastore [29]. Overall, the original SOSP'13 paper on EPaxos [19] has been cited more than 500 times. Surprisingly, despite all this impact, EPaxos remains poorly understood and is subject to controversies. As we explain next, this is because the protocol is complex, ambiguously specified, and suffers from nontrivial bugs.

Complexity. At a high level, EPaxos implements SMR by getting the processes to agree on a *dependency graph* – a directed graph whose vertices are commands and edges represent constraints on their execution order. Processes then execute commands in an order determined by the graph. To ensure the agreement on the dependency graph, the process that submits a particular command drives the consensus on the set of predecessors of the command in the graph. If this initial command *coordinator* fails, another process takes over its job. This failure recovery procedure is the most subtle part of the protocol.

The original EPaxos paper [19] describes in detail only a simplified version of the protocol, which has fast quorums of size $n - 1 = 2f$, but in exchange, has a simple failure recovery procedure. The paper only sketches the full version of the protocol with fast quorums of size $f + \lfloor \frac{f+1}{2} \rfloor$, where recovery is much more complex. Unfortunately, more detailed descriptions of this recovery procedure available in technical reports [17, 18] are ambiguous and incomplete. The TLA$^+$ specification of the protocol [17] does not cover recovery in full, and the open-source implementation [1] is not aligned with the available protocol descriptions.

Bugs. As shown in [30], EPaxos has a bug in its use of Paxos-like ballot variables. The usual Paxos partitions the execution of the protocol into *ballots*, which determine the current leader [12]. Each Paxos process uses two variables storing ballot numbers – the ballot the process currently participates in, and the one where it last voted. EPaxos uses a similar concept of ballots for each single command, to determine which process is responsible for computing this command's position in the execution order. However, the protocol uses only a single ballot variable, which may lead to non-linearizable executions [30]. This bug can be easily fixed for the simple version of EPaxos with fast quorums of size $n-1$. But as we explain in §6, fixing it for the full version of the protocol with fast quorums of size $f+\lfloor\frac{f+1}{2}\rfloor$ is nontrivial, because in this case recovery is much more complex than in Paxos. We have also found a new bug where the recovery can get deadlocked even during executions with finitely many submitted commands (§6 and [27, §E]).

Lack of rigorous proofs. The available correctness proofs for the version of EPaxos with smaller fast quorums [17, §3.6.5] are not rigorous: they make arguments without referring to the actual pseudocode of the protocol and often posit claims without sufficient justification. More rigor may have permitted to find that the protocol is incorrect.

Moreover, the existing proofs are given under an assumption that a process submitting a command selects a fast quorum it will use a priori, and communicates only with this quorum. This version of the protocol, which the authors call *thrifty*, takes the fast path more frequently in failure-free scenarios and is simpler to prove correct. But it does not allow a process to take the fast path even with a single failure, if this failure happens to be within the selected fast quorum. The non-thrifty version of EPaxos is more robust. However, it is underspecified, and the authors of EPaxos never proved its correctness.

Contributions. We are thus left in an uncomfortable situation where a protocol underlying a significant body of research suffers from multiple problems. In this paper we make the following contributions to improve this situation:

- *Clarity.* We propose a simpler and correct variant of Egalitarian Paxos, which we call EPaxos*. Our protocol is explained pedagogically, starting from key notions and invariants. The main technical novelty of EPaxos* is its failure recovery procedure, which significantly simplifies the one in the original EPaxos and allows us to fix its bugs. At the same time, EPaxos* does not significantly change the failure-free processing of the original protocol, thus preserving its steady-state performance properties.
- *Correctness.* We rigorously demonstrate the correctness of EPaxos*, proving key properties as we explain the protocol, and covering the rest of the proof in [27, §D]. We give proofs for both non-thrifty and thrifty versions of the protocol.
- *Optimality.* We generalize the original EPaxos to handle a broader spectrum of failure thresholds under which it can execute commands fast (e) or execute any commands at all (f). To this end, we formalize the desired fault-tolerance properties by a notion of an *f-resilient e-fast* SMR protocol, and show that EPaxos* satisfies it when $n \geq \max\{2e+f-1, 2f+1\}$. Building on our recent result about fast consensus [26], we also show that this number of processes is optimal. The original EPaxos aims to handle the special case of this inequality when $f=\lfloor\frac{n-1}{2}\rfloor$ and $e=\lceil\frac{f+1}{2}\rceil$, but falls short due to its bugs. Our version is thus the first provably correct SMR protocol with optimal values of e and f. Given a fixed n, it permits trading off lower overall fault tolerance (f) for higher fault tolerance of the fast path (e).

2 Fast State-Machine Replication

We consider a set Π of $n \geq 3$ processes communicating via reliable links. At most f of the processes can crash. The system is partially synchronous [7]: after some global stabilization time (GST) messages take at most Δ units of time to reach their destinations. The bound Δ is known to the processes, while GST is unknown. Processes are equipped with clocks that can drift unboundedly from real time before GST, but can accurately measure time thereafter.

State-machine replication (SMR) is a common approach to implementing fault-tolerant services in this system [28]. A service is defined by a deterministic state machine with an appropriate set of commands. Processes maintain their own local copy of the state machine and accept commands from clients (external to the system). An *SMR protocol* coordinates the execution of commands at the processes, ensuring that service replicas stay in sync. The protocol provides an operation `submit`(c), which allows a process to submit a command c for execution on behalf of a client. The protocol may also trigger an event `execute`(c) at a process, asking it to apply c to the local service replica; after execution, the process that submitted the command may return the outcome of c to the client. Without loss of generality, we assume that each submitted command is unique.

A fault-tolerant service implemented using SMR is supposed to be *linearizable* [10]. Informally, this means that commands appear as if executed sequentially on a single copy of the state machine in an order consistent with the real-time order, i.e., the order of non-overlapping command invocations. As observed in [13, 23] for the replicated service to be linearizable, the SMR protocol does not need to ensure that commands are executed at processes in the exact same order: it is enough to agree on the order of non-commuting commands. Formally, two commands c and c' *commute* if in every state s of the state machine: *(i)* executing c followed by c' or c' followed by c in s leads to the same state; and *(ii)* c returns the same response in s as in the state obtained by executing c' in s, and vice versa. If c and c' do not commute, we say that they *conflict* and write $c \bowtie c'$. Then the specification of the SMR protocol sufficient for linearizability is given by the following properties:

Validity. If a process executes a command c, then some process has submitted c before.

Integrity. A process executes each command at most once.

Consistency. Processes execute conflicting commands in the same order.

Liveness. In any execution with finitely many submitted commands, if a command is submitted by a correct process or executed at some process, then it is eventually executed at all correct processes.

Note that Egalitarian Paxos is not guaranteed to terminate when infinitely many commands are submitted [8], hence the extra assumption in Liveness.

We next define a class of SMR protocols that can execute commands fast under favorable conditions. One such condition is that the system is synchronous in the following sense. We say that the events that happen during the time interval $[0, \Delta)$ form *the first round*, the events that happen during the time interval $[\Delta, 2\Delta)$ *the second round*, and so on.

▶ **Definition 1.** *Given $E \subseteq \Pi$, a run is **E-faulty synchronous**, if:*

- *All processes in $\Pi \setminus E$ are correct, and all processes in E are faulty.*
- *Processes in E crash at the beginning of the first round.*
- *All messages sent during a round are delivered at the beginning of the next round.*
- *All local computations are instantaneous.*

Another condition for a command to be executed fast is that there are no concurrent conflicting commands, as defined in the following.

▶ **Definition 2.** *We say that c **precedes** c' in a run if c' is submitted at a process when this process has already executed c. Commands c and c' are **concurrent** when neither precedes the other. A command c is **conflict-free** if it does not conflict with any concurrent command.*

▶ **Definition 3.** *A run of an SMR protocol is **fast** when any conflict-free command submitted at time t by a process p gets executed at p by time $t+2\Delta$. A protocol is **e-fast** if for all $E \subseteq \Pi$ of size e, every E-faulty synchronous run of the protocol is fast. A protocol is **f-resilient** if it implements SMR under at most f process failures.*

Thus, e defines the maximal number of failures the protocol can tolerate while executing commands fast, and f the failures it can tolerate while executing any commands at all (we assume $e \leq f$). These thresholds are subject to a trade-off stated by the following theorem, which follows from our recent result about consensus [26]. We defer its proof to [27, §A].

▶ **Theorem 4.** *Any f-resilient e-fast SMR protocol requires $n \geq \max\{2e+f-1, 2f+1\}$.*

We compare this lower bound with related ones in §7. In classic SMR protocols, such as Paxos [12], commands submitted at the leader execute within 2Δ in any $\lfloor \frac{n-1}{2} \rfloor$-faulty synchronous run, but commands submitted at other processes do not enjoy such a low latency. Hence, these "undemocratic" protocols are not e-fast for any e. EPaxos aims to be $\lceil \frac{f+1}{2} \rceil$-fast for $n = 2f+1 = 2e+f-1$, which would match the bound in Theorem 4. Unfortunately, as we explained in §1, the corresponding version of the protocol is not well-specified and suffers from nontrivial bugs. In the rest of the paper we present EPaxos* – a simpler and correct variant of Egalitarian Paxos. EPaxos* is the first provably correct SMR protocol that matches the lower bound in Theorem 4. It furthermore generalizes Egalitarian Paxos to cover the whole spectrum of parameters e and f allowed by the theorem: e.g., for a fixed n it allows decreasing f in exchange for increasing e. For instance, when $n \geq 5$ we can let $e = f = \lfloor \frac{n+1}{3} \rfloor$, which maximizes e to the extent possible.

3 EPaxos*: Baseline Protocol for $n \geq \max\{2e+f+1, 2f+1\}$

To simplify the presentation, we explain EPaxos* in two stages. In this section we present a simpler version of the protocol that requires $n \geq \max\{2e+f+1, 2f+1\}$ – same as in Generalized Paxos [13]. In the next section we explain how to reduce the number of processes to $n \geq \max\{2e+f-1, 2f+1\}$, so that the protocol matches the lower bound in Theorem 4.

EPaxos* implements SMR by getting the processes to agree on a *dependency graph* – a directed graph whose vertices are application commands and edges represent constraints on their execution order. We say that a command c is a *dependency* of c' when (c, c') is an edge in the graph. Processes progressively agree on the same dependency graph by agreeing on the set of dependencies of each vertex. Processes then execute commands in an order determined by the graph. The graph may contain cycles, which are broken deterministically as we describe in the following.

In EPaxos*, each submitted command is associated with a unique numerical *identifier*. An array `cmd` at each process maps command identifiers to their actual payloads. Processes also maintain local copies of the dependency graph in an array `dep`, which maps the identifier of a command to the set of identifiers of its dependencies (a *dependency set*). A command *id* goes through several *phases* at a given process – INITIAL, PREACCEPTED, ACCEPTED and COMMITTED – with the COMMITTED phase indicating that consensus has been reached on

```
1 while true
2     let G ⊆ dep be the largest subgraph such that ∀id ∈ G. phase[id] = COMMITTED ∧ dep[id] ⊆ G
3     for C ∈ SCC(G) in topological order do
4         for id ∈ C in the order of command identifiers do
5             if id ∉ executed ∧ cmd[id] ≠ Nop then
6                 execute(cmd[id])
7                 executed ← executed ∪ {id}
```

Figure 1 EPaxos*: execution protocol.

the values of cmd[id] and dep[id]. An array phase maps command identifiers to their current phases (by default, INITIAL). A command is *pre-accepted*, *accepted*, or *committed* when its identifier is in the corresponding phase.

In case of failures, the protocol may end up committing a command id with a special Nop payload that is not executed by the protocol and conflicts with all commands (so in the end cmd[id] = Nop at all processes). In this case, the command should be resubmitted (for simplicity, we omit the corresponding mechanism from the protocol description). The protocol ensures the following key invariants:

▶ **Invariant 1** (Agreement)**.** If a command id is committed at two processes with dependency sets D and D' and payloads c and c' then $D = D'$ and $c = c'$.

▶ **Invariant 2** (Visibility)**.** If distinct commands id and id' are both committed with conflicting non-Nop payloads and dependency sets D and D', respectively, then either $id \in D'$ or $id' \in D$.

These invariants are enough to ensure that conflicting commands are executed in the same order at all processes, as we describe next.

3.1 Execution Protocol

Each process runs a background task that executes commands after they are committed (Figure 1). At a high level, the execution protocol works as follows. A command is ready to be executed once it is committed and so are all its transitive dependencies. When this happens, the protocol executes all ready commands in a batch in an order consistent with the dependency graph, breaking cycles using command identifiers. The identifiers of the executed commands are added to an executed variable. More concretely, the execution protocol performs the following steps.

1. Compute the largest subgraph G of the dependency graph consisting of committed commands that transitively depend only on other committed commands (line 2).
2. Split G into strongly connected components and sort them in the topological order (line 3).
3. Execute the commands in each component in the identifier order, skipping Nops (lines 4–7).

▶ **Example 5.** Consider four commands c_1, c_2, c_3 and c_4, where c_1, c_2 and c_3 pairwise conflict, while c_4 conflicts only with c_1. Assume the commands are committed with dependencies that yield the graph in Figure 2, where $c \to c'$ means that c is a dependency of c'. When the protocol in Figure 1 is executed on this graph, it will consider its three strongly connected components: $\{c_1\}$, $\{c_2, c_3\}$, and $\{c_4\}$. These can be topologically ordered in two ways: $\{c_1\} \to \{c_2, c_3\} \to \{c_4\}$ and $\{c_1\} \to \{c_4\} \to \{c_2, c_3\}$. The protocol will execute commands following one of these orders, breaking ties within strongly connected components using

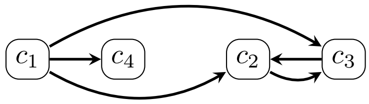

Figure 2 Example of a dependency graph.

command identifiers. Thus, one process can use the order c_1, c_2, c_3, c_4, while the other can use c_1, c_4, c_2, c_3. This does not violate Consistency because c_4 does not conflict with c_2 or c_3, and thus can be executed in either order with respect to them.

It is easy to show that, if Invariants 1-2 hold, then the protocol in Figure 1 ensures the Consistency property of SMR (§2); we defer a formal proof to [27, §D]. Informally, Invariant 1 ensures that processes agree on the same dependency graph, while Invariant 2 that the graph relates any two conflicting commands by at least one edge. Thus, processes will use the same constraints to determine the order of execution for conflicting commands.

The original EPaxos protocol includes an optimization that speeds up the execution of commands that do not return a value: a process that submits such a command returns to the client as soon as the command is committed, without waiting for its dependencies to be committed too. If applied straightforwardly, this would violate linearizability, so the protocol introduces additional mitigations that complicate it further. As observed by follow-up work on EPaxos [31], commands without any return values at all are rare in real systems. Hence, we do not consider this optimization in our protocol.

3.2 Commit Protocol

In EPaxos* a client submits its command to one of the processes, called the *initial coordinator*, which will drive the agreement on the command's dependencies. If the initial coordinator is suspected of failure, another process executes a *recovery* protocol to take over as a new coordinator. For each command, a process follows only one coordinator at a time. To ensure this, as in Paxos [12], the lifecycle of the command is divided into a series of *ballots*, each managed by a designated coordinator. A ballot is represented by a natural number, with ballot 0 reserved for the initial coordinator. Each process stores the current ballot number for a command *id* in bal[*id*]. We now describe the protocol for committing commands at ballot 0 (Figure 3). This protocol closely follows the original EPaxos: the main differences for EPaxos* are in recovery (§3.3).

Computing dependencies. When the initial coordinator receives a command c from a client (line 8), it first assigns to c a unique identifier *id*, from which the coordinator's identity can be extracted as `initCoord`(*id*). The coordinator then computes the *initial dependencies* of c as the set of all conflicting commands it is aware of, and broadcasts them together with c in a `PreAccept` message. Upon receiving a `PreAccept` (line 11), a process records the command payload in cmd[*id*] as well as initCmd[*id*], and the dependencies computed by the initial coordinator in initDep[*id*][1]. The process then computes its own proposal for the dependency set by completing the initial dependencies with the set of conflicting commands it is aware of (line 16). Finally, it advances *id* to a PREACCEPTED phase and replies to the coordinator with

[1] As we noted earlier, in exceptional cases cmd[*id*] may end up assigned to Nop. Keeping the originally submitted payload in initCmd[*id*] is needed for the recovery protocol (§3.3).

8 `submit`(c):
9   **let** $id = \texttt{new\_id}()$
10   **send** $\texttt{PreAccept}(id, c, \{id' \mid \mathsf{cmd}[id'] \bowtie c\})$ **to all**

11 **when received** $\texttt{PreAccept}(id, c, D)$ **from** q
12   **pre:** $\mathsf{bal}[id] = 0 \wedge \mathsf{phase}[id] = \textsc{initial}$
13   $\mathsf{cmd}[id] \leftarrow c$
14   $\mathsf{initCmd}[id] \leftarrow c$
15   $\mathsf{initDep}[id] \leftarrow D$
16   $\mathsf{dep}[id] \leftarrow D \cup \{id' \mid \mathsf{cmd}[id'] \bowtie \mathsf{cmd}[id]\}$
17   $\mathsf{phase}[id] \leftarrow \textsc{preaccepted}$
18   **send** $\texttt{PreAcceptOK}(id, \mathsf{dep}[id])$ **to** q

19 **when received** $\texttt{PreAcceptOK}(id, D_q)$ **from all** $q \in Q$
20   **pre:** $\mathsf{bal}[id] = 0 \wedge \mathsf{phase}[id] = \textsc{preaccepted} \wedge |Q| \geq n - f$
21   **let** $D = \bigcup_{q \in Q} D_q$
22   **if** $|Q| \geq n - e \wedge \forall q \in Q.\, D_q = \mathsf{initDep}[id]$ **then**
23   **send** $\texttt{Commit}(0, id, \mathsf{cmd}[id], D)$ **to all**
24   **else**
25   **send** $\texttt{Accept}(0, id, \mathsf{cmd}[id], D)$ **to all**

26 **when received** $\texttt{Accept}(b, id, c, D)$ **from** q
27   **pre:** $\mathsf{bal}[id] \leq b \wedge (\mathsf{bal}[id] = b \Longrightarrow \mathsf{phase}[id] \neq \textsc{committed})$
28   $\mathsf{bal}[id] \leftarrow b$
29   $\mathsf{abal}[id] \leftarrow b$
30   $\mathsf{cmd}[id] \leftarrow c$
31   $\mathsf{dep}[id] \leftarrow D$
32   $\mathsf{phase}[id] \leftarrow \textsc{accepted}$
33   **send** $\texttt{AcceptOK}(b, id)$ **to** q

34 **when received** $\texttt{AcceptOK}(b, id)$ **from** Q
35   **pre:** $\mathsf{bal}[id] = b \wedge \mathsf{phase}[id] = \textsc{accepted} \wedge |Q| \geq n - f$
36   **send** $\texttt{Commit}(b, id, \mathsf{cmd}[id], \mathsf{dep}[id])$ **to all**

37 **when received** $\texttt{Commit}(b, id, c, D)$ **from** q
38   **pre:** $\mathsf{bal}[id] = b$
39   $\mathsf{abal}[id] \leftarrow b$
40   $\mathsf{cmd}[id] \leftarrow c$
41   $\mathsf{dep}[id] \leftarrow D$
42   $\mathsf{phase}[id] \leftarrow \textsc{committed}$

Figure 3 EPaxos*: commit protocol. Self-addressed messages are delivered immediately. Initially, for all command identifiers id: $\mathsf{phase}[id] = \textsc{initial}$, $\mathsf{bal}[id] = \mathsf{abal}[id] = 0$, $\mathsf{initCmd}[id] = \mathsf{cmd}[id] = \bot$, $\mathsf{initDep}[id] = \mathsf{dep}[id] = \varnothing$. The value $\bot$ does not conflict with any command.

a `PreAcceptOK` message, carrying its proposal. After the coordinator receives `PreAcceptOK` from a *quorum* Q of at least $n - f$ processes (line 19), it decides whether to take the *fast path* or the *slow path*. The former commits the command immediately, while the latter requires an additional round of communication.

Fast path. The coordinator takes the fast path (line 22) if: *(i)* quorum Q is *fast*, i.e., contains at least $n - e$ processes (recall that $e \leq f$); and *(ii)* all the dependencies received from the processes in Q are identical to the initial ones. In this case the coordinator broadcasts a `Commit` message, which informs all the processes about the agreed dependencies. Upon receiving this message (line 37), a process advances the command to the COMMITTED phase, making it eligible for execution (§3.1).

For example, consider Figure 4, where $n = 5$, $f = 2$ and $e = 1$. Process p_1 coordinates a command c with identifier id, and sends a `PreAccept` message with $\varnothing$ as its initial dependencies. Since this command is the first one to be received at p_2, p_3 and p_4, they reply with $\varnothing$. Together with p_1, these processes form a fast quorum of size $n - e = 4$. Thus, p_1 immediately sends a `Commit` message for id (depicted by dashed lines).

The fast path mechanism ensures that the protocol is e-fast, as defined in §2. To see why, consider some E-faulty synchronous run with $|E| \leq e$. Assume that a conflict-free command c is submitted by a process p at time t, and let D_0 be the initial dependencies that p sends in a `PreAccept` message for c. Since c is conflict-free, each conflicting command c' either belongs to D_0, or cannot be submitted until c is executed at some process. Then every `PreAcceptOK` message for c sent back to p will carry D_0. Since at most e processes fail, there will be $n - e$ such messages. Then, since the run is synchronous, p will execute c by $t + 2\Delta$.

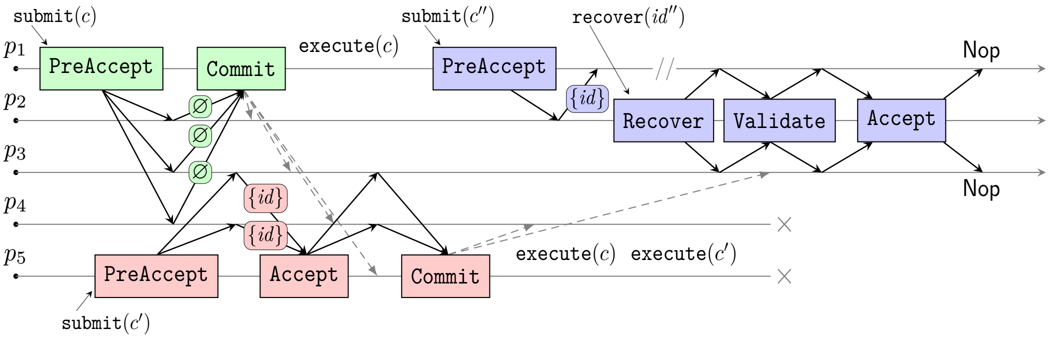

Figure 4 Example of an EPaxos* run with pairwise conflicting commands c, c', and c'', which have identifiers id, id' and id'', respectively. We assume $n = 5$, $f = 2$ and $e = 1$. During the run, p_1 is briefly disconnected (//), while p_4 and p_5 fail ($\times$).

Slow path. If the coordinator does not manage to assemble a fast quorum agreeing on the dependencies, it takes a slow path (line 24), which requires an additional round of communication. The coordinator computes the final dependencies of the command as the union of the proposals it received in `PreAcceptOK`s and broadcasts them together with the initial dependencies in an `Accept` message, analogous to the `2A` message of Paxos. A recipient records the dependencies, advances the command to an ACCEPTED phase, and replies to the coordinator with `AcceptOK`, analogous to the `2B` message of Paxos (line 26). Once the coordinator receives a quorum of `AcceptOK` messages (line 34), it broadcasts a `Commit` message, informing the processes that the dependencies have been agreed. The slow path code is reused at ballots > 0, and thus, the above messages carry the ballot of the coordinator. A process records the last ballot where it accepted a slow path proposal for each command in an abal array (line 29). This corresponds to the second ballot variable of Paxos that was missing from the original EPaxos protocol (§1).

For example, in Figure 4 process p_5 coordinates a command c' with identifier id', and sends a `PreAccept` message with $\varnothing$ as its initial dependencies. Since processes p_3 and p_4 receive this message after pre-accepting a conflicting command c with identifier id, they reply with $\{id\}$. Due to this disagreement, p_5 takes the slow path, and proposes $\{id\}$ as the dependency set for id'. This value is committed after another round of communication. In the end, c' is executed after c (§3.1).

Ordering guarantee. A key property of the above protocol is that it preserves Invariant 2 for pairs of conflicting commands committed at ballot 0: at least one of the two commands is in the dependency set of the other.

Proof of Invariant 2 for ballot 0. Take two distinct commands id and id', committed at ballot 0 with payloads c and c' and dependency sets D and D', respectively. Assume that $c \bowtie c'$. According to line 21, there is a quorum Q of $\texttt{PreAcceptOK}(id, D_q)$ messages such that $|Q| \geq n - f$ and $D = \bigcup_{q \in Q} D_q$. Similarly, there is a quorum Q' of $\texttt{PreAcceptOK}(id', D'_q)$ messages such that $|Q'| \geq n - f$ and $D' = \bigcup_{q \in Q'} D'_q$. By line 13, before sending `PreAcceptOK`, processes in Q (respectively, Q') store c (respectively, c') in cmd. Since $n \geq 2f + 1$, we have $|Q \cap Q'| \geq n - 2f \geq 1$, and thus $Q \cap Q' \neq \varnothing$. Take any process $q \in Q \cap Q'$ and assume without loss of generality that q sends $\texttt{PreAcceptOK}(id, D_q)$ before $\texttt{PreAcceptOK}(id', D'_q)$. Then the computation at line 16 ensures that $id \in D'_q$, and thus $id \in D'$. ◀

3.3 Recovery Protocol

If a coordinator of a command is suspected of failure, another process is nominated to take over; this nomination is done using standard techniques based on failure detectors [5], and we defer the details to [27, §C]. The new coordinator executes a recovery protocol for a command with identifier *id* by calling `recover`(id) in Figure 5 (for now, we ignore the highlighted parts of the code, which are explained in §4). The coordinator picks a ballot it owns higher than any it has previously participated in and asks processes to join it with a `Recover` message, analogous to the `1A` message of Paxos. Processes join the ballot if they do not already participate in a higher one. In this case they reply with a `RecoverOK` message, which is analogous to the `1B` message of Paxos and carries all the local information about the command being recovered (line 46). The new coordinator waits until it receives `RecoverOK` messages from a quorum Q of at least $n - f$ processes (line 50), after which it computes its proposal for the dependency set. To maintain consensus on dependencies (Invariant 1), the new coordinator must propose the same dependencies that were decided in earlier ballots, if any. The latter could be done either on the slow or the fast path, and the protocol addresses the two cases separately.

Recovering slow path decisions. As in Paxos, the new coordinator first considers the subset $U \subseteq Q$ of processes with the highest reported abal (line 53, assigned at line 29): the states of these processes supersede those from lower ballots. If a process in U has already committed the command *id* (line 54), then the coordinator broadcasts a `Commit` message with the committed dependency set. Similarly, if a process in U has accepted the command *id* (line 55), then the coordinator resumes the commit protocol interrupted by its predecessor by broadcasting an `Accept` message with the accepted dependency set and the new ballot. These rules are enough to recover dependencies committed via the slow path: in this case there must exist a quorum of $\geq n-f$ processes each having $\mathsf{phase}[id] \in \{\text{ACCEPTED}, \text{COMMITTED}\}$; since $n \geq 2f+1$, this quorum intersects with the recovery quorum Q, so one of the conditions at lines 54 or 55 must hold.

Recovering fast path decisions. Even if none of the processes in U has committed or accepted *id*, it is still possible that the initial coordinator committed the command via the fast path. The recovery of such decisions is much more subtle. A naive approach would be to do this similarly to Fast Paxos [14]. If *id* took the fast path then, since the size of the fast quorum is $n - e$ (line 22), at least $|Q| - e$ processes in the recovery quorum Q must have pre-accepted *id*'s initial dependencies. Thus, if the latter condition is satisfied (line 57), the new coordinator could recover the command *id* with its initial dependencies by broadcasting them in an `Accept` message in the new ballot (line 64). Otherwise, the coordinator would *abort* the recovery of *id* by proposing Nop as its payload in the new ballot (line 80). This approach preserves Invariant 1, but unfortunately, it breaks Invariant 2, as we now illustrate.

▶ **Example 6.** Consider again Figure 4. After commands id and id' are committed with payloads c and c', process p_1 submits another command id'', with payload c'' that conflicts with c and c'. The initial dependency set of id'' is $\{id\}$. Since p_2 did not participate in committing id', it pre-accepts id'' with its initial dependencies. At this point, suppose the network connection at p_1 is briefly disrupted. Process p_2 suspects that p_1 failed and starts recovering id''. If it gathers a quorum $Q_{id''} = \{p_1, p_2, p_3\}$, it will observe itself and p_1 voting for the same dependency set $\{id\}$. Together with p_4 and p_5 these processes *could* have formed a fast quorum $\{p_1, p_2, p_4, p_5\}$ to commit id'' with $\{id\}$. Thus, p_2 may think that it must recover id'' with the dependency set $\{id\}$ to preserve Invariant 1. But then the pair of commands id' and id'' breaks Invariant 2, and thus, Consistency (§2).

43 $\texttt{recover}(id)$:
44 let $b =$ (a ballot owned by p such that $b > \mathsf{bal}[id]$)
45 **send** $\texttt{Recover}(b, id)$ **to all**

46 **when received** $\texttt{Recover}(b, id)$ **from** q
47 **pre:** $\mathsf{bal}[id] < b$
48 $\mathsf{bal}[id] \leftarrow b$
49 **send** $\texttt{RecoverOK}(b, id, \mathsf{abal}[id], \mathsf{cmd}[id], \mathsf{dep}[id], \mathsf{initDep}[id], \mathsf{phase}[id])$ **to** q

50 **when received** $\texttt{RecoverOK}(b, id, abal_q, c_q, dep_q, initDep_q, phase_q)$ **from all** $q \in Q$
51 **pre:** $\mathsf{bal}[id] = b \wedge |Q| \geq n - f$
52 **let** $b_{\max} = \max\{abal_q \mid q \in Q\}$
53 **let** $U = \{q \in Q \mid abal_q = b_{\max}\}$
54 **if** $\exists q \in U.\, phase_q = \text{COMMITTED}$ **then send** $\texttt{Commit}(b, id, c_q, dep_q)$ **to all**
55 **else if** $\exists q \in U.\, phase_q = \text{ACCEPTED}$ **then send** $\texttt{Accept}(b, id, c_q, dep_q)$ **to all**
56 **else if** $\texttt{initCoord}(id) \in Q$ **then send** $\texttt{Accept}(b, id, \mathsf{Nop}, \varnothing)$ **to all**
57 **else if** $\exists R \subseteq Q.\, |R| \geq |Q| - e \wedge \forall q \in R.\, (phase_q = \text{PREACCEPTED} \wedge dep_q = initDep_q)$ **then**
58 **let** $R_{\max}$ be the largest set R that satisfies the condition at line 57
59 **let** $(c, D) = (c_q, dep_q)$ for any $q \in R$
60 **send** $\texttt{Validate}(b, id, c, D)$ **to all processes in** Q
61 **wait until received** $\texttt{ValidateOK}(b, id, I_q)$ **from all** $q \in Q$
62 **let** $I = \bigcup_{q \in Q} I_q$
63 **if** $I = \varnothing$ **then**
64 **send** $\texttt{Accept}(b, id, c, D)$ **to all**
65 **else if** $(\exists(id', \text{COMMITTED}) \in I) \vee (|R_{\max}| = |Q| - e \wedge \exists(id', \_) \in I.\, \texttt{initCoord}(id') \notin Q)$ **then**
66 **send** $\texttt{Accept}(b, id, \mathsf{Nop}, \varnothing)$ **to all**
67 **else**
68 **send** $\texttt{Waiting}(id, |R_{\max}|)$ **to all**
69 **wait until**
70 **case** $\exists(id', \_) \in I.\, \mathsf{phase}[id'] = \text{COMMITTED} \wedge (\mathsf{cmd}[id'] \neq \mathsf{Nop} \wedge id \notin \mathsf{dep}[id'])$ **do**
71 **send** $\texttt{Accept}(b, id, \mathsf{Nop}, \varnothing)$ **to all**
72 **case** $\forall(id', \_) \in I.\, \mathsf{phase}[id'] = \text{COMMITTED} \wedge (\mathsf{cmd}[id'] = \mathsf{Nop} \vee id \in \mathsf{dep}[id'])$ **do**
73 **send** $\texttt{Accept}(b, id, c, D)$ **to all**
74 **case** $\exists(id', \_) \in I.\, (p$ **received** $\texttt{Waiting}(id', k')) \wedge k' > n - f - e$ **do**
75 **send** $\texttt{Accept}(b, id, \mathsf{Nop}, \varnothing)$ **to all**
76 **case** p **received** $\texttt{RecoverOK}(b, id, \_, cmd, dep, \_, phase)$ **from** $q \notin Q$ **with** $phase = \text{COMMITTED} \vee phase = \text{ACCEPTED} \vee q = \texttt{initCoord}(id)$ **do**
77 **if** $phase = \text{COMMITTED}$ **then send** $\texttt{Commit}(b, id, cmd, dep)$ **to all**
78 **else if** $phase = \text{ACCEPTED}$ **then send** $\texttt{Accept}(b, id, cmd, dep)$ **to all**
79 **else send** $\texttt{Accept}(b, id, \mathsf{Nop}, \varnothing)$ **to all**
80 **else send** $\texttt{Accept}(b, id, \mathsf{Nop}, \varnothing)$ **to all**

81 **when received** $\texttt{Validate}(b, id, c, D)$ **from** q
82 **pre:** $\mathsf{bal}[id] = b$
83 $\mathsf{cmd}[id] \leftarrow c$
84 $\mathsf{initCmd}[id] \leftarrow c$
85 $\mathsf{initDep}[id] \leftarrow D$
86 **let** $I = \{(id', \mathsf{phase}[id']) \mid id' \neq id \wedge id' \notin D \wedge$
$(\mathsf{phase}[id'] = \text{COMMITTED} \implies \mathsf{cmd}[id'] \neq \mathsf{Nop} \wedge \mathsf{cmd}[id'] \bowtie c \wedge id \notin \mathsf{dep}[id']) \wedge$
$(\mathsf{phase}[id'] \neq \text{COMMITTED} \implies \mathsf{initCmd}[id'] \neq \bot \wedge \mathsf{initCmd}[id'] \bowtie c \wedge id \notin \mathsf{initDep}[id'])\}$
87 **send** $\texttt{ValidateOK}(b, id, I)$ **to all**

Figure 5 EPaxos*: recovery protocol at a process p. Self-addressed messages are delivered immediately. The highlighted parts of the code enable the optimization of §4.

Validation phase overview. The above pitfall arises because the new coordinator can only *suspect* that a command took the fast path, since it observes only a part of the fast quorum: $|Q| - e$ votes out of $n - e$. The coordinator can then resurrect commands (such as id'' in Example 6) that did not actually take the fast path and are missing from the dependencies of conflicting commands (such as id'). To avoid this, EPaxos* augments the above naive recovery with a special *validation phase*, which performs a more precise analysis of whether the command being recovered could have taken the fast path. This validation phase is the key novel contribution of our protocol: as we explain in §6, it is simpler than the original EPaxos recovery, and has been rigorously proved correct [27, §D].

During the validation phase, the new coordinator sends to its recovery quorum Q a `Validate` message with the payload c and the dependencies D that it would like to propose in the new ballot for the command being recovered (line 60). The recipients reply with `ValidateOK` messages carrying additional information (line 87). To see what kind of information is helpful to the new coordinator, consider Example 6 again. Process $p_3 \in Q_{id''}$ committed id' with dependencies that do not contain id'', and thus it knows that id'' could not take the fast path with a dependency set that does not contain id': this would contradict Invariant 2 for commands committed at ballot 0, which always holds, as we proved in §3.2. In this case, we say that command id' *invalidates* the recovery of id''. The `ValidateOK` messages carry a set I including the identifiers of such commands (the first implication at line 86).

▶ **Definition 7.** *A command id' **invalidates** the recovery of a command id with a payload c and a dependency set D, if a process has committed id' with a payload $c' \neq \mathsf{Nop}$ and a dependency set D' such that $c' \bowtie c$, $id' \notin D$ and $id \notin D'$.*

As illustrated by the above example, recovering id with c and D in the presence of an invalidating command would violate Invariant 2. Hence, in this case the coordinator aborts the recovery of id by replacing its payload with a Nop command. This does not violate Invariant 1 because in this case we can prove that it could not take the fast path. For instance, if the invalidating command is committed at ballot 0, then this follows from the proof of Invariant 2 for ballot 0 given in §3.2. The set carried by `ValidateOK` messages also includes the identifiers of *potentially invalidating* commands – those that are not yet committed but may become invalidating once they are (the second implication at line 86).

▶ **Definition 8.** *A command id' **potentially invalidates** the recovery of a command id with a payload c and a dependency set D, if a process has not committed id', but stores it with an initial payload c' and an initial dependency set D'_0 such that $c' \bowtie c$, $id' \notin D$ and $id \notin D'_0$.*

We can prove that any uncommitted command id' that may eventually invalidate id must satisfy this definition. In particular, note that the initial dependencies D'_0 of id' are always included into its committed dependencies (line 16). Hence, if $id \in D'_0$ (contradicting Definition 8), then id' cannot be committed with dependencies D' such that $id \notin D'$, and thus id' cannot invalidate id. If the coordinator finds potentially invalidating commands, it cannot yet determine whether it is safe to recover id. Thus, it waits for those commands to be committed before making a decision.

Note that the second disjunct of line 86 and Definition 8 consider a command id' to be potentially invalidating for id even if a process has $\mathsf{cmd}[id'] = \mathsf{Nop}$, but $\mathsf{initCmd}[id'] \neq \bot$ and $\mathsf{initCmd}[id'] \bowtie c$. This is because we cannot yet be sure that id' will be committed with a Nop: if the coordinator trying to do this fails, another coordinator may query a different recovery quorum and decide to propose the original payload $\mathsf{initCmd}[id']$.

Validation details. In more detail, upon receiving `ValidateOK` messages from the recovery quorum Q (line 61), the new coordinator acts as follows:

1. If no invalidating commands are detected at the quorum, the coordinator recovers id with c and D (line 63). In this case, no invalidating command will appear in the future: since the payload c is now stored at the quorum (line 83), any future conflicting command will depend on id.
2. If some invalidating command is detected, then the coordinator aborts the recovery of id by replacing its payload with a Nop command (line 65).
3. If none of the above holds, the coordinator waits until the potentially invalidating commands are committed. If some of these commands end up being invalidating, just like in item 2, the coordinator aborts the recovery of id (line 70). Otherwise, the coordinator recovers id with c and D (line 72).
4. Item 3 may require the coordinator of id to wait for a potentially invalidating command id' that is itself undergoing recovery. The recovery of id' may then also need to wait for another command. One may get worried that this will lead to cycles of commands waiting for each other, making the protocol stuck. Fortunately, if the coordinator of id finds out that id' is itself blocked, the coordinator can stop waiting for id'. To enable this, before any process starts the wait at line 69, it broadcasts a `Waiting` message, carrying the identifier of the command it is recovering (line 68). While the coordinator of id is waiting, it listens for `Waiting` messages from other coordinators. Upon receiving `Waiting`(id'), the coordinator aborts the recovery of id (line 74). The following proposition implies that this is safe because, in this case, id could not take the fast path.

▶ **Lemma 9.** *Consider commands id and id' with conflicting initial payloads c and c', and assume that neither of the commands is included in the initial dependencies of the other. If a message* `Waiting`(id') *has been sent, then id did not take the fast path.*

Proof. By contradiction, assume that id takes the fast path with a fast quorum Q_f, i.e., $|Q_f| \geq n - e$ and all processes in Q_f pre-accepted id with its initial dependencies D (line 22). Since `Waiting`(id') has been sent, there exists a recovery quorum Q' and a set $R' \subseteq Q'$ such that $|R'| \geq |Q'| - e \geq n - f - e$ and all processes in R' pre-accepted id' with its initial dependencies D' (line 57). Since $n \geq 2e + f + 1$, the intersection of Q_f and R' has a cardinality

$$\geq (n - e) + (n - f - e) - n = n - 2e - f \geq 1. \tag{1}$$

Thus, some process p pre-accepted id with D and id' with D'. Assume without loss of generality that the former happens before the latter. When p pre-accepts id, it assigns $\mathsf{cmd}[id]$ to c (line 13). It is easy to see that, at any time after this, p has $\mathsf{cmd}[id] = c$ or $\mathsf{cmd}[id] = \mathsf{Nop}$. In particular, this holds when p pre-accepts id'. Then, since $c \bowtie c'$ and $\mathsf{Nop} \bowtie c'$, line 16 implies $id \in D'$, which contradicts our assumption. ◀

Using this result, in [27, §C] we prove the correctness of our protocol.

▶ **Theorem 10.** *The above-presented* EPaxos* *protocol implements an f-resilient e-fast SMR protocol, provided $n \geq \max\{2e + f + 1, 2f + 1\}$.*

4 EPaxos*: Optimized Protocol for $n \geq \max\{2e+f-1, 2f+1\}$

We now explain how to optimize the protocol presented so far to require $n \geq \max\{2e+f-1, 2f+1\}$, which matches the lower bound in Theorem 4. For $n = 2f+1$ this allows setting $e = \lceil \frac{f+1}{2} \rceil$ – the threshold the original EPaxos aimed to achieve [19]. In particular, for the values $n = 3$ and $n = 5$, frequent in practice, e can be a minority of processes. In these cases, the fast path contacts quorums of the same size as in the usual, slower Paxos [12].

The optimization is enabled by adding the highlighted code to the recovery protocol in Figure 5. First, before attempting to recover a fast path decision for a command *id*, the new coordinator checks whether the initial coordinator of *id* belongs to the recovery quorum (line 56) – if so, it aborts the recovery, proposing a Nop. This is safe because in this case *id* cannot take the fast path. Indeed, the initial coordinator of *id* could not take the fast path before sending its `RecoverOK` message: otherwise, it would report the command as committed, and the new coordinator would take the branch at line 54. The initial coordinator will also not take the fast path in the future: when sending `RecoverOK`, it switches to a ballot > 0, which disables the fast path (line 20).

We also make a change to the case when the new coordinator waits for potentially invalidating commands to commit (line 69). If during this wait the new coordinator receives an additional `RecoverOK` message for *id* from a process that has committed or accepted *id*, or from the initial coordinator of *id* (line 76), it handles the message just like a previously received one, completing the recovery as in lines 54–56.

The next change is that the new coordinator of *id* aborts the recovery if exactly $|Q| - e$ processes in the recovery quorum Q satisfy the condition at line 57, and the coordinator of a potentially invalidating non-Nop command is not part of Q (line 65). The following proposition shows that this is safe.

▶ **Lemma 11.** *If the first disjunct at line 65 does not hold, but the second one does, then id did not take the fast path.*

Proof. By contradiction, assume that *id* takes the fast path with a fast quorum Q_f, i.e., $|Q_f| \geq n - e$ and all processes in Q_f pre-accepted *id* with its initial dependencies D (line 22). By our assumption, $|R_{\max}| = |Q| - e$ and for some id' and $phase'$ we have $(id', phase') \in I$, $\mathtt{initCoord}(id') \notin Q$, and $phase' \neq$ COMMITTED. By line 86, we also have $id' \notin D$. Let D' be the initial dependency set of id'. Notice that by lines 15 and 85, whenever a process has $\mathsf{initCmd}[id'] \neq \bot$, it also has $\mathsf{initDep}[id'] = D'$. Then by line 86, the process whose `ValidateOK` message resulted in $(id', phase')$ being included into I at line 62 must have had $id \notin \mathsf{initDep}[id'] = D'$. Moreover, by the same line, *id* and id' have conflicting initial payloads. Since $|R_{\max}| = |Q| - e$, we know that $|Q| - e$ processes in Q pre-accepted *id* with D, and the remaining e did not. Since $|Q_f| \geq n - e$ processes pre-accepted *id* with D, all processes that are not in Q must belong to Q_f. Then since $\mathtt{initCoord}(id') \notin Q$, we must have $\mathtt{initCoord}(id') \in Q_f$, so $\mathtt{initCoord}(id')$ pre-accepted *id* with D and id' with D'. Given that the initial payloads of *id* and id' conflict, we obtain a contradiction in the same way as in the proof of Lemma 9. ◀

Finally, in the baseline protocol the new coordinator of *id* could abort the recovery if it received a $\mathtt{Waiting}(id')$ message for a potentially invalidating command id' (line 74). This cannot generally be done in the optimized protocol, because Lemma 9 no longer holds: its proof relies on the fact that any two sets of $n - e$ and $n - f - e$ processes intersect (Eq. 1), which is not ensured with $n \geq \max\{2e+f-1, 2f+1\}$. Hence, the optimized protocol imposes an additional constraint on when the new coordinator can abort the recovery. Each `Waiting` message sent at line 68 includes the size of the largest set R satisfying the condition at line 57. Then if the new coordinator of id receives `Waiting`(id', k'), it can only abort the recovery if $k' > n - f - e$ (note that we always have $k' \geq n - f - e$). The following weaker version of Lemma 9 shows that this is safe.

▶ **Lemma 12.** *Consider commands id and id' with conflicting initial payloads c and c', and assume that neither of the commands is included in the initial dependencies of the other. If a message* `Waiting`(id', k') *has been sent with $k' > n - f - e$, then id did not take the fast path.*

Proof. By contradiction, assume that id takes the fast path with a fast quorum Q_f, i.e., $|Q_f| \geq n - e$ and all processes in Q_f pre-accepted id with its initial dependencies D (line 22). Since `Waiting`(id', k') has been sent, there exists a recovery quorum Q' and a set $R' \subseteq Q'$ such that $|R'| = k' > n - f - e$ and all processes in R' pre-accepted id' with its initial dependencies D' (line 57). Moreover, `initCoord`(id') also pre-accepted id' with D', and line 56 ensures that `initCoord`$(id') \notin R'$. Since $n \geq 2e + f - 1$, the intersection of Q_f and $R' \cup \{$`initCoord`$(id')\}$ has a cardinality

$$\geq (n - e) + (n - f - e + 2) - n = n - 2e - f + 2 \geq 1.$$

Thus, some process pre-accepted id with D and id' with D'. From this we obtain a contradiction in the same way as in the proof of Lemma 9. ◀

We next prove that, even though the optimized protocol restricts when the recovery can be aborted because of `Waiting` messages, the recovery eventually terminates.

▶ **Lemma 13.** *Every command id is eventually committed at all correct processes.*

Proof sketch. Assume by contradiction that some process never commits a command id. Our coordinator nomination mechanism (§3.3 and [27, §C]) ensures that a command keeps getting recovered until all correct processes commit it, and that eventually after GST and after failures have stopped, each command can only be recovered by a single process. Let p be this process for the command id, and let D be id's initial dependency set. We can show that every recovery attempt by p must get stuck at line 69, meaning id always encounters a potentially invalidating command at line 86 that is itself blocked.

As we consider runs with only finitely many commands, there is a command id' that blocks id infinitely many times, i.e., process p computes a set I at line 62 such that $id' \in I$. Eventually, after GST, id' is also recovered only by a single process. Let p' be this process, and D' be the initial dependencies of id'. Then, since id repeatedly blocks on id' (and in particular never satisfies the first disjunct at line 65), line 86 implies: *(i)* $id' \notin D$; *(ii)* $id \notin D'$; and *(iii)* the initial payloads of id and id' conflict. By line 84, eventually, the initial payload of id is stored at a recovery quorum of id. Recovery quorums of id and id' intersect, so eventually on every recovery of id', process p' will receive a `ValidateOK` message from some correct process q' storing id's initial payload. Notice that q' never commits id: otherwise id's recovery would eventually complete at line 54 or 77. Then, due to *(i)–(iii)*, process q' will add id to the invalidating set at line 86, so eventually, every recovery of id' by p' will compute a set I' at line 62 such that $id \in I'$.

Due to line 68, process p' will eventually receive `Waiting`(id, k) from p. We must have $k \leq n - f - e$: otherwise, p' would satisfy the condition at line 74 and finish the recovery, contradicting our assumption. Hence, when p recovers id, it has $|R_{\max}| \leq n - f - e$. On the other hand, by lines 50 and 58, $|R_{\max}| \geq |Q| - e \geq n - f - e \geq |R_{\max}|$, so that $|R_{\max}| = |Q| - e$. Since the conditions at lines 56 and 76 never hold for id', `initCoord`(id') must be faulty. But then eventually, p will initiate a recovery of id for which the condition at line 65 will hold, letting p finish the recovery. This contradicts our assumption. ◀

The full proof of correctness for EPaxos* is given in [27, §C] and uses the invariants and lemmas we have presented so far.

▶ **Theorem 14.** *The optimized* EPaxos* *protocol implements an f-resilient e-fast SMR protocol, provided* $n \geq \max\{2e+f-1, 2f+1\}$.

5 Thrifty Protocol Version

The EPaxos* protocol can also be changed to incorporate the *thrifty* optimization of the original EPaxos. This allows the coordinator to take the fast path even when the rest of the fast quorum processes disagree with it on dependencies, provided they agree among themselves. In exchange, the coordinator has to select the fast quorum it will use a priori, and communicate only with this quorum. Hence, the coordinator will not be able to take the fast path even with a single failure, if this failure happens to be within the selected fast quorum. Thus, the resulting protocol is no longer e-fast for any $e > 0$ (§2). We defer the description of this protocol to [27, §B] and its proof of correctness to [27, §D].

6 Comparison with the Original Egalitarian Paxos

The main subtlety of the original EPaxos protocol for $e = \lceil \frac{f+1}{2} \rceil$ is the failure-recovery procedure. When there is a suspicion that a command *id* took the fast path with payload c and dependency set D (akin to line 57 in Figure 5), the new coordinator in EPaxos performs a *Tentative Pre-accept* phase: it tries to convince at least $f+1$ processes to pre-accept the values c and D for *id* (as in line 11). Unlike our Validation phase, the Tentative Pre-accept phase has a major side effect: processes participating in it that did not already know about the command *id* pre-accept it with the dependencies suggested by the new coordinator. As we now explain, this difference makes the protocol more complex and error-prone.

Safety. As mentioned in §1, Egalitarian Paxos has a bug in its use of Paxos-like ballot variables: the protocol uses a single ballot variable instead of two. Because recovery in this protocol is much more complex than in Paxos, fixing this is nontrivial. In particular, when a process tentatively pre-accepts a command, it is unclear if the second variable (corresponding to our abal) should be updated or not. We avoid this choice altogether by not forcing a process to commit to a choice of dependencies during the validation phase.

As we also mentioned in §1, the available protocol descriptions [17,19] only detail recovery for the thrifty version of the protocol, where the coordinator fixes a fast quorum for each command (§5). Adjusting it to the non-thrifty version (corresponding to our protocol in §3-4) is nontrivial. The reason is that, in EPaxos, the new coordinator recovering a command may encounter processes in its recovery quorum that tentatively pre-accepted dependencies proposed by past coordinators. For each such process, the new coordinator needs to check whether an old coordinator is part of the initial fast path quorum [17, page 43, conditions 7.d and 7.e]. It is unclear how this check can be done in the non-thrifty version, where *any* process can potentially be part of the fast quorum that committed the command. Without this missing piece of information, the non-thrifty version of EPaxos is incomplete.

Liveness. In the thrifty version of EPaxos, two recovery attempts for the same command with different dependencies can block each other indefinitely. For example, some processes may tentatively pre-accept D or D', preventing either set from gathering the necessary $f+1$ pre-accepts and causing a recovery deadlock. We detail this bug in [27, §E].

Another liveness problem comes from the fact that a process does not tentatively pre-accept D and c if it has already recorded an *interfering* command id' – analogous to an invalidating command in EPaxos*. Any interfering command id' is reported back to the new coordinator. When the coordinator receives this information, it pauses the recovery of id and starts recovering id'. As a consequence, multiple processes may simultaneously attempt to coordinate the same command, even after `GST`, which may block progress. In EPaxos* we avoid this problem due to our use of `Waiting` messages (lines 69-79).

7 Related Work

State-machine replication. SMR is a classic approach to implementing fault-tolerant services [11, 28]. Schiper and Pedone [23], along with Lamport [13], observed that an SMR protocol can execute commutative commands in any order. To determine the predecessors of a command, these works use partially ordered consensus instances. Zielinski [33] further observed that the partial order can be replaced by a directed graph, which may have cycles. EPaxos [19] is the first practical SMR protocol to use this approach, which decentralizes the task of ordering state-machine commands.

Derivatives of EPaxos. The EPaxos protocol has inspired a number of follow-up works, surveyed in [16, 25, 32]. In particular, several proposals of protocols with similar architecture tried to achieve better performance in failure-free cases [3, 8, 9]. However, none of these protocols has optimal fault-tolerance: the values of e and f they admit are lower than what is required by our Theorem 4, and what EPaxos aimed to achieve. Janus [21] adapted EPaxos to implement transactions. Due to the complexity of the original protocol, it used its unoptimized version where fast quorums contain *all* processes. Our EPaxos* protocol can serve as a basis for a variant of Janus that reduces the fast quorum size without compromising correctness. Gryff [4] integrated EPaxos with ABD to improve the performance of writes in distributed key-value stores, while Tollman, Park and Ousterhout [31] proposed an enhancement to EPaxos that improves its performance. Since these works use EPaxos as a black box, they inherit its bugs. Our version would be a drop-in correct replacement.

Lower bounds. Pedone and Schiper [24] have previously demonstrated that an SMR protocol can tolerate up to $e = f \leq \frac{n}{3}$ failures while executing a command in two message delays. Lamport [15] generalized this result, showing that any protocol requires $n \geq \max\{2e + f + 1, 2f + 1\}$ processes (matched by the protocol in §3). These prior works measure the minimal time it takes to execute a command at *all* correct processes. In practice, a client typically only needs a fast response from the single replica to which it submitted the request. Our recent work [26] defined a notion of fast consensus using this more pragmatic approach. In §2, we generalized this notion to SMR and established the corresponding lower bound on the number of processes: $n \geq \max\{2e + f - 1, 2f + 1\}$. To the best of our knowledge, EPaxos* is the first provably correct protocol to match this bound.

References

1 `https://github.com/efficient/epaxos`.

2 Karolos Antoniadis, Julien Benhaim, Antoine Desjardins, Elias Poroma, Vincent Gramoli, Rachid Guerraoui, Gauthier Voron, and Igor Zablotchi. Leaderless consensus. *J. Parallel Distributed Comput.*, 176, 2023. `doi:10.1016/J.JPDC.2023.01.009`.

3 Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *Conference on Dependable Systems and Networks (DSN)*, 2017.

4 Matthew Burke, Audrey Cheng, and Wyatt Lloyd. Gryff: Unifying consensus and shared registers. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

5 Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2), 1996. doi:10.1145/226643.226647.

6 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

7 Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2), 1988. doi:10.1145/42282.42283.

8 Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability. In *European Conference on Computer Systems (EuroSys)*, 2021.

9 Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *European Conference on Computer Systems (EuroSys)*, 2020.

10 Maurice Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990. doi:10.1145/78969.78972.

11 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978. doi:10.1145/359545.359563.

12 Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998. doi:10.1145/279227.279229.

13 Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

14 Leslie Lamport. Fast Paxos. *Distributed Comput.*, 19(2), 2006. doi:10.1007/S00446-006-0005-X.

15 Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Comput.*, 19(2), 2006. doi:10.1007/S00446-006-0155-X.

16 Giuliano Losa, Sebastiano Peluso, and Binoy Ravindran. Brief announcement: A family of leaderless generalized-consensus algorithms. In *Symposium on Principles of Distributed Computing (PODC)*, 2016.

17 Iulian Moraru. Egalitarian distributed consensus. Technical report, CMU-CS-14-133, Carnegie Mellon University, PhD Thesis, 2014.

18 Iulian Moraru, David G. Andersen, and Michael Kaminsky. A proof of correctness for Egalitarian Paxos. Technical Report CMU-PDL-13-111, Carnegie Mellon University, 2013.

19 Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

20 Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting more concurrency from distributed transactions. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

21 Shuai Mu, Lamont Nelson, Wyatt Lloyd, and Jinyang Li. Consolidating concurrency control and consensus for commits under conflicts. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

22 Diego Ongaro and John K. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

23 Fernando Pedone and André Schiper. Generic broadcast. In *International Symposium on Distributed Computing (DISC)*, 1999.

24 Fernando Pedone and André Schiper. Brief announcement: On the inherent cost of generic broadcast. In *Symposium on Principles of Distributed Computing (PODC)*, 2004.

25 Tuanir França Rezende and Pierre Sutra. Leaderless state-machine replication: Specification, properties, limits. In *Symposium on Distributed Computing (DISC)*, 2020.

26 Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. Brief announcement: Revisiting lower bounds for two-step consensus. In *Symposium on Principles of Distributed Computing (PODC)*, 2025.

27 Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. Making democracy work: Fixing and simplifying Egalitarian Paxos (extended version). *arXiv*, 2511.02743, 2025. URL: `http://arxiv.org/abs/2511.02743`.

28 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22, 1990. `doi:10.1145/98163.98167`.

29 Benedict Elliott Smith, Tony Zhang, Blake Eggleston, and Scott Andreas. CEP-15: Fast general purpose transactions. https://cwiki.apache.org/confluence/display/CASSANDRA/CEP-15%3A+General+Purpose+Transactions?preview=/188744725/188744736/Accord.pdf, 2023.

30 Pierre Sutra. On the correctness of Egalitarian Paxos. *Inf. Process. Lett.*, 156, 2020. `doi:10.1016/J.IPL.2019.105901`.

31 Sarah Tollman, Seo Jin Park, and John K. Ousterhout. EPaxos revisited. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

32 Michael J. Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M. Hellerstein, and Ion Stoica. SoK: A generalized multi-leader state machine replication tutorial. *J. Syst. Res.*, 1(1), 2021. `doi:10.5070/SR31154817`.

33 Piotr Zieliński. Optimistic generic broadcast. In *Conference on Distributed Computing (DISC)*, 2005.