Making Democracy Work: Fixing and Simplifying Egalitarian Paxos (Extended Version)

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

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 Elliot 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, 27]. 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.

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 [30]. 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 [28]. 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 [29], 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 [29]. 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 §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 §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 [27]. 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 §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

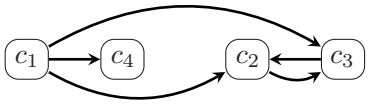

Figure 2 Example of a dependency graph.

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 §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 [30], 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 $\mathsf{bal}[id]$. 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 $\texttt{initCoord}(id)$. 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 $\mathsf{cmd}[id]$ as well as $\mathsf{initCmd}[id]$, and the dependencies computed by the initial coordinator in $\mathsf{initDep}[id]$[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 $\mathsf{cmd}[id]$ may end up assigned to Nop. Keeping the originally submitted payload in $\mathsf{initCmd}[id]$ is needed for the recovery protocol (§3.3).

```
 8 submit(c):
 9     let id = new_id()
10     send PreAccept(id, c, {id' | cmd[id'] ⋈ c})
         to all
11 when received PreAccept(id, c, D) from q
12     pre: bal[id] = 0 ∧ phase[id] = INITIAL
13     cmd[id] ← c
14     initCmd[id] ← c
15     initDep[id] ← D
16     dep[id] ← D ∪ {id' | cmd[id'] ⋈ cmd[id]}
17     phase[id] ← PREACCEPTED
18     send PreAcceptOK(id, dep[id]) to q
19 when received PreAcceptOK(id, D_q)
     from all q ∈ Q
20     pre: bal[id] = 0 ∧ phase[id] = PREACCEPTED ∧
            |Q| ≥ n − f
21     let D = ⋃_{q∈Q} D_q
22     if |Q| ≥ n − e ∧ ∀q ∈ Q. D_q = initDep[id] then
23         send Commit(0, id, cmd[id], D) to all
24     else
25         send Accept(0, id, cmd[id], D) to all
26 when received Accept(b, id, c, D) from q
27     pre: bal[id] ≤ b ∧
            (bal[id] = b ⟹ phase[id] ≠ COMMITTED)
28     bal[id] ← b
29     abal[id] ← b
30     cmd[id] ← c
31     dep[id] ← D
32     phase[id] ← ACCEPTED
33     send AcceptOK(b, id) to q
34 when received AcceptOK(b, id) from Q
35     pre: bal[id] = b ∧ phase[id] = ACCEPTED ∧
            |Q| ≥ n − f
36     send Commit(b, id, cmd[id], dep[id]) to all
37 when received Commit(b, id, c, D) from q
38     pre: bal[id] = b
39     abal[id] ← b
40     cmd[id] ← c
41     dep[id] ← D
42     phase[id] ← COMMITTED
```

Figure 3 EPaxos*: commit protocol. Self-addressed messages are delivered immediately. Initially, for all command identifiers id: $\mathsf{phase}[id] = \text{INITIAL}$, $\mathsf{bal}[id] = \mathsf{abal}[id] = 0$, $\mathsf{initCmd}[id] = \mathsf{cmd}[id] = \bot$, $\mathsf{initDep}[id] = \mathsf{dep}[id] = \varnothing$. 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 $t + 2\Delta$.

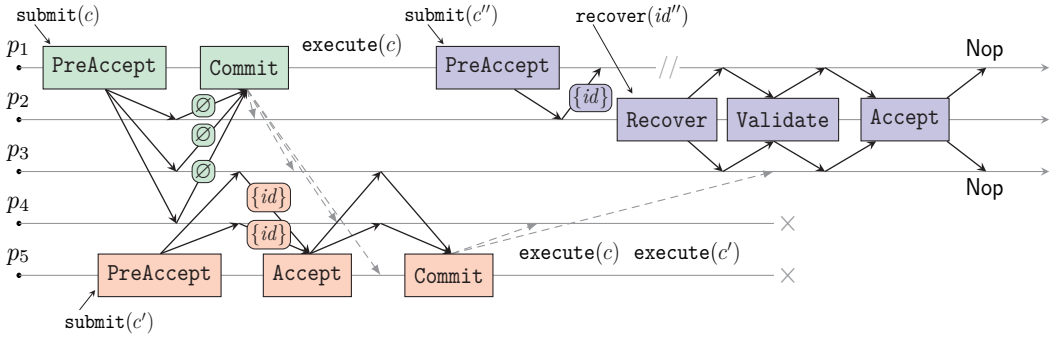

Figure 4 Example of an EPaxos* run with pairwise conflicting commands c, c', and c'', which have identifiers id, id' and id'', respectively. We assume $n = 5$, $f = 2$ and $e = 1$. During the run, p_1 is briefly disconnected (//), while p_4 and p_5 fail (×).

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 §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 56), 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 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 56 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 60), the new coordinator could recover the command id with its initial dependencies by broadcasting them in an `Accept` message in the new ballot (line 67). Otherwise, the coordinator would *abort* the recovery of id by proposing Nop as its payload in the new ballot (line 83). 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

```
43 recover(id):
44     let b = (a ballot owned by p such that b > bal[id])
45     send Recover(b, id) to all

46 when received Recover(b, id) from q
47     pre: bal[id] < b
48     bal[id] ← b
49     send RecoverOK(b, id, abal[id], cmd[id], dep[id], initDep[id], phase[id]) to q

50 when received RecoverOK(b, id, abal_q, c_q, dep_q, initDep_q, phase_q) from all q ∈ Q
51     pre: bal[id] = b ∧ |Q| ≥ n − f
52     let b_max = max{abal_q | q ∈ Q}
53     let U = {q ∈ Q | abal_q = b_max}
54     if ∃q ∈ U. phase_q = COMMITTED then
55         send Commit(b, id, c_q, dep_q) to all
56     else if ∃q ∈ U. phase_q = ACCEPTED then
57         send Accept(b, id, c_q, dep_q) to all
58     else if initCoord(id) ∈ Q then
59         send Accept(b, id, Nop, ∅) to all
60     else if ∃R ⊆ Q. |R| ≥ |Q| − e ∧ ∀q ∈ R. (phase_q = PREACCEPTED ∧ dep_q = initDep_q) then
61         let R_max be the largest set R that satisfies the condition at line 60
62         let (c, D) = (c_q, dep_q) for any q ∈ R
63         send Validate(b, id, c, D) to all processes in Q
64         wait until received ValidateOK(b, id, I_q) from all q ∈ Q
65         let I = ⋃_{q∈Q} I_q
66         if I = ∅ then
67             send Accept(b, id, c, D) to all
68         else if (∃(id', COMMITTED) ∈ I) ∨ (|R_max| = |Q| − e ∧ ∃(id', _) ∈ I. initCoord(id') ∉ Q) then
69             send Accept(b, id, Nop, ∅) to all
70         else
71             send Waiting(id, |R_max|) to all
72             wait until
73                 case ∃(id', _) ∈ I. phase[id'] = COMMITTED ∧ (cmd[id'] ≠ Nop ∧ id ∉ dep[id']) do
74                     send Accept(b, id, Nop, ∅) to all
75                 case ∀(id', _) ∈ I. phase[id'] = COMMITTED ∧ (cmd[id'] = Nop ∨ id ∈ dep[id']) do
76                     send Accept(b, id, c, D) to all
77                 case ∃(id', _) ∈ I. (p received Waiting(id', k')) ∧ k' > n − f − e do
78                     send Accept(b, id, Nop, ∅) to all
79                 case p received RecoverOK(b, id, _, cmd, dep, _, phase) from q ∉ Q with
                   phase = COMMITTED ∨ phase = ACCEPTED ∨ q = initCoord(id) do
80                     if phase = COMMITTED then send Commit(b, id, cmd, dep) to all
81                     else if phase = ACCEPTED then send Accept(b, id, cmd, dep) to all
82                     else send Accept(b, id, Nop, ∅) to all
83     else send Accept(b, id, Nop, ∅) to all

84 when received Validate(b, id, c, D) from q
85     pre: bal[id] = b
86     cmd[id] ← c
87     initCmd[id] ← c
88     initDep[id] ← D
89     let I = {(id', phase[id']) | id' ∉ D ∧
                   (phase[id'] = COMMITTED ⟹ cmd[id'] ≠ Nop ∧ cmd[id'] ⋈ c ∧ id ∉ dep[id']) ∧
                   (phase[id'] ≠ COMMITTED ⟹ initCmd[id'] ≠ ⊥ ∧ initCmd[id'] ⋈ c ∧ id ∉ initDep[id'])}
90     send 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.

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).

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 (§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 63). The recipients reply with `ValidateOK` messages carrying additional information (line 90). 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 89).

▶ **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 89).

▶ **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 89 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 64), 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 66). In this case, no invalidating command will appear in the future: since the payload c is now stored at the quorum (line 86), 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 68).
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 73). Otherwise, the coordinator recovers id with c and D (line 75).
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 72, it broadcasts a `Waiting` message, carrying the identifier of the command it is recovering (line 71). While the coordinator of id is waiting, it listens for `Waiting` messages from other coordinators. Upon receiving $\texttt{Waiting}(id')$, the coordinator aborts the recovery of id (line 77). 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* $\texttt{Waiting}(id')$ *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 $\texttt{Waiting}(id')$ 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 60). 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 §D 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 58) – 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 72). 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 79), it handles the message just like a previously received one, completing the recovery as in lines 54–59.

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 60, and the coordinator of a potentially invalidating non-Nop command is not part of Q (line 68). The following proposition shows that this is safe.

▶ **Lemma 11.** *If the first disjunct at line 68 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 $(id', phase') \in I$, $\texttt{initCoord}(id') \notin Q$, and $phase' \neq \text{COMMITTED}$. By line 89, we also have $id' \notin D$. Let D' be the initial dependency set of id'. Notice that by lines 15 and 88, whenever a process has $\mathsf{initCmd}[id'] \neq \bot$, it also has $\mathsf{initDep}[id'] = D'$. Then by line 89, the process whose `ValidateOK` message resulted in $(id', phase')$ being included into I at line 65 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 $\texttt{initCoord}(id') \notin Q$, we must have $\texttt{initCoord}(id') \in Q_f$, so $\texttt{initCoord}(id')$ 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 $\texttt{Waiting}(id')$ message for a potentially invalidating command id' (line 77). 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 71 includes the size of the largest set R satisfying the condition at line 60. Then if the new coordinator of id receives $\texttt{Waiting}(id', k')$, 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 $\texttt{Waiting}(id', k')$ 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 $\texttt{Waiting}(id', k')$ 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 60). Moreover, $\texttt{initCoord}(id')$ also pre-accepted id' with D', and line 58 ensures that $\texttt{initCoord}(id') \notin R'$. Since $n \geq 2e + f - 1$, the intersection of Q_f and $R' \cup \{\texttt{initCoord}(id')\}$ 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 §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 72, meaning id always encounters a potentially invalidating command at line 89 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 65 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 68), line 89 implies: *(i)* $id' \notin D$; *(ii)* $id \notin D'$; and *(iii)* the initial payloads of id and id' conflict. By line 87, 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 55 or 80. Then, due to *(i)*–*(iii)*, process q' will add id to the invalidating set at line 89, so eventually, every recovery of id' by p' will compute a set I' at line 65 such that $id \in I'$.

Due to line 71, process p' will eventually receive $\texttt{Waiting}(id, k)$ from p. We must have $k \leq n - f - e$: otherwise, p' would satisfy the condition at line 77 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 61, $|R_{\max}| \geq |Q| - e \geq n - f - e \geq |R_{\max}|$, so that $|R_{\max}| = |Q| - e$. Since the conditions at lines 58 and 79 never hold for id', $\texttt{initCoord}(id')$ must be faulty.

But then eventually, p will initiate a recovery of id for which the condition at line 68 will hold, letting p finish the recovery. This contradicts our assumption. ◀

The full proof of correctness for EPaxos* is given in §D and uses the invariants and propositions 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 §B and its proof of correctness to §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 60 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 §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 72-82).

7 Related Work

State-machine replication. SMR is a classic approach to implementing fault-tolerant services [11, 27]. 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 [32] 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, 31]. 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 [30] 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.
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.
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.
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.
11. Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7), 1978.
12. Leslie Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2), 1998.
13. Leslie Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.
14. Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2), 2006.
15. Leslie Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2), 2006.
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 Soma Chaudhuri and Shay Kutten, editors, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, PODC 2004, St. John's, Newfoundland, Canada, July 25-28, 2004*, page 401. ACM, 2004. doi:10.1145/1011767.1011868.

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 Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22, 1990.

28 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.

29 Pierre Sutra. On the correctness of Egalitarian Paxos. *Inf. Process. Lett.*, 156, 2020.

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

31 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.

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

A Proof of Theorem 4

To prove Theorem 4, we use the lower-bound result in [26] and the fact that (object) consensus reduces to SMR. A consensus object is an atomic object with a single operation, *propose*(v). When a process invokes *propose*(v), it proposes the value v; the operation returns the consensus decision. Consensus object ensures that *(i)* every decision is the proposal of some process; *(ii)* no two decisions are different; and that *(iii)* every correct process eventually decides.

We prove Theorem 4 by contradiction. Suppose the existence of an f-resilient e-fast SMR protocol $\mathcal{P}$ satisfying $n < \max\{2e + f - 1, 2f + 1\}$. To solve consensus with $\mathcal{P}$, we proceed as follows: Commands are pairs (v, p) where v is some consensus input value and p a process identifier. Commands (v, p) and (v', p') conflict when $v \neq v'$. To propose a value v, process p creates command (v, p) and submits it to $\mathcal{P}$. The first command executed locally contains the consensus decision.

It is straightforward to see that the above construction implements object consensus: No two processes execute conflicting commands in different orders. Hence, they must take the same decision. Moreover, by the Validity property of SMR, the decided value is proposed by some process.

Because protocol $\mathcal{P}$ tolerates up to f crashes, the construction is an f-resilient solution to consensus. We show that it is also e-two-step, as defined in [26]. Recall that this precisely means that for all $E \subseteq \Pi$ of size e:

1. For every value v and process $p \in \Pi \setminus E$, there exists an E-faulty synchronous run in which only p calls *propose*(), the proposed value is v, and p decides a value by time 2Δ.
2. For every value v and process $p \in \Pi \setminus E$, there exists an E-faulty synchronous run in which all processes in $\Pi \setminus E$ call *propose*(v) at the beginning of the first round, and p decides a value by time 2Δ.

Fix a subset $E \subseteq \Pi$ of size e. Assume some value v, a process $p \in \Pi \setminus E$, and a run of the above construction in which p calls *propose*(v) at the start of the first round and the other processes do not propose anything. Because $\mathcal{P}$ is e-fast, p executes the command (v, p), and thus decides, at time 2Δ. Similarly, consider a run in which all the processes in $\Pi \setminus E$ propose the same value v initially. In this run, all the submitted commands are conflict-free. Hence, each correct process decides by time 2Δ.

Theorem 2 in [26] states that an f-resilient e-two-step consensus object is implementable iff $n \geq \max\{2e + f - 1, 2f + 1\}$. This contradicts the existence of protocol $\mathcal{P}$.

B Thrifty Protocol Details

Figures 6–7 show the modifications to the optimized EPaxos* (§4) that yield the thrifty version of the protocol (for simplicity, we do not consider applying the thrifty mode to the baseline protocol of §3). Like in §4, we assume $n \geq \max\{2e + f - 1, 2f + 1\}$.

Modifications to the commit protocol (Figure 6). In the thrifty version, the initial coordinator sends the `PreAccept` message to a single fast quorum, which must consist of exactly $n - e$ processes and includes the coordinator (line 10t). The fast path can be taken even with a dependency set D different from the initial one, as long as all the processes in the fast quorum except the initial coordinator agree on it (line 22t). An important restriction is that a command *id* cannot take the fast path if a fast quorum process detects that *id* must depend on a command in the INITIAL phase (line 17t). This situation can occur when,

```
 8  submit(c):
 9      let id = new_id()
10t     send PreAccept(id, c, {id' | cmd[id'] ⋈ c}) to any Q such that |Q| = n − e ∧ p ∈ Q
11  when received PreAccept(id, c, D) from q
12      pre: bal[id] = 0 ∧ phase[id] = INITIAL
13      cmd[id] ← c
14      initCmd[id] ← c
15      initDep[id] ← D
16      dep[id] ← D ∪ {id' | cmd[id'] ⋈ cmd[id]}
17t     if ∀id ∈ dep[id]. phase[id] ≠ INITIAL then
            phase[id] ← PREACCEPTED
            send PreAcceptOK(id, dep[id], OK) to q
18t     else
            send PreAcceptOK(id, dep[id], notOK) to q
19t 19t when received PreAcceptOK(id, D_q, OK_q) from all q ∈ Q
20      pre: bal[id] = 0 ∧ phase[id] = PREACCEPTED ∧ |Q| ≥ n − f
21      let D = ⋃_{q∈Q} D_q
22t     if |Q| ≥ n − e ∧ (∀q ∈ Q \ {p}. D_q = D ∧ OK_q = OK) then
23          send Commit(0, id, cmd[id], D) to all
24      else
25          send Accept(0, id, cmd[id], D) to all
```

Figure 6 Modifications to the commit protocol of the optimized EPaxos* (§4) that yield the thrifty version of the protocol: code at process p. Self-addressed messages are delivered immediately.

prior to receiving id, a process in the fast quorum has received a `Validate` message for a conflicting command id', stored its payload at line 86, but has not yet advanced id' out of the INITIAL phase. In this case the process adds an additional `notOK` argument to its reply (line 18t), which disables the fast path (line 22t). As we show in §D, this restriction is required to ensure liveness.

Modifications to the recovery protocol (Figure 7). Since in the thrifty version the fast path can be taken with any dependency set, we also need to adjust the recovery protocol:

- In the thrifty version, a new coordinator suspects that the command id may have taken a fast path only if all processes in the recovery quorum pre-accepted id with the same dependencies D, and there are at least $|Q| - e$ such processes (line 60t). Since the thrifty version uses a single fast quorum, any disagreement on pre-accepted dependencies at line 60t indicates that the command could not have taken the fast path.
- Each `Validate` message now includes an additional argument to carry the initial dependency set (lines 62t–63t, 84t, 88t): in the thrifty version this set may be different from the set D used in recovery (line 62t).
- In the thrifty version we impose an extra constraint on when the new coordinator can abort a command id if it receives a `Waiting` message for a potentially invalidating command (line 77 in Figure 5). This is because the proof of Lemma 12 (§4) assumes that the fast path can only be taken with the initial dependencies, which is no longer the case in the thrifty version. To deal with this, `Waiting` messages now also carry the suspected fast path dependencies of the command being recovered (line 71t). The new coordinator of a command id aborts this command only if it does not belong to the fast path dependencies of the potentially invalidating command (line 77t).

```
    ...
60t else if ∃D. ∃R. |R| ≥ |Q| − e ∧ (∀q ∈ Q. phase_q = PREACCEPTED ⟺ q ∈ R) ∧ (∀q ∈ R. dep_q = D) then
61      let R_max be the largest set R that satisfies the condition at line 60
62t     let (c, D, D_0) = (c_q, dep_q, initDep_q) for any q ∈ R
63t     send Validate(b, id, c, D, D_0) to all processes in Q
        ...
70      else
71t       send Waiting(id, D, |R_max|) to all
72        wait until
73          case ∃(id', _) ∈ I. phase[id'] = COMMITTED ∧ (cmd[id'] ≠ Nop ∧ id ∉ dep[id']) do
74            send Accept(b, id, Nop, ∅) to all
75          case ∀(id', _) ∈ I. phase[id'] = COMMITTED ∧ (cmd[id'] = Nop ∨ id ∈ dep[id']) do
76            send Accept(b, id, c, D) to all
77t          case ∃(id', _) ∈ I. (p received Waiting(id', D', k')) ∧ k' > n − f − e ∧ id ∉ D' do
78            send Accept(b, id, Nop, ∅) to all
79t          case p received RecoverOK(b, id, _, cmd, dep, _, phase) from q ∉ Q with
             phase = COMMITTED ∨ phase = ACCEPTED ∨ q = initCoord(id)
             ∨ (phase = PREACCEPTED ∧ dep ≠ D) do
80            if phase = COMMITTED then send Commit(b, id, cmd, dep) to all
81            else if phase = ACCEPTED then send Accept(b, id, cmd, dep) to all
82            else send Accept(b, id, Nop, ∅) to all
    ...
84t when received Validate(b, id, c, D, D_0) from q
85      pre: bal[id] = b
86      cmd[id] ← c
87      initCmd[id] ← c
88t     initDep[id] ← D_0
        ...
```

Figure 7 Modifications to the recovery protocol of the optimized EPaxos* (§4) that yield the thrifty version of the protocol: code at process p. Self-addressed messages are delivered immediately.

- Finally, to ensure liveness, we add an extra case in which a new coordinator recovering id with a dependency set D can abort the recovery when it receives an additional `RecoverOK` message from outside of the original recovery quorum (line 79 in Figure 5). Namely, the new coordinator can now abort the recovery of id if it receives `RecoverOK`(_, id, _, _, dep, _, PREACCEPTED) with $dep \neq D$ (line 79t): in this case id could not have taken the fast path with D, since some process in the fast quorum pre-accepted a different dependency set.

▶ **Theorem 15.** *The thrifty version of* EPaxos* *protocol implements an f-resilient 0-fast SMR protocol.*

C Starting Recovery

In Figure 8, we present the protocol that replica p uses to invoke recovery for a command id. The mechanism relies on per-command leader detectors: each command id has an associated leader detector $\Omega[id]$, which eventually outputs the same correct process at all replicas [5]. Replica p periodically checks whether id is committed. If not, it sends a `TryRecover` message to $\Omega[id]$ (line 91), asking the process currently believed to be the coordinator of id to impose its leadership. Upon receiving `TryRecover`(id), a replica checks whether it considers itself the coordinator of id and, if so, starts recovery (line 95).

```
89 periodically (with increased delays)
90   if phase[id] ≠ COMMITTED then
91     send TryRecover(id) to Ω[id]
92   else if cmd[id] = Nop ∧ p = initCoord(id) then
93     submit(initCmd[id])

94 when received TryRecover(id)
95   if p = Ω[id] then recover(id)
```

Figure 8 EPaxos*: recovery policy of a command id at a replica p.

Recall that in EPaxos*, a command id submitted with payload c may sometimes be committed as a Nop. To ensure that every command submitted by a correct process is eventually executed, a process resubmits its original payload if it detects that id was replaced by a Nop (lines 92–93). Note that commands are committed as Nop only during recovery, and this stops occurring once the system becomes synchronous.

D Correctness of EPaxos*

D.1 Proof of Invariants 1 (Agreement) and 2 (Visibility)

This proof covers all the variants of the protocol we presented: the baseline protocol (§3), its optimized version (§4), and its thrifty version (§B). In the following we make it clear which parts of the proof are relevant for which version of the protocol. The proof relies on the low-level invariants listed in Figure 9. We omit the proofs of Invariants 3–6, as they easily follow from the structure of the protocol.

Proof of Invariant 7 (all protocol versions). Message $\texttt{Commit}(0, id, c, D)$ is sent at line 23. Only the process owning ballot 0, that is $\texttt{initCoord}(id)$, can send this message. Since messages are delivered immediately, the precondition at line 20 guarantees that it happens at most once during an execution. We note Q_f the quorum used to trigger the corresponding handler at line 19, and Q_f^* the set $Q_f \setminus \{\texttt{initCoord}(id)\}$. Every process in Q_f^* replied $\texttt{PreAcceptOK}(id, c, D)$ to $\texttt{initCoord}(id)$. According to line 22 (line 22t in the thrifty version), Q_f^* contains at least $n - e - 1$ processes.

Suppose a process p executes line 60 (line 60t in the thrifty version) for some dependency set D'. According to line 50, this happens because a quorum Q of processes sent a RecoverOK message to p. According to line 60 (line 60t in the thrifty version), there exists $R \subseteq Q$ with $|R| \geq |Q| - e \geq n - f - e$ such that every process in R has pre-accepted id with $\mathsf{dep}[id] = D'$. Let us observe that:

$$|Q_f^*| + |Q| \geq n - e - 1 + n - f - e = 2n - (2e + f - 1) \geq n$$

However, $\texttt{initCoord}(id)$ is neither in Q_f^* (by definition), nor in Q (by line 58). Thus, there is some process in $Q_f^* \cap Q$. According to the pseudo-code of EPaxos*, $\mathsf{phase}[id]$ is either PREACCEPTED, ACCEPTED, COMMITTED at that process when it replies to p with a ValidateOK message. The last two phases would have triggered line 56 and line 54, respectively. Thanks to the precondition at line 12, a command reaches the PREACCEPTED phase only once at a process (Invariant 3). It follows that $D' = D$, as required. ◀

Proof of Invariant 8 (all protocol versions). Assume that a process p pre-accepts a command id with an initial payload c and a dependency set D, as well as a command id' with a

3. Each command is pre-accepted at most once by each process.
4. Once a process moves a command to the ACCEPTED or COMMITTED phase, it remains in one of these phases for the entire execution.
5. At every process and for every id, if $\mathsf{abal}[id] > 0$, then $\mathsf{phase}[id] \in \{\text{ACCEPTED}, \text{COMMITTED}\}$.
6. Assume that at time t a process p pre-accepts or validates a command id with a payload c (i.e., p executes line 13 or line 86). At any moment after t, the process p has $\mathsf{cmd}[id] = c$ or $\mathsf{cmd}[id] = \mathsf{Nop}$, with the latter possible only if $\mathsf{phase}[id] \neq \text{PREACCEPTED}$.
7. Assume that a process sends $\texttt{Commit}(0, id, c, D)$ on the fast path (line 23). If a process executes line 60 (or line 60t), then the condition in this line evaluates to true, and only for the dependency set D.
8. If a process sends $\texttt{PreAcceptOK}(id, D)$ and $\texttt{PreAcceptOK}(id', D')$ for two commands with conflicting initial payloads, then $id \in D'$ or $id' \in D$.
9. Assume that a quorum of processes has received $\texttt{Accept}(b, id, c, D)$ and replied with $\texttt{AcceptOK}(b, id)$.
   a. For any $\texttt{Accept}(b', id, c', D')$ sent, if $b' \geq b$, then $c' = c$ and $D' = D$;
   b. For any $\texttt{Commit}(b', id, c', D')$ sent, if $b' \geq b$, then $c' = c$ and $D' = D$.
10. Assume that a process sends $\texttt{Commit}(0, id, c, D)$ at line 23.
    a. For any $\texttt{Accept}(\_, id, c', D')$ sent, $c' = c$ and $D' = D$;
    b. For any $\texttt{Commit}(\_, id, c', D')$ sent, $c' = c$ and $D' = D$.

Figure 9 Additional invariants of EPaxos*. All invariants hold for all three versions of EPaxos*.

conflicting initial payload c' and a dependency set D'. Without loss of generality, assume that the former happens before the latter. By the time p sends $\texttt{PreAcceptOK}(id', D')$ it has $\mathsf{cmd}[id] = c$ or $\mathsf{cmd}[id] = \mathsf{Nop}$ (Invariant 6). Then, since $c \bowtie c'$ and $\mathsf{Nop} \bowtie c'$, by line 16, $id \in D'$, as required. ◀

We have already proved Lemma 9 in §3.3 and Lemma 11 for the optimized protocol in §4. We now prove Lemma 11 for the thrifty protocol.

Proof of Lemma 11 (thrifty protocol). By contradiction, assume that id takes the fast path with a fast quorum Q_f. Hence, $|Q_f| \geq n - e$, all processes in Q_f except $\texttt{initCoord}(id)$ pre-accepted id with a dependency set D (line 22t), and $\texttt{initCoord}(id)$ pre-accepted id with a set $D_0 \subseteq D$ (because it handles its own PreAccept message immediately). Before executing line 68, a recovering process executes line 60t, and satisfies the condition at this line for D (Invariant 7). Hence, $\forall (id', \_) \in I.\, id' \notin D$ (line 89). By the assumption of the proposition, $|R_{\max}| = |Q| - e$ and for some id' we have $(id', phase') \in I$, where $\texttt{initCoord}(id') \notin Q$ and $phase' \neq \text{COMMITTED}$. Let D' be the initial dependency set of id'. Notice that by lines 15 and 88, whenever a process has $\mathsf{initCmd}[id'] \neq \bot$, it also has $\mathsf{initDep}[id'] = D'$. Moreover, by line 89, the process whose ValidateOK message resulted in $(id', phase')$ being included into I must have had $id \notin \mathsf{initDep}[id'] = D'$; and 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. Moreover, by line 58, $\texttt{initCoord}(id')$ is not among these e processes. But then since $|Q_f| \geq n - e$, all processes that are not in Q must belong to Q_f. Then since $\texttt{initCoord}(id') \notin Q$, we must have $\texttt{initCoord}(id') \in Q_f$. Thus, $\texttt{initCoord}(id')$ pre-accepted id with either D or $D_0 \subseteq D$, and it also pre-accepted id' with D'. But then,

since the initial payloads of id and id' conflict, by Invariant 8, either $id' \in D$ or $id \in D'$, which contradicts the facts we have established before. ◀

▶ **Lemma 16** (thrifty protocol, couterpart of Lemma 12). *Assume that $n \geq 2e + f - 1$ and that $\texttt{Waiting}(id, D, k)$ and $\texttt{Waiting}(id', D', k')$ have been sent for commands with conflicting initial payloads such that $id \notin D'$ and $id' \notin D$. If $k' > n - f - e$, then id did not take the fast path.*

Proof. By contradiction, assume that id takes the fast path with a fast quorum Q_f. Hence, $|Q_f| \geq n - e$, all processes in Q_f except $\texttt{initCoord}(id)$ pre-accepted id with a dependency set $\hat{D}$ (line 22t), and $\texttt{initCoord}(id)$ pre-accepted id with a set $D_0 \subseteq \hat{D}$ (because it handles its own `PreAccept` message immediately). By Invariant 7, the condition at line 60t is satisfied for $\hat{D}$. Since $\texttt{Waiting}(id, D, k)$ has been sent, it must be the case that $\hat{D} = D$. Since $\texttt{Waiting}(id', D', k')$ 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 dependencies D' (line 60t). Moreover, $\texttt{initCoord}(id')$ pre-accepted id' with a set $D'_0 \subseteq D'$, and line 58 ensures that $\texttt{initCoord}(id') \notin R'$. Since $n \geq 2e + f - 1$, the intersection of Q_f and $R' \cup \{\texttt{initCoord}(id')\}$ has a cardinality

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

Thus, some process pre-accepted id with either D or $D_0 \subseteq D$, and this process also pre-accepted id' with either D' or $D'_0 \subseteq D'$. But then by Invariant 8, either $id' \in D$ or $id \in D'$, contradicting our assumption. ◀

Proof of Invariant 9a (all protocol versions). Assume that a quorum of processes Q_0 has received $\texttt{Accept}(b, id, c, D)$ and replied with $\texttt{AcceptOK}(b, id)$. We prove by induction on b' that for any $\texttt{Accept}(b', id, c', D')$ sent, if $b' \geq b$, then $c = c'$ and $D = D'$. The base case of $b' = b$ trivially holds because the `Accept` message is sent at most once per ballot and each ballot is owned by a single process.

For the induction step we focus on the handler at line 50, as this is the only handler that causes processes to send `Accept` messages for ballots greater than b. Assume that process p executes this handler upon receiving $\texttt{RecoverOK}(b', id, abal_q, c_q, dep_q, initDep_q, phase_q)$ from each process q in a quorum Q. Let $b_{\max} = \texttt{max}\{abal_q \mid q \in Q\}$ be the ballot computed at line 52. Since $Q_0 \cap Q \neq \varnothing$, there exists a process q that sends both messages $\texttt{AcceptOK}(b, id)$ and `RecoverOK`. When q sends `RecoverOK` it sets $\texttt{bal}[id]$ to b'. Since $b' > b$, by the precondition at line 27, process q has to send $\texttt{AcceptOK}(b, id)$ before sending `RecoverOK`. Hence, $b_{\max} \geq b$.

Assume first that $b_{\max} = b$. After sending $\texttt{AcceptOK}(b, id)$, the process q has $\texttt{phase}[id] \in \{\text{ACCEPTED}, \text{COMMITTED}\}$ (Invariant 4). Hence, q satisfies either the condition at line 54 or the condition at line 56. Thus, process p sends $\texttt{Accept}(b', id, c', D')$ from line 57. Consider any $q' \in Q$ satisfying the condition at line 56. Since $abal_{q'} = b_{\max} = b$, process q' stores the payload and the dependencies that it received in the $\texttt{Accept}(b, id, c, D)$ message from the coordinator of id at b. Hence, $c' = c$ and $D' = D$, as required.

Assume now that $b_{\max} > b$; then $b_{\max} > 0$. Furthermore, by Invariant 5, at any process, any command with a ballot greater than 0 is either accepted or committed. This means that process p sends $\texttt{Accept}(b', id, c', D')$ from line 57. By line 56, there exists a process $q' \in Q$ with $phase_{q'} = \text{ACCEPTED}$, $abal_{q'} = b_{\max}$, $c_{q'} = c'$ and $dep_{q'} = D'$. The process q' thus received $\texttt{Accept}(b_{\max}, id, c', D')$, so by induction hypothesis, $c' = c$ and $D' = D$, as required. ◀

Proof of Invariant 9b (all protocol versions). The proof is established by induction. A $\texttt{Commit}(b', id, c', D')$ message is sent either at line 23, 36, or 54. Let p be the corresponding process. In the first case, $b' = 0$ due to line 19, leading to $b = 0$. It follows that p executes line 25, not line 23; a contradiction. Next, suppose the `Commit` message is sent at line 36. As the precondition in line 34 holds, p received a quorum of $\texttt{AcceptOK}(b', id)$ messages. It follows that p sends an $\texttt{Accept}(b', id, c', D')$ message previously. Applying Invariant 9a, $c' = c$ and $D' = D$, as required. Finally, suppose p sends `Commit` at line 54. At line 50, p receives $\texttt{RecoverOK}(b', id, abal_q, c_q, dep_q, initDep_q, phase_q)$ from each process q in some quorum Q. Let $b_{\max} = \texttt{max}\{abal_q \mid q \in Q\}$ be the ballot computed at line 52. As in the proof of Invariant 9a, it must be the case that $b_{\max} \geq b$. If $b_{\max} > b$, we apply our induction hypothesis. Otherwise, suppose $b_{\max} = b$. For some $q \in Q$, the coordinator of id at ballot b sends a $\texttt{Commit}(b, id, c_q, D_q)$ message. This process is the one who sends the $\texttt{Accept}(b, id, c, D)$ message. Hence, $c = c_q = c'$ and $D = D_q = D'$, as required. ◀

▶ **Proposition 17** (all protocol versions). *At every process, for every id such that* $c = \mathsf{cmd}[id] \neq \mathsf{Nop}$, $\mathsf{phase}[id] = \text{COMMITTED}$ *and* $\mathsf{dep}[id] = D$, *one of the two properties holds:*

1. *There exists a quorum Q such that every process $q \in Q$ sent a* $\texttt{PreAcceptOK}(id, D_q)$ *message with* $\bigcup_{q \in Q} D_q = D$;
2. *A process sent an* $\texttt{Accept}(\_, id, c, D)$ *from line 67 or line 76.*

Proof. By induction on the length of the execution. It is easy to see that the proposition holds initially. For the induction step, suppose a process p either *(i)* commits id, that is set $\mathsf{phase}[id]$ to COMMITTED, or *(ii)* changes $\mathsf{cmd}[id]$ to a non-Nop value once id is committed. We examine in turn each situation.

Situation *(i)* may only happen at line 42, after p received a $\texttt{Commit}(\_, id, c, D)$ message. This message is sent either at line 23, line 36, or line 55, by some process, say $\hat{p}$. In the first case, $\hat{p}$ received a $\texttt{PreAcceptOK}(id, D_q)$ message from some quorum Q at line 19, with $\bigcup_{q \in Q} D_q = D$, as required. If line 55 occurs, then the proposition is true by the induction hypothesis. Now, suppose $\hat{p}$ executes line 36. If c equals Nop, the invariant holds. Otherwise, $\hat{p}$ received a quorum of `AcceptOK` messages. These messages replied to an `Accept` message sent previously by $\hat{p}$ either at line 25 in the commit protocol, or one of the following lines in the recovery protocol: 57, 59, 67, 69, 74, 76, 81, or 83. If either line 67 or line 76 happened, the proposition holds up to that point in time. Now if line 57 or 81 triggered, then a process received an `Accept` message in ballot $b_{\max}$ and the coordinator of id at that ballot executed line 67 or 76, as required. Because $c \neq \mathsf{Nop}$. all the other cases cannot happen.

Next, consider situation *(ii)*. $\mathsf{cmd}[id]$ is modified at line 13, 29, or 40 in the commit protocol, and at line 86 in the recovery protocol. Line 13 cannot happen because id is already committed. This is also the case with line 86. Indeed, since p already committed id, the coordinator of id (at the ballot where this occurred) should evaluate line 54 to true. In case line 29 occurs, process p received a message `Accept` sent from either line 67 or 76, as required. Finally, the remaining case, at line 40, boils down to situation *(i)* above. ◀

▶ **Lemma 18.** *If Invariant 10a holds until time t then so does Invariant 10b.*

Proof. The proof is by induction over the length of the execution. Suppose a message $\texttt{Commit}(b, id, c, D)$ is sent at time t by a process p. If there is no other such message, Invariant 10b vacuously holds. Otherwise, assume that $\texttt{Commit}(b', id, c', D')$ is sent at an earlier time $t' < t$. We prove that if Invariant 10a holds until time t and either $b = 0$ or $b' = 0$, then $c = c'$ and $D = D'$. We perform a case split based on the values of b and b'.

- $(b = b' = 0)$ This case is trivial since only the initial coordinator may take the fast (line 23) or slow (line 36) path for id at ballot 0.
- $(b' = 0,\ b > 0)$ Because $b > 0$, $\texttt{Commit}(b, id, c, D)$ is sent either at line 36, 55 or 80. In the last two cases, id is committed at some process with $\mathsf{cmd}[id] = c$ and $\mathsf{dep}[id] = D$. Hence, this process received a $\texttt{Commit}(\_, id, c, D)$ message at an earlier time t''. At time $max(t', t'') < t$, by our induction hypothesis Invariant 10b holds. This gives us $c = c'$ and $D = D'$, as required. Consider now that $\texttt{Commit}(b, id, c, D)$ is sent in the slow path, at line 36. According to line 34, p received a message $\texttt{AcceptOK}(b, id)$ from some quorum. This message was a reply to an $\texttt{Accept}(b, id, c'', D'')$ message p sent at an earlier time $t'' < t$. Since Invariant 10a holds at time t'', $c'' = c'$ and $D'' = D'$. Moreover, because this message is received immediately, p has $\mathsf{cmd}[id] = c'$, $\mathsf{dep}[id] = D'$ and $\mathsf{bal}[id] = b$ at time t''. Observe that at time t, this must still hold. Hence, $c = c'$ and $D = D'$, as required.
- $(b = 0,\ b' > 0)$ Process p takes either the fast (line 23) or the slow (line 36) path for id at ballot 0. To take the fast (respectively, slow) path, p receives a quorum Q of $\texttt{PreAcceptOK}(0, id, \_)$ (resp., $\texttt{AcceptOK}(0, id)$) messages. Similarly, since $\texttt{Commit}(b', id, c', D')$ is sent, there must exist a quorum Q' of $\texttt{AcceptOK}(b'', id)$ messages, with $0 < b'' \leq b'$. Observe that for any process $q \in Q \cap Q'$, q sends its $\texttt{AcceptOK}(0, id)$ before $\texttt{AcceptOK}(b'', id)$. Let $\hat{q}$ be the last such process (there must be one) and $\hat{t}$ the time at which this happens. The key observation is that between $\hat{t}$ and t process p cannot take a single step without violating the precondition to commit id at ballot 0, that is, lines 19 and 35 for the fast and slow paths, respectively. Hence, the execution is equivalent to an execution in which p sends $\texttt{Commit}(b, id, c, D)$ before $\texttt{Commit}(b', id, c', D')$ was sent. It follows that this case reduces to the previous one.

◀

Proof of Invariant 10a (all protocol versions). Assume that a process sends $\texttt{Commit}(0, id, c, D)$ at line 23, i.e., id takes the fast path with payload c and dependency set D. Let Q_f be the fast quorum used for this. We prove by induction on the length of the execution that for any $\texttt{Accept}(b, id, c', D')$ message sent by a process p, $c' = c$ and $D' = D$. For starters, let us observe that if $b = 0$, then the message must be sent at line 25, that is in the slow path. In such a case, necessarily $p = \texttt{initCoord}(id)$, i.e., p is the initial coordinator, and the result is immediate. Now, consider the handler at line 50, which is the only handler that results in process p sending the $\texttt{Accept}$ message with $b > 0$. Let Q be a quorum of $\texttt{RecoverOK}(b, id, abal_q, c_q, dep_q, initDep_q, phase_q)$ messages with which p executes the handler at line 50, and let $b_{\max} = \texttt{max}\{abal_q \mid q \in Q\}$ be the ballot the process computes at line 52. Process p sends $\texttt{Accept}(b, id, c', D')$ from one of the following lines: 57, 59, 67, 69, 74, 76, 78, 81, 82, or 83. We now make a case split on these options.

- *Lines 57 (baseline protocol), 81 (optimized and thrifty protocols).* In all these cases process p sends the $\texttt{Accept}$ message with the payload c' and dependency set D' received from a process q that previously received $\texttt{Accept}(b', id, c', D')$ message. But then by induction hypothesis, $c = c'$ and $D = D'$, as required.
- *Line 59 (optimized and thrifty protocols).* Since the conditions at lines 54 and 56 do not hold, by Invariant 5 we have $b_{\max} = 0$. Hence, the coordinator $\texttt{initCoord}(id) \in Q$ sends a message of the form $\texttt{RecoverOK}(b, id, 0, c, \_, \_, phase)$. We know that it also sends $\texttt{Commit}(0, id, c, D)$ at line 23. Moreover, by lines 20 and 47–48, the coordinator sends $\texttt{Commit}$ before $\texttt{RecoverOK}$. Since all self-addressed messages are delivered immediately, it

follows that $phase = \text{COMMITTED}$. However, because the condition at line 54 does not hold, it must be that $phase \neq \text{COMMITTED}$: a contradiction.

- *Line 82 (optimized protocol).* In this situation, the case at line 79 is triggered. Since the conditions at lines 80 and 81 do not hold, by Invariant 5, process p receives a $\texttt{RecoverOK}(b, id, 0, c, \_, \_, phase)$ message from the coordinator $\texttt{initCoord}(id)$. However, before sending RecoverOK the coordinator also sends $\texttt{Commit}(0, id, c, D)$ at line 23. Since all self-addressed messages are delivered immediately, it follows that $phase = \text{COMMITTED}$, contradicting the fact that the condition at line 80 does not hold.
- *Line 82 (thrifty protocol).* The conditions at lines 80 and 81 do not hold. By line 79t, process p received either $\texttt{RecoverOK}(b, id, 0, c, \_, \_, phase)$ from $\texttt{initCoord}(id)$, or $\texttt{RecoverOK}(b, id, 0, c, D'', \_, \text{PREACCEPTED})$ from a process $q \neq \texttt{initCoord}(id)$, with $D'' \neq D$. The proof of the former case is identical to the proof above for the optimized protocol. For the latter case, since q has pre-accepted id, it belongs to the fast quorum of id (line 10). But then, since $q \neq \texttt{initCoord}(id)$ and $D'' \neq D$, by line 22t and Invariant 7, the initial coordinator cannot take the fast path: a contradiction.
- *Lines 67, 76 (all protocol versions).* In this case D' satisfies the condition at line 60 (or line 60t in the thrifty version). Hence, $D' = D$ (Invariant 7) and $c' = c$ (Invariant 6), as required.
- *Line 69, executed due to the first disjunct of line 68, and line 74 (all protocol versions).* Before sending the Accept message from one of these lines, p has to satisfy the condition at line 60 (60t). By Invariant 7, this condition can only be satisfied for the dependency set D. By lines 68 and 73, and line 89, there exists a process q that has committed an invalidating command id'' with a payload c'' and a dependency set D'', such that $c \bowtie c'' \neq \mathsf{Nop}$, $id \notin D''$ and $id'' \notin D$.

  According to Proposition 17, D'' is computed either from a quorum of processes Q'' that pre-accepted id'', or during the recovery of id'', after executing line 67 or line 76. The former is impossible: in this case, since $Q_f \cap Q'' \neq \varnothing$, by Invariant 8 we would have $id \in D''$ or $id'' \in D$.

  Thus, some process q'' has executed line 67 or line 76 for id'' and D''. Before that, q'' must have satisfied the condition at line 60 (60t) with D'' after receiving a quorum Q'' of RecoverOK messages. Since $Q_f \cap Q'' \neq \varnothing$, there exists a process q that pre-accepts id and sends a $\texttt{ValidateOK}(\_, id'', I'')$ message (line 64). Notice that q must have pre-accepted id before sending $\texttt{ValidateOK}(\_, id'', I'')$: otherwise, by Invariant 6, it would have included id'' in its dependency set at line 16, implying $id'' \in D$. We now prove that $id \in I''$, i.e., all three conjuncts at line 89 hold for id at q when it computes I'':

  (i) $id \notin D''$;
  (ii) $\mathsf{phase}[id] = \text{COMMITTED} \implies \mathsf{cmd}[id] \neq \mathsf{Nop} \wedge \mathsf{cmd}[id] \bowtie c'' \wedge id'' \notin \mathsf{dep}[id]$; and
  (iii) $\mathsf{phase}[id] \neq \text{COMMITTED} \implies \mathsf{initCmd}[id] \neq \bot \wedge \mathsf{initCmd}[id] \bowtie c'' \wedge id'' \notin \mathsf{initDep}[id]$.

  Conjunct (i) holds immediately from the equations established earlier. For conjunct (ii), by Lemma 18 applied to the induction hypothesis, q must have committed id with command c and dependency set D. We know $c \neq \mathsf{Nop}$, $c \bowtie c''$, and $id'' \notin D$, as required. For conjunct (iii), notice that by the time q sends $\texttt{ValidateOK}(\_, id'', I'')$ it has already pre-accepted id. Thus, by lines 14–15, at that moment q has $\mathsf{initCmd}[id] = c$ and $\mathsf{initDep}[id] = D_0$, where D_0 is the initial dependency set of id. Since initial dependencies are always included in final dependencies, and $id'' \notin D$, it follows that $id'' \notin D_0$ as required.

  Now that we have proved that $id \in I''$, it follows that q'' did not execute line 67, and therefore executes line 76. However, since $id \in I''$, by line 75, id is committed either

with a payload Nop or with dependencies that include id''. The former case contradicts Lemma 18 applied to the induction hypothesis and the fact that a process went on the fast path for id. In the latter, this contradicts that $id'' \notin D$ and Invariant 7.

- *Line 69, executed due to the second disjunct of line 68 (optimized and thrifty protocols).* This case is impossible as it contradicts Lemma 11.
- *Line 78 (baseline protocol).* By line 77, there exists a process $q \in Q$ that sends a $\texttt{Validate}(\_, id', I')$ message, and there also exists a process that sends a $\texttt{Waiting}(id')$ message. We know that when q sends Validate, id' is not committed at q: otherwise, process p would have satisfied the condition at line 68, and thus would never have executed line 78. Thus, by line 89, the initial payload of id' conflicts with c, and id is not in the initial dependency set of id'. By the same line, we also have $id' \notin D$. But then, by Lemma 9, id did not take the fast path, contradicting our assumption. Therefore, this case is impossible.
- *Line 78 (optimized and thrifty protocols).* These cases are likewise impossible, by applying analogous reasoning as above and replacing Lemma 9 with Lemma 12 for the optimized protocol, or Lemma 16 for the thrifty version.
- *Line 83 (all protocol versions).* In this case the condition at line 60 (60t) does not hold for id, and thus this case is impossible as it contradicts Invariant 7.

◀

Proof of Invariant 10b (all protocol versions). Follows from Invariant 10a and Lemma 18. ◀

Proof of Invariant 1 (all protocol versions). Suppose a command id committed by two processes p and p'. Process p (respectively, p') commits id with the payload c (resp. c') and dependency set D (resp., D'). Committing a command happens in the handler at line 37, after a Commit message is received. Let $\texttt{Commit}(b, id, c, D)$ and $\texttt{Commit}(b', id, c', D')$ be the messages received by p and p', respectively. Observe that a single process may send a Commit message at ballot 0, the initial coordinator. Hence, the case $b = b' = 0$ is trivial. Assume now that $b = 0$ and $b' > 0$. Applying Invariant 10, we have $c = c'$ and $D = D'$, as required. The case $b' = 0$ and $b > 0$ is symmetrical to the previous one and thus omitted. It remains to investigate the case $b > 0$ and $b' > 0$. In such a case, the Commit message is sent at line 36, 55, or 80. With the last two situations, id was already committed at another process and we may thus proceed by induction, using our previous analysis. As a consequence, it is supposed hereafter that both messages are sent at line 36. For the handler at line 34 to execute, a process needs a quorum of AcceptOK messages. Hence, for each $\texttt{Commit}(b, id, c, D)$ and $\texttt{Commit}(b', id, c', D')$ message, there exists a corresponding message $\texttt{Accept}(b, id, c, D)$ and $\texttt{Accept}(b', id, c', D')$. The result then holds by Invariant 9. ◀

Proof of Invariant 2 (all protocol versions). Assume that messages $\texttt{Commit}(\_, id, c, D)$ and $\texttt{Commit}(\_, id', c', D')$ are sent, where $c \neq \mathsf{Nop}$, $c' \neq \mathsf{Nop}$ and $c \bowtie c'$. We prove that $id \in D'$ or $id' \in D$. Notice that both commands satisfy one of the two conditions of Proposition 17.

1. Assume that both commands satisfy condition 1 of Proposition 17. Then there are two quorums Q and Q' such that every process $q \in Q$ (resp., $q \in Q'$) has sent a $\texttt{PreAcceptOK}(id, D_q)$ message (resp., $\texttt{PreAcceptOK}(id', D'_q)$) with $\bigcup_{q \in Q} D_q = D$ (resp., $\bigcup_{q \in Q'} D'_q = D'$). Since $Q \cap Q' \neq \varnothing$, there exists a process that sends both $\texttt{PreAcceptOK}(id, D_q)$ and $\texttt{PreAcceptOK}(id', D'_q)$. By Invariant 8, $id \in D'_q \subseteq D'$ or $id' \in D_q \subseteq D$, as required.

2. Assume that id' satisfies condition 1 of Proposition 17, while id satisfies condition 2 (the symmetric case is analogous). Then there exists a quorum Q' such that every process $q \in Q'$ has sent a $\texttt{PreAcceptOK}(id', D'_q)$ message with $\bigcup_{q \in Q'} D'_q = D'$. Furthermore, a process p has sent an $\texttt{Accept}(\_, id, c, D)$ from line 67 or line 76. Let Q be a quorum that enables the corresponding invocation of the handler at line 50 for id. Prior to executing line 67 or line 76, process p has received $\texttt{ValidateOK}(\_, id, I_q)$ from every process $q \in Q$ (line 64). Take a process $q \in Q \cap Q'$; then this process sends $\texttt{PreAcceptOK}(id', D'_q)$ and $\texttt{ValidateOK}(\_, id, I_q)$. If q sends $\texttt{ValidateOK}(\_, id, I_q)$ first, then by line 86 and Invariant 6, at the moment q sends $\texttt{PreAcceptOK}(id', D'_q)$ it has $\mathsf{cmd}[id] = c$ or $\mathsf{cmd}[id] = \mathsf{Nop}$. Since c and Nop both conflict with c', by line 16, $id \in D'_q$, as required.
   Consider now the remaining case in which q sends $\texttt{PreAcceptOK}(id', D'_q)$ before $\texttt{ValidateOK}(\_, id, I_q)$. Then by lines 14–15 we know that q has $\mathsf{initCmd}[id'] = c' \bowtie c$ when it sends $\texttt{ValidateOK}$. We make a case split depending on whether $id' \in I_q$ or not. Assume first that $id' \notin I_q$: the command id' is not selected at line 89 upon validation of id. Then one of the following must hold at process q:
   (i) $id' \in D$; or
   (ii) $\mathsf{phase}[id'] = \textsc{committed}$ and $\mathsf{cmd}[id'] = \mathsf{Nop} \vee \mathsf{cmd}[id'] \not\bowtie c \vee id \in \mathsf{dep}[id']$; or
   (iii) $\mathsf{phase}[id'] \neq \textsc{committed}$ and $id \in \mathsf{initDep}[id']$.
   In case (i) the required property follows immediately. In case (ii), Invariant 1 ensures that process q commits id' with $\mathsf{cmd}[id'] = c' \neq \mathsf{Nop}$ and $\mathsf{dep}[id'] = D'$, which implies $id \in D'$, as required. In case (iii), recall that when q sends the $\texttt{ValidateOK}$ message it has $\mathsf{initCmd}[id'] \neq \bot$. But then by lines 15 and 88, it also has $\mathsf{initDep}[id'] = D'_0 \subseteq D'$, where D'_0 is the initial dependency set of id'. Thus, $id \in D'$, as required.
   Assume now that $id' \in I_q$: the command id' potentially invalidates id. Notice that since $I_q \neq \varnothing$, process p could not have executed line 67. Thus it sends $\texttt{Accept}(\_, id, c, D)$ at line 76. By line 75 it has committed id' either with a dependency that contains id, or with Nop. However, by Invariant 1, it has to be committed with $c' \neq \mathsf{Nop}$ and D'. Hence, $id \in D'$, as required.
3. Assume that both commands satisfy condition 2 of Proposition 17. In this case, they both validate their dependencies at their recovery quorums at lines 63–64. Thus, there exists a process that sends $\texttt{ValidateOK}(\_, id, I)$ and $\texttt{ValidateOK}(\_, id', I')$. Assume that it first sends $\texttt{ValidateOK}(\_, id', I')$ (the symmetric case is analogous). By line 87, at the moment it sends $\texttt{ValidateOK}(\_, id, I)$ it has $\mathsf{initCmd}[id'] = c' \neq \bot$. The proof concludes with the same case-splitting as before: either id' belongs to I, or it does not.

◀

D.2 Proof of Consistency

Below, we prove that the execution protocol in Figure 1 maintains the Consistency property of the replicated state-machine. To this end, we use two auxiliary lemmas. These lemmas hold for any two committed commands id and id' with conflicting non-Nop payloads c and c', respectively.

▶ **Lemma 19.** *Assume that a process p executes c before executing c', and if the process executes both commands, then this happens in different iterations of the loop at line 1. Then $id' \notin \mathsf{dep}[id]$.*

Proof. By contradiction, suppose $id' \in \mathsf{dep}[id]$. Let G be the subgraph computed during the iteration of the loop when id is executed, so that id belongs to G (line 2). By the definition

of G, when p executes c, this process has committed c and all its transitive dependencies. It follows that id' also belongs to G. By line 5, by the end of this iteration of the loop, p will execute c'. But then either c' is executed in the same iteration of the loop as c, or it was executed before c: a contradiction. ◀

▶ **Lemma 20.** *If c is executed before c' in the same iteration of the loop at line 1, then there is a path from id to id' in the subgraph G computed at line 2.*

Proof. By contradiction, assume that there is no a path from id to id' in G. This implies that id is not a transitive dependency of id'. Then by Invariants 1 and 2, id' must be a dependency of id. Thus, id and id' must be in different strongly connected components of G, and the one containing id' must be ordered before the one containing id at line 3. Then c' is executed before c: a contradiction. ◀

Proof of Consistency. The proof is done by contradiction. Consider two commands with conflicting non-Nop payloads c and c' and whose identifiers are id and id', respectively. Suppose that a process p executes c before executing c', while another process q does the opposite. There are two possible execution paths at p:

- If p executes c and c', then this happens in different iterations of the loop at line 1. By Lemma 19, we get $id' \notin \mathsf{dep}[id]$. Then by Invariants 1 and 2, we get $id \in \mathsf{dep}[id']$.
- p executes both c and c', and this happens in the same iteration of the loop at line 1. Then by Lemma 20, there is a dependency chain going from id to id'.

Thus, in both cases, there is a dependency chain going from id to id'. Considering process q and using the same arguments, we can show that there is a dependency chain from id' to id. Thus, id and id' always belong to the same strongly connected component at line 3, so are always executed in the order of their identifiers: a contradiction. ◀

D.3 Proof of Liveness

▶ **Proposition 21** (all protocol versions)**.** *For each command id, there is a time after* GST *and after all failures have occurred, such that id can be recovered only by a single process with the highest ballot* $\mathsf{bal}[id]$*.*

Proof of Lemma 13 (all protocol versions). The proof is done by contradiction. Assume by contradiction that some process never commits a command id. By Proposition 21, there exists a time $t \geq \mathsf{GST}$ after all failures have occurred, and after which id can be recovered only by a single process p. We may also assume that by time t, the recovery timer at p is sufficiently high (line 89), ensuring that p does not start a new recovery at line 89 before receiving all `RecoverOK` and `ValidateOK` messages from the correct processes to which it has sent `Recover` and `Validate`, respectively.

Since id is never committed, every recovery attempt by p after time t must get stuck. This can happen only because on each new attempt id blocks on at least one potentially invalidating command (line 72). Since we are considering only runs with finitely many submitted commands, there is a command $id' \in I$ (line 89) on which id blocks infinitely often. Just as process p is the only process that attempts to recover id after t, by Proposition 21, there exists a time $t' \geq \mathsf{GST}$ after which id' can be recovered only by a single process p'. We can assume that $t > t'$, and that by time t, the recovery timer at p' is sufficiently high (line 89), ensuring that p' does not start a new recovery at line 89 before receiving all `RecoverOK` and `ValidateOK` messages from the correct processes to which it has sent `Recover` and `Validate`, respectively.

Since id is blocked on id', and id' is never committed, id' must itself be blocked in recovery. Notice that in every recovery attempt after t, process p (resp., p') selects the same dependency set D (resp., D') at line 62. Indeed: *(i)* in the non-thrifty versions, D (resp., D') is the initial dependency set, which is unique (line 60); and *(ii)* in the thrifty version, once p (resp., p') selects some dependency set at line 62, any subsequent attempt in which it would select a different set must finish after executing lines 79t and 82t. Eventually, process p' broadcasts a `Waiting` message for id' at line 71 (71t in the thrifty version), which p eventually receives. This concludes the proof for the baseline protocol, since in this case p unconditionally aborts the recovery at line 78. We continue the proof for the optimized and thrifty versions.

Since id repeatedly blocks on id' (and in particular never satisfies the first disjunct at line 68), line 89 implies:

(i) $id' \notin D$,
(ii) the initial payloads of id and id' conflict, and
(iii) $id \notin D'_0$, where D'_0 is the initial dependency set of id'.

Moreover, after time t, the initial payload of id is stored at a recovery quorum of id (line 87). Since every recovery quorum of id intersects with every recovery quorum of id', on each recovery attempt after t, process p' receives a `ValidateOK` message for id' from at least one correct process q' that stores the initial payload of id. We now make a case split depending on whether $id \in D'$ or not.

- $id \notin D'$. Notice that q' never commits id: otherwise id's recovery would eventually complete at line 55 or 80. Then, due to *(i)–(iii)*, after time t, each time process q' receives a `Validate` message for id', it includes id in the set of invalidating and potentially invalidating commands at line 89. Thus, every recovery of id' by p' eventually computes a set I' at line 65 such that $id \in I'$. Due to line 71, 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 77 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 61, $|R_{\max}| \geq |Q| - e \geq n - f - e \geq |R_{\max}|$, so that $|R_{\max}| = |Q| - e$. Since the conditions at lines 58 and 79 (79t in the thrifty version) 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 68 will hold, letting p finish the recovery. This contradicts our assumption.
- $id \in D'$. Consider first the non-thrifty version of the protocol. Then we have $D'_0 = D'$, and hence, $id \in D'_0$, which contradicts our previous claims.
  Consider now the thrifty version. Since id does not potentially invalidate id' (because $id \in D'$), the latter must have been blocked on some other command. Thus, there exists a sequence of commands $(id_n)_{n \in \mathbb{N}}$, where each command is blocked by the preceding one. Recall that we are considering a run with only finitely many submitted commands. Therefore, the sequence $(id_n)_{n \in \mathbb{N}}$ must be circular.
  For each n, let D_n be the recovering dependency set of id_n. Notice that if there exists n, such that $id_n \notin D_{n+1}$, then the rest of the proof proceeds exactly as in the case where $id \notin D'$, which is proven above. Hence, for all n, $id_n \in D_{n+1}$.
  Consider any command id_m from $(id_n)_{n \in \mathbb{N}}$. Since id_m is blocked on recovery at line 72, by line 60t there exists at least one process q that has pre-accepted id_m with D_m. Since $id_{m-1} \in D_m$, process q stores command id_{m-1}. Eventually, the new coordinator of id_{m-1} receives a `RecoverOK`$(\_, id_{m-1}, \_, \_, dep_q, \_, phase_q)$ message from q. By line 17t and line 79t, we know that $phase_q =$ PREACCEPTED and $dep_q = D_{m-1}$. This means

that process q has pre-accepted both id_{m-1} and id_m, with dependencies D_{m-1} and D_m, respectively. Since $id_{m-2} \in D_{m-1}$, process q must also store id_{m-2}. Applying the same reasoning, we conclude that q has pre-accepted id_{m-2} with D_{m-2}. By repeating this argument for all commands in the sequence $(id_n)_{n\in\mathbb{N}}$, we conclude that process q has pre-accepted each command with its recovering dependencies. Notice that these dependencies form a cyclic dependency graph. However, processes cannot locally pre-accept cycles, which leads to contradiction.

◀

Proof of Liveness. As long as no process starts the recovery of a command, every submitted command is eventually delivered. If the run involves the recovery of a command id then, according to lines 90–91, processes will continue attempting to recover id until it is committed. By Lemma 13 we know that eventually there will be a time after which id is committed. Finally, by lines 92–93, if id is committed as a Nop, the correct process that originally submitted it will retry, eventually committing it with the initially submitted payload. ◀

D.4 Proofs of Theorems 10, 14 and 15

We prove all three theorems together. We have already proven that all three versions of EPaxos* satisfy Consistency (§D.2) and Liveness (§D.3) properties of the SMR specification (§2). It is easy to see that they also satisfy Validity and Integrity. Moreover, since Liveness holds in every execution with at most f crashes, all versions of EPaxos* are f-resilient. This immediately concludes the proof of Theorem 15, as the protocol is trivially 0-fast.

To conclude the proofs of Theorems 10 and 14, consider any E-faulty synchronous run with $|E| \leq e$. Suppose a conflict-free command id is submitted by some process p at time t, and let D be the initial dependency set that p includes in the `PreAccept` message for id. Because id is conflict-free, any conflicting command id' must either already be in D or cannot be submitted until after id is executed by some process. Thus, every `PreAcceptOK` message carries the same dependency set D. Since at most e processes may fail, p receives at least $n - e$ responses. By synchrony, p therefore executes id by time $t + 2\Delta$. Hence, the baseline and the optimal versions of EPaxos* are e-fast, which completes the proof.

E A Liveness Bug in EPaxos Recovery

We consider a system of $n = 9$ processes $p_1, \ldots, p_9$, tolerating at most $f = 4$ failures. The size of a fast quorum is $f + \lfloor \frac{f+1}{2} \rfloor = 6$. Process p_1 is the initial coordinator of a command c with an identifier id, and process p_2 is the initial coordinator of a conflicting command c' with an identifier id'. Following the protocol as specified in Moraru's PhD thesis [17], the counter-example is as follows.

1. Process p_1 sends $\texttt{PreAccept}(c, id, \varnothing, 0)$ to $\{p_1, p_5, p_6, p_7, p_8, p_9\}$.
2. Processes p_8 and p_9 pre-accept id with $\mathsf{dep}[id] = \varnothing$ and $\mathsf{seq}[id] = 0$.
3. Processes p_8 starts recovering id.
   a. It sends `Prepare` and receives $f + 1 = 5$ replies from $\{p_2, p_3, p_4, p_8, p_9\}$ (page 41, step 1).
   b. Steps 2 on page 41 and 3–4 on page 42 are not applicable.
   c. However, there are $\lfloor \frac{f+1}{2} \rfloor = 2$ processes (p_8 and p_9) that pre-accepted id with $\mathsf{dep}[id] = \varnothing$ and $\mathsf{seq}[id] = 0$ at ballot 0 (the "If" part of step 5 on page 42). Hence, p_8

sends `TentativePreAccept`$(id, c, \varnothing, 0)$ to $\{p_1, p_2, p_3, p_4\}$ (step 6 on page 42). For the moment, nobody receives this message.

4. Process p_2 sends `PreAccept`$(c', id', \varnothing, 0)$ to $\{p_2, p_5, p_6, p_7, p_8, p_9\}$.
5. Process p_2 pre-accepts id' with $\mathsf{dep}[id'] = \varnothing$ and $\mathsf{seq}[id'] = 0$.
6. Processes p_5, p_6, and p_7 all pre-accept id' and then id. That is, $\mathsf{dep}[id'] = \varnothing$, $\mathsf{seq}[id'] = 0$, and $\mathsf{dep}[id] = \{id'\}$, $\mathsf{seq}[id] = 1$.
7. Process p_2 starts recovering id'.
   a. It sends `Prepare` and receives replies from $f + 1 = 5$ processes $\{p_2, p_3, p_4, p_8, p_9\}$ (page 41, step 1).
   b. Steps 2 on page 41 and 3–4 on page 42 are not applicable.
   c. There is only one process (p_2) that pre-accepted id'. Hence, p_2 executes the "Else" part of step 5 on page 42, and restarts proposing id from Phase 1 of the protocol.
8. Processes p_3 and p_4 receive `TentativePreAccept`$(id, c, \varnothing, 0)$ from p_8 and (tentatively) pre-accept id with $\mathsf{dep}[id] = \varnothing$ and $\mathsf{seq}[id] = 0$.
9. Process p_2 successfully commits id' with $\mathsf{dep}[id'] = \{id\}$ and $\mathsf{seq}[id'] = 1$ on the slow path using a quorum $\{p_2, p_3, p_4, p_8, p_9\}$.
10. Processes p_1, p_2 and p_3 fail.
11. Process p_4 starts recovering of id. It broadcasts a `Prepare` message and receives $f + 1 = 5$ replies from $\{p_4, p_5, p_6, p_7, p_8\}$. Steps 2 on page 41 and 3–4 on page 42 are not applicable. However, there are $3 > \lfloor \frac{f+1}{2} \rfloor = 2$ processes (p_5, p_6 and p_7) that pre-accepted id with $\mathsf{dep}[id] = \{id'\}$ and $\mathsf{seq}[id] = 1$ at ballot 0 (the "If" part of step 5 on page 42). Hence, p_4 sends `TentativePreAccept`$(id, c, \{id'\}, 1)$. Command id' does not interfere with id. As a consequence, processes that receive this message simply update $\mathsf{dep}[id]$ and $\mathsf{seq}[id]$ accordingly and send back `TentativePreAcceptOK` to p_4.
12. Process p_8 recovers id again using $\{p_4, p_5, p_6, p_8, p_9\}$. (The previous recovery stalled because p_1 crashed.) There are 3 processes (p_4, p_8 and p_9) that pre-accepted id with $\mathsf{dep}[id] = \varnothing$ and $\mathsf{seq}[id] = 0$. As a consequence, p_8 sends `TentativePreAccept`$(id, c, \varnothing, 0)$. Notice that because id' does not interfere with id for such values of $\mathsf{dep}[id]$ and $\mathsf{seq}[id]$, processes that receive such a message update these values accordingly and send back a `TentativePreAcceptOK` to p_8.
13. The messages sent by p_4 and p_8 interleave in a manner that keeps the state of $\mathsf{dep}[id]$ and $\mathsf{seq}[id]$ at all the remaining processes identical to before p_4 starts recovering (i.e., item 10 above).
14. Process p_4 receives `TentativePreAcceptOK` messages from the quorum. Observe that there are at most $4 = f$ potential processes that pre-accepted id with $\mathsf{dep}[id] = \{id'\}$ and $\mathsf{seq}[id] = 1$ (that is, p_5, p_6 and p_7, and potentially the coordinator of id, p_1 – on page 42 in [17] step 7(a) reads *"we can count Q here too even if it does not reply"*). Moreover, there are no interfering commands (step 6) since the only other submitted command c' is committed with $\mathsf{dep}[id'] = \{id\}$. Hence, the steps listed on pages 42–43 do not hold. Recovery at p_4 is blocked. By a similar line of arguments, recovery is also blocked at p_8.
15. Items 11–14 can be repeated forever for any two processes storing different values of $\mathsf{dep}[id]$ and $\mathsf{seq}[id]$. As a consequence, recovery does not make progress.