Tolerating Slowdowns in Replicated State Machines using Copilots: Pseudocode and Proof of Correctness

Khiem Ngo[⋆], Siddhartha Sen[†], Wyatt Lloyd[⋆]
[⋆]*Princeton University,* [†]*Microsoft Research*

Abstract

This technical report presents the pseudocode and a proof of correctness for Copilot replication, a new consensus protocol that achieves 1-slowdown-tolerance: it delivers normal latency despite the slowdown of any 1 replica. Copilot achieves this by using two distinguished replicas—the pilot and copilot—to proactively add redundancy to all stages of processing a client's command. We give an overview of the Copilot protocol and then present its pseudocode. We also provide a proof of correctness in the asynchronous crash failure model, showing that: Copilot requires $2f+1$ replicas to tolerate at most f crash failures, while guaranteeing safety (linearizability) despite any number of failures, and liveness as long as a majority of replicas communicate in a timely manner. These guarantees are the same as those of Multi-Paxos and EPaxos, which Copilot draws inspiration from. However, Copilot is the only protocol that provides 1-slowdown-tolerance in the presence of any one slow replica.

For a detailed discussion of Copilot's design, optimizations, and evaluation, please refer to the conference paper.[1]

1 Introduction

Replicated state machines (RSMs) are used to implement small services that require strong consistency and fault tolerance [25], and are used throughout large-scale systems such as distributed databases [7, 9], cloud storage [4, 5], and service managers [15, 22]. While each RSM is individually small, their widespread use at scale means that it is common for some machines to be slow [2, 10]. Slowdowns arise for a myriad of reasons, including misconfigurations, host-side network problems, partial hardware failures, garbage collection events, and many others. Unfortunately, no existing consensus protocol tolerates all types of slowdowns.

Slowdowns can be transient, lasting only a few seconds to minutes, or they can be long-term, lasting hours to days. Monitoring mechanisms within and around a system will detect some long-term slowdowns and reconfigure the slow replica out of the RSM to restore normal performance [1, 3, 13, 14, 18, 19]. What remains unsolved is how to tolerate transient slowdowns, or long-term slowdowns that are not detected, or long-term slowdowns that are detected in the time between their onset, eventual detection, and the end of reconfiguration.

Slowdown tolerance. We seek a consensus protocol that can tolerate the slowdowns mentioned above. To formalize this task, we introduce the notion of s-slowdown tolerance. An RSM is *s-slowdown-tolerant* if it provides a service that is not slow despite s replicas being slow. A replica is *slow* when its responses to messages take more than a threshold time t over its normal response time. The precise setting of t depends on the scenario and may even vary over time; we assume the term "slow" captures the current definition and build the notion of slowdown tolerance on top of this term. More specifically, sort the replicas $\{r_1, ..., r_s, ..., r_n\}$ of an RSM according to the current definition of slow, such that $\{r_1, ..., r_s\}$ are the s slowest replicas. Let T represent how slow the RSM is and T' represent how slow the RSM would be if replicas $\{r_1, ..., r_s\}$ were all replaced by clones of r_{s+1}. An RSM is s-slowdown-tolerant if the difference between T and T' is close to zero. In other words, the presence of s slow replicas should not appreciably slow down the RSM relative to an ideal scenario where those s replicas are not slow. This work focuses on the practical case of 1-slowdown-tolerance.

Copilot vs. existing protocols. To provide 1-slowdown-tolerance, a consensus protocol must be able to tolerate a slowdown in all stages of processing a client's command: receive, order, execute, and reply. No existing consensus protocol is 1-slowdown-tolerant because none can handle a slow replica in the ordering stage. Existing ordering protocols all either rely on a single leader [3, 6, 16] or rely on the collaboration of multiple replicas [20, 23]. A single leader is not slowdown-tolerant because if it is slow, then it slows down the RSM. Multiple replicas collaboratively ordering commands is not slowdown-tolerant because if any of those replicas is slow, it slows down the RSM for all commands it is responsible for. Although aggressive leader election can partially address this [3], it relies on reactive detection mechanisms that cannot catch all slowdowns.

Copilot replication is the first 1-slowdown-tolerant consensus protocol. It takes a proactive approach that uses distinguished replicas, the pilot and copilot. The two pilots do all stages of processing a client's command in parallel: clients send commands

[1]See "Tolerating Slowdowns in Replicated State Machines" at OSDI 2020 [24].

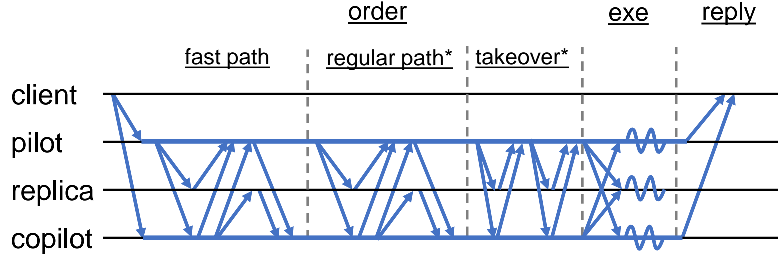


Figure 1: Message diagram for Copilot. Phases that are only sometimes necessary are marked with asterisks (*). The takeover phase (§2.2) only executes when it is necessary to prevent one pilot from waiting too long on the other pilot. Copilot's optimizations (§2.3) keep it on the fast path when both pilots are fast and mostly avoid fast takeovers when one pilot is slow.

to both pilots, and both pilots order, execute, and reply to the client. This ensures all steps happen quickly even if one pilot is slow. Copilot's proactive redundancy protects against a slowdown but also makes it more challenging to preserve consistency and efficiency. We provide an overview of Copilot replication and state its correctness guarantees in §2. We then present pseudocode for the protocol in §3 and prove its correctness in §4.

This technical report does not cover certain details of Copilot's design and optimizations, and does not cover its comparison and evaluation against the Multi-Paxos [16] and EPaxos [23] protocols. These can be found in our conference paper [24].

2 Copilot Replication

Copilot assumes the *asynchronous crash failure model*: a failed process stops executing and stops responding to messages, and there is no bound on the time it takes to deliver a message or the relative speed at which processes execute instructions. Copilot requires $2f+1$ replicas to tolerate at most f failures: it guarantees safety (linearizability) despite any number of failures, and liveness as long as a majority of replicas communicate in a timely manner. *Linearizability* is a consistency model that ensures that client commands are (1) executed in some total order, and (2) this order is consistent with the real-time ordering of client commands, i.e., if command a completes in real-time before command b begins, then a must be ordered before b [12]. Copilot's safety and liveness guarantees are the same as Multi-Paxos and EPaxos. Indeed, Copilot's design is inspired by these two protocols. However, Copilot is the only protocol that provides 1-slowdown-tolerance when any one replica becomes slow.

The core idea behind Copilot is to use two distinguished replicas, the *pilot* (P) and the *copilot* (P'), to redundantly process every client command. This ensures that a client command is never blocked by a single replica, the main problem affecting all existing consensus protocols. For example, in the Multi-Paxos protocol, only the leader receives client commands and runs the ordering protocol, so if it becomes slow, the entire RSM slows down. Figure 1 shows the life of an individual command in Copilot, which begins with a client sending the command to both pilots. By providing two disjoint paths for processing a command at every stage, Copilot prevents any single slow replica from slowing down the RSM.

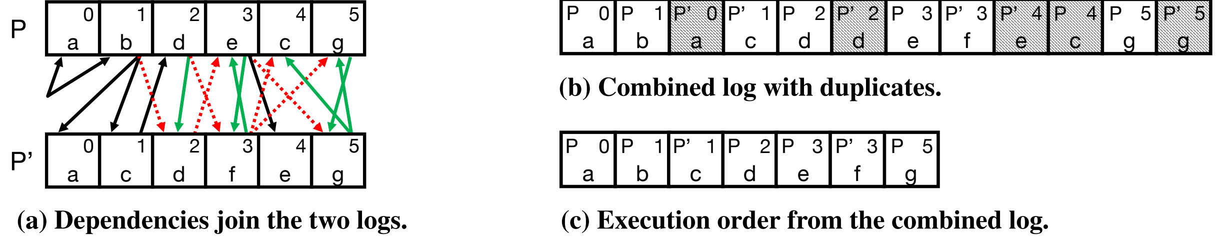
The following subsections provide an abbreviated version of Copilot's design that defines our terminology and includes enough detail to understand the pseudocode and proofs presented in later sections. For a full description of Copilot's design, please refer to sections 3 and 5 of our conference paper [24].

2.1 Main Protocol

We assume there are no failures or slowdowns for now; §2.2 addresses failures and slowdowns.

Ordering. Clients send commands to both the pilot and the copilot. The pilot and copilot assign commands to increasing entries in the *pilot log* and *copilot log* respectively. The two separate logs are ordered together using *dependencies* that indicate the prefix of the other log that should be executed before a given entry. When a pilot receives a new command from a client it assigns it to the next available entry in its log. A pilot also sets the *initial dependency* for that entry to be the highest known assigned entry in the other pilot's log. For example, the pilot may assign command a to entry $P.i$ in its own log with an initial dependency on entry $P'.j$ in the copilot's log.

A pilot sends FastAccept messages to the other replicas to try to persist its assignment of command and initial dependency to an entry in a single round. Replicas accept a FastAccept by replying with a FastAcceptOk message as long as the proposed ordering passes a *compatibility check* that ensures it is compatible with all previously accepted orderings. An entry $P.i$ with initial dependency $P'.j$ is compatible if the replica has not already accepted a later entry $P'.k$ ($k > j$) from the other pilot P' with a dependency earlier than $P.i$, i.e., $P'.k$'s dependency is $< P.i$. In other words, a pair of dependencies are compatible if at



(a) Dependencies join the two logs.

(b) Combined log with duplicates.

(c) Execution order from the combined log.

Figure 2: Dependencies are used to combine the pilot (*P*) and copilot (*P*′) logs (a) into the combined log (b) that is deduplicated and then used for execution (c). (a) Solid black arrows indicate initial dependencies that became final dependencies because an entry was committed on the fast path. Dotted red arrows indicate initial dependencies rejected by the compatibility check because they could lead to different execution orders—e.g., *P*.3 or *P*′.3 could be executed seventh. Solid green arrows indicate final dependencies for entries whose initial dependency was rejected and thus committed on the regular path. Green arrows may contain cycles, which are consistently ordered by the execution protocol to derive a combined log. (b) The combined log has duplicates of most commands, shown in gray. (c) A command is only executed in its first position in the combined log.

least one orders its entry after the other. If the compatibility check fails, the replica sends a FastAcceptReply message to the pilot with its latest entry for the other pilot, $P'.k$, as its *suggested dependency*.

Figure 2a shows examples of compatible and incompatible dependencies. $P'.1$ with dependency $P.1$ and $P.2$ with dependency $P'.1$ are compatible because $P.2$ is ordered after $P'.1$. $P'.3$ with dependency $P.2$ and $P.3$ with dependency $P'.2$ are incompatible because neither is ordered after the other. Incompatible dependencies must be avoided because they could lead to replicas with different subsets of the pilot and copilot logs executing entries in different orders, e.g., one replica executing $P.3$ then $P'.3$ and another executing $P'.3$ then $P.3$. Our safety proof in §4.1 relies crucially on the compatibility check.

If a pilot receives a *fast quorum* of $f + \lfloor \frac{f+1}{2} \rfloor$ FastAcceptOk replies (including from itself), then the entry has committed on the *fast path*. Otherwise, the pilot selects a *final dependency* for the entry based on the suggested dependencies in the responses to the FastAccept round. (All FastAcceptOk messages (including the pilot's) suggest the initial dependency.) The pilot sorts the suggested dependencies in ascending order and then selects the $(f+1)$-th as the final dependency. This dependency is high enough to capture the necessary ordering constraint on this entry: it must use the $(f+1)$-th dependency to ensure quorum intersection with any command that has already been committed and potentially executed by the other pilot, so that this entry is ordered after that entry as required by linearizability. It is no higher to avoid creating more cycles for the other pilot: any dependency beyond the $(f+1)$-th will have its own dependency on this entry because this entry arrived at a majority quorum first. The pilot then persists that final dependency by sending Accept messages to the replicas. When the pilot receives $f+1$ AcceptOks (including from itself) it has committed the entry on the *regular path*.

Once the entry is committed, the pilot sends Commit messages to the other replicas and proceeds to execution (execution happens concurrently with sending the Commit message).

Execution. Replicas execute commands in the order specified by the combination of the pilot and copilot logs and the dependencies between their entries. The combined log contains each client command twice. Execution deduplication ensures that a replica will only execute a command in its first position in the combined order. After executing a command, the pilot and copilot reply to the command's client (the client ignores the second response). Figure 2 shows an example of a combined log and its executed subset.

The total order of commands in the combined log is determined by the partial order of each pilot's log, the dependencies between them, and a priority rule. (1) The partial order of each pilot's log is just the order of its entries, e.g., $P.0 < P.1 < P.2$ in Figure 2a. The dependencies between the logs sometimes create cycles. (2) When there is no cycle, the total order follows the dependency order, e.g., $P.1 < P'.0 < P'.1 < P.2$ in Figure 2a. (3) When the dependencies form a cycle, the total order is determined by the *priority* of the pilots: the pilot's entries are ordered before the copilot's, e.g., $P.4 < P.5 < P'.5$ in Figure 2a.

A replica executes a command once its entry is committed and all preceding entries in the total order have been executed. Specifically, a replica can execute a command in entry $P.i$ with dependency $P'.j$ if (0) $P.i$ is committed, and (1) it has executed $P.(i-1)$, and then one of the following two conditions holds: (2) it has executed $P'.j$, or (3) P is the pilot log and cycles exist between $P.i$ and all P''s log entries $\leq P'.j$ that have not been executed. Rules 1–3 correspond to the rules that define the total order above.

2.2 Fast Takeover

Before executing a command, a pilot needs to wait for the command's dependency to commit in the other pilot's log. Waiting on the other pilot for a long time would not be slowdown tolerant, so Copilot includes a fast takeover mechanism that allows a fast pilot to complete the necessary ordering work of a slow pilot.

All entries in the logs for both pilots have associated ballot numbers, and all messages include ballot numbers as in Paxos's proposal numbers [17]. When a replica is elected as either pilot or copilot, all of the entries in its log are associated with a ballot number b. When a pilot becomes slow, the other pilot can take over specific entries by proposing a higher ballot number for those entries using Paxos's two phases of prepare and accept. Specifically, the fast pilot sends a Prepare message with a higher ballot number b' to all replicas. If a replica determines b' is higher than the set ballot number for an entry, the replica updates the entry's ballot number to b' and replies with a PreparedOk message. The PreparedOk message indicates the progress of the entry at a replica—not-accepted, fast-accepted, accepted, or committed—and also includes the highest ballot number for which it has fast or regular accepted the entry, the command and dependency associated with the entry, and the id of the dependency's proposing pilot.

After sending the Prepare messages, the fast pilot waits for at least $f+1$ PrepareOks (including from itself). If any PrepareOk message indicates an entry is committed, the pilot short-circuits waiting and commits that entry with the same command and dependency. Otherwise, the fast pilot uses the value picking procedure described below to select a command and dependency. It then sends Accept messages for that command and final dependency, waits for $f+1$ AcceptOk replies, and then continues the execution protocol.

Fast takeovers are triggered using a timeout mechanism that is applied to each committed command whose final dependency has not yet committed. Standard techniques are used to avoid dueling proposers, e.g., randomized exponential backoff.

Recovery value picking procedure `choose_value()`. We use *value* to indicate the command and dependency for a log entry. The fast takeover mechanism and view change mechanism (§2.3) use the recovery value picking procedure `choose_value()` to correctly recover a command and dependency for any entry that could have been committed (and thus could have executed). This procedure is the most complex part of the Copilot protocol: the full list of cases are detailed in our pseudocode (§3.2) and are required by our safety proof to ensure all replicas execute all commands in the same combined log order (§4.1).

The procedure starts by examining the set S of PrepareOk replies that include the highest seen ballot number. The first three cases are straightforward:

1. There are one or more replies $r \in S$ with accepted as their progress. Then pick r's command and dependency.
2. There are $< \lfloor \frac{f+1}{2} \rfloor$ replies $r \in S$ with fast-accepted as their progress. Then pick no-op with an empty dependency.
3. There are $\geq f$ replies $r \in S$ with fast-accepted as their progress. Then pick r's command and dependency.

In the first case, the value may have been committed with a lower ballot number in an Accept phase, so the same value must be used. In the second case, the value could not have been committed in either an Accept phase or a FastAccept phase, so it is safe to pick a no-op. In the third case, the value may have been committed with a lower ballot number in a FastAccept phase and it is safe to use the same value. It is safe because the f or more fast-accept replies plus the entry's original proposing pilot form a majority quorum of replicas that passed the compatibility check. This ensures that any incompatible entries from the other pilot's log will be ordered after this entry, making it safe to commit this entry with its initial dependency.

The remaining case is when there are in the range of $[\lfloor \frac{f+1}{2} \rfloor, f)$ replies $r \in S$ with fast-accepted as their progress. In this case, the value may have been committed with a lower ballot number in a FastAccept phase, or it might not have because an incompatible entry in the other pilot's log reached the replicas first. In the first subcase we must commit using the same value, and in the second subcase we must not. To distinguish between these subcases, the recovering replica examines the possible incompatible entries in the other pilot's log. The possible incompatible entries are gathered from the PrepareOk replies of the replicas that did not fast accept the initial dependency of the recovered entry and suggested different dependencies instead.

2.3 Additional Design and Optimizations

The remaining parts of Copilot's design are similar to normal RSM designs and are not discussed in this report. These include: how to trigger fast takeovers, which relies on an appropriately chosen timeout and exponential backoff to avoid dueling proposers; ensuring at-most-once semantics, which relies on caching command ids and outputs; handling non-determinism, which relies on precomputing pseudorandom seeds; and performing garbage collection.

Copilot uses *view changes* to elect a pilot and copilot for the pilot and copilot logs, respectively. The core view change protocol is analogous to Multi-Paxos's leader election [21]. However, during a view change, a newly elected pilot must use the recovery value picking procedure described above to commit all unresolved entries in the log.

Copilot includes two optimizations that improve performance and enhance slowdown tolerance, but are not required for correctness. The first optimization, ping-pong batching, coordinates the pilots so they propose compatible orderings when both

are fast. This is achieved by having each pilot batch their client commands until they receive a FastAccept message from the other pilot, or a ping-pong-wait timeout fires. This causes FastAccepts to ping-pong back and forth between the two pilots, ensuring that the the dependencies they propose are compatible and thus commit on the fast path. The ping-pong timeout is necessary for slowdown tolerance to ensure a fast pilot does not wait indefinitely to propose its batch.

The second optimization, null dependency elimination, allows a fast pilot to avoid waiting on commits from a slow pilot. If a committed entry $P.i$ in the fast pilot's log depends on an entry $P'.j$ that has not yet committed, but the command inside $P'.j$ has already been executed by the fast pilot (recall that each command appears in *both* pilot logs), then this dependency can be ignored because it will not be executed on account of deduplication. We call such a dependency a *null dependency* and allow the fast pilot to simply proceed with executing $P.i$. Thus when one pilot is continually slow, null dependency elimination allows the fast pilot to avoid doing a fast takeover for each uncommitted dependency, because these dependencies will become null dependencies—i.e., they will contain commands that have already been executed by the fast pilot.

2.4 Slowdown Tolerance

Copilot achieves 1-slowdown-tolerance by ensuring a client command is never blocked on a single path. That is, there are always two disjoint paths in the processing of a command, from when it is received by the RSM to when a response is sent to the client, and one of the paths must be fast.

When both pilots are fast, 1-slowdown-tolerance is trivially achieved even if up to f (non-pilot) replicas are slow or failed. This is because the regular path only requires a majority of replicas, allowing both pilots' entries (and their dependencies) to commit and execute. If one of the pilots becomes slow or fails, then the fast pilot can still commit its entries, but some of these entries might depend on uncommitted entries in the slow pilot's log. In this case, the fast pilot does a fast takeover of these entries and commits them. Shortly after a slowdown, the fast pilot stops acquiring dependencies on uncommitted entries—or acquires only null dependencies, as explained above—eliminating the need for any fast takeovers. Thus, the performance of the RSM reduces to that of the faster pilot, satisfying 1-slowdown-tolerance.

Fast takeovers have a superficial resemblance to leader election because both are triggered by one replica timing out while waiting to hear from another replica. However, there is a crucial difference: fast takeovers are triggered when one pilot is waiting to execute a specific client command, putting them on the processing path of every command. In contrast, leader elections are triggered by detection mechanisms—e.g., receiving a heartbeat or a new proposal in time—which means that a slow leader can remain undetected. This is not 1-slowdown-tolerant.

When one pilot is continually slow or failed, Copilot will trigger a view change to elect a new pilot. The two separate logs of the pilots allow this election to occur while the active pilot continues to order and commit commands in its own log. During this time, the active pilot will acquire no new dependencies and can thus commit on the fast path and execute commands without waiting on any entries in the other log.

3 Pseudocode

This section provides pseudocode for the main components of Copilot replication.

3.1 Pilots and Replicas

Each replica maintains two view states, one for the pilot P and one for the copilot P'. Every ordering message pertaining to a pilot's log includes the view ID of the sender's current view for this pilot. A replica only processes the ordering message if its current view for this pilot is `ACTIVE`—i.e., it is not undergoing a view change (§3.3)—and its current view ID for the pilot matches the view ID in the message. Our ordering protocol description assumes that all replicas participating in the ordering of a pilot's log are in the same view and their views are `ACTIVE`.

Figure 3 shows a pilot's main functions for ordering commands. The pilot takes a batch of commands and assigns them to the next available entry ($P.i$) with a dependency on the latest entry in the copilot's log ($P'.j$) (line 4). The entry is initially fast-accepted at the pilot itself (lines 6–7) and a FastAccept message is sent to all other replicas. If the pilot receives at least $f+\lfloor\frac{f+1}{2}\rfloor$ FastAcceptOk replies, then the entry can be committed on the fast path (line 18). Otherwise, if the compatibility check fails at too many replicas—i.e., they respond with FastAcceptReply—or if insufficient replies are received and a timeout expires, then the pilot proceeds to the Accept phase on the regular path (line 20). In this phase, all the suggested dependencies collected from FastAcceptOk/FastAcceptReply messages are sorted and the $(f+1)$-th dependency is selected as the final dependency (lines 25–27). An Accept message for this entry with the final dependency is is sent to all replicas (line 30). After at least f AcceptOk replies are received, the entry is accepted. After an entry is accepted on either the fast or regular path, the pilot sends a Commit message to all replicas (lines 35–38). No responses are required during the commit phase; if an entry remains uncommitted at any replica due to a failure (e.g., a dropped message or a failed pilot), then the fast recovery and/or view change protocols will be initiated to resolve these entries (§3.3).

```
1  Pilot
2  func propose(cmds, P, i) {
3    // Acquire dependency on latest entry in other pilot's log
4    j := latestEntry[P']; dep := P'.j
5    // Entry is fast-accepted at this replica
6    Logs[P][i].status = FAST_ACCEPTED
7    FastAcceptOks := 1
8    FADeps[P][i] = dep // used for compatibility check
9    FAReplyDeps := [dep]
10   send FastAccept(P, i, cmds, dep) to all replicas
11   wait for at least f other FastAcceptOk/FastAcceptReply replies; beyond f replies, wait up to a timeout
12   foreach FastAcceptOk reply {
13     FastAcceptOks++
14     FAReplyDeps.add(dep)
15   }
16   foreach FastAcceptReply reply { FAReplyDeps.add(reply.dep) }
17   if FastAcceptOks ≥ f+⌊(f+1)/2⌋ {
18     commit(P, i)
19   } else {
20     accept(P, i)
21   }
22 }
23
24 func accept(P, i) {
25   sort(FAReplyDeps)
26   // Select (f+1)-th suggested dependency as final dependency
27   Logs[P][i].dep = FAReplyDeps[f]
28   // Entry is accepted at this replica
29   Logs[P][i].status = ACCEPTED
30   send Accept(P, i, Logs[P][i].cmds, Logs[P][i].dep) to all replicas
31   wait for at least f other AcceptOk replies; beyond f replies, wait up to a timeout
32   commit(P, i)
33 }
34
35 func commit(P, i) {
36   Logs[P][i].status = COMMITTED
37   send Commit(P, i, Logs[P][i].cmds, Logs[P][i].dep) to all replicas
38 }
```

Figure 3: Pseudocode for Copilot's ordering protocol at a pilot.

Figure 4 shows a replica's main functions for ordering commands (this includes the replica code executed by a pilot, since a pilot acts as a replica for entries being ordered by its copilot). Each replica maintains a log of the entries seen for each pilot (in `Logs`). Upon receiving a FastAccept message for an entry *P.i* from pilot *P*, a replica records the proposed command in the *i*-th entry of *P*'s log. It then checks if the initial dependency of *P.i* is compatible with all previously fast-accepted orderings by calling `checkCompatibility()`. `checkCompatibility()` follows the rules in §2.1 to determine if a dependency is compatible: if the check passes, the replica sends a FastAcceptOk message to *P*; otherwise, it sends a FastAcceptReply message to *P* with the dependency suggested by `checkCompatibility()`. The replica also records the suggested dependency of all entries (in `FADeps`), which `checkCompatibility()` consults to ensure that future entries are also compatible. Upon receiving an Accept message for an entry *P.i*, the replica updates the status and final dependency of *P.i*, and sends an AcceptOk message to the sender. Similarly, upon receiving a Commit message for *P.i*, a replica updates the status of *P.i* and commits the entry. Committed entries are eligible for execution, as discussed next.

Figure 5 shows the execution protocol for a replica (and a pilot acting as a replica). The execution protocol combines the pilot and copilot logs by iterating over each log and constructing a total order of the commands. This total order is based on the partial order of each log, the dependencies between log entries, and a priority rule. The partial order is enforced by executing the entries within the same log in increasing order: entry $P.(e+1)$ can only be executed after entry *P.e* has been executed. To execute an entry *P.e*, we first check if it has been committed (line 9). Then, we execute it if any of the following cases apply. (1) *P.e* is a no-op, (2) *P.e* has no dependency, or (3) its dependency, $P'.e'$, has already been executed. These cases are handled by line 20. (4) A cycle exists between *P.e* and the unexecuted entries of the other pilot's log, but *P* has higher priority so its entries are executed first (line 27). (5) $P'.e'$, the dependency of *P.e*, and all unexecuted entries preceding it in the other pilot's log are all nullable, meaning that the contained commands have already been executed at this replica. (The `nullDepSafe` flag of an entry

```
1 Replica
2 func checkCompatibility(P, i, dep) {
3   P' = dep.pid
4   for e' := latestEntry[P']; e' > dep.e; e'-- {
5     if FADeps[P'][e'] < i {
6       return (false, P'.e')
7     }
8   }
9   return (true, dep)
10 }
11
12 on Receiving FastAccept(P, i, cmds, dep)
13   isCompatible, dep'  = checkCompatibility(P, i, dep)
14   Logs[P][i].cmds = cmds
15   Logs[P][i].dep = dep'
16   FADeps[P][i] = dep'
17   if isCompatible {
18     Logs[P][i].status = FAST_ACCEPTED
19     send FastAcceptOk() to P
20   } else {
21     Logs[P][i].status = NOT_ACCEPTED
22     send FastAcceptReply(dep') to P
23   }
24
25 on Receiving Accept(P, i, cmds, dep)
26   Logs[P][i].cmds = cmds
27   Logs[P][i].dep = dep
28   Logs[P][i].status = ACCEPTED
29   send AcceptOk() to P
30
31 on Receiving Commit(P, i, cmds, dep)
32   Logs[P][i].cmds = cmds
33   Logs[P][i].dep = dep
34   Logs[P][i].status = COMMITTED
```

Figure 4: Pseudocode for handling the ordering protocol's messages at a replica.

indicates whether it is safe to consider nullifying the entry's dependency, and is updated while ordering and committing the entry. Further details can be found in Copilot's code repository [8].) If any of these cases apply, entry *P.e* is executed by calling `executeCmds()`, which executes the commands in *P.e* and updates the state of the RSM. The commands are only executed where they first appear in the combined total order (lines 59–61)—recall that each command appears twice in the logs because they are processed by bot the pilot and the copilot. After executing a command, if this is the pilot or copilot, a reply is sent to the client who issued the command. If none of the above cases apply, then this implies that some entries from the other pilot's log need to be executed before *P.e* can be executed, and so execution switches to the other pilot's log (line 44).

If the entries that potentially precede *P.e* in the total order are taking too long to commit—these include the dependency $P'.e'$ and all uncommitted entries preceding it in the log of P'—the pilot P invokes the fast takeover mechanism to complete the ordering work of these uncommitted entries (lines 37–43). This ensures that Copilot remains 1-slowdown-tolerant, since otherwise the execution protocol would be blocked waiting on these dependencies to commit.

3.2 Fast Takeover

Figure 6 shows the pseudocode for the fast takeover procedure, which is continued in Figures 7 and 8. A fast takeover is invoked when a pilot needs to complete the ordering work of a slow copilot, or when a new pilot needs to complete the partial entries of a failed pilot during pilot election. A pilot takes over the ordering work of an entry *P.i* using Paxos's two phases of prepare and accept, summarized in lines 2–14. It creates a new ballot number that is higher than any ballot number that has been prepared for *P.i*, and sends Prepare messages with this higher ballot number to all replicas. The pilot waits for at least f replies from other replicas. We are guaranteed to get f replies by our liveness assumptions; beyond f replies, we wait up to a timeout. If it gathers at least $f+1$ PrepareOk messages, it calls the `choose_value()` procedure, which picks the correct value of *P.i* based on this set of PrepareOk messages (described further below). Otherwise, the pilot retries the prepare phase with a higher ballot number.

Upon receiving a Prepare message, if the proposed ballot number is higher than the previously set ballot number for *P.i*, a replica replies with a PrepareOk message and updates the prepared ballot number for *P.i* (lines 19–20). The PrepareOk message

```
1 Pilot/Replica
2 execMap := map[int → bool] // mapping command ID to true/false to keep track executed commands
3 maxDep := map[{P,P'} → int] // max dependency at each pilot's log; used for checking if a cycle exists
4 currPilot := P
5 while !stop { // execution thread loops until RSM stops
6   for (e := executedUpTo[currPilot] + 1; e <= latestEntry[currPilot]; e++) {
7     entry = Logs[currPilot][e]
8     // Case 0: Current entry is not committed
9     if entry.status != COMMITTED {
10      currPilot = (currPilot == P) ? P' : P; break  // switch to the other pilot's log
11    }
12
13    // update max dependency among committed entries (up to e) of currPilot's log
14    maxDep[currPilot] = max(maxDep[currPilot], entry.dep.e)
15    depPilot = entry.dep.pid; e' = entry.dep.e
16    depEntry = Logs[depPilot][e']
17
18    // Case 1: Current entry is no-op, or has no dependency, or dependency has been executed
19    if entry.cmds == no-op || e' == ∅ || depEntry.status == EXECUTED {
20      executeCmds(currPilot, e); executedUpTo[currPilot] = e; continue
21    }
22
23    // Case 2: A cycle exists between entry and depEntry and currPilot has higher priority
24    // (if it has lower priority, the cycle will be handled while processing the other pilot's log)
25    hasCycle := maxDep[depPilot] ≥ e
26    if hasCycle && hasHighPriority(currPilot) {
27      executeCmds(currPilot, e); executedUpTo[currPilot] = e; continue
28    }
29
30    // Case 3: Null dependency elimination succeeds
31    if entry.nullDepSafe && views[depPilot].view.vid ≥ entry.depViewId
32        && checkDepsNullable(depPilot, e') {
33      executeCmds(currPilot, e); executedUpTo[currPilot] = e; continue
34    }
35
36    // Case 4: A pilot invokes fast takeover if the takeover timeout fires
37    if this.isPilot && time.Since(entry.commitTime) ≥ TAKEOVER_TIMEOUT {
38      for i := committedUpTo[depPilot] + 1; i ≤ e'; i++ {
39        if Logs[depPilot][i].status != COMMITTED {
40          takeover(depPilot, i)
41        }
42      }
43    }
44    currPilot = (currPilot == P) ? P' : P; break  // switch to the other pilot's log
45  }
46 }
47
48 func checkDepsNullable(P, e) {
49   for i := executedUpTo[P] + 1; i ≤ e; i++ {
50     if Logs[P][i] == nil || !execMap[Logs[P][i].cmds.id] {
51       return false
52     }
53   }
54   return true
55 }
56
57 func executeCmds(P, e) {
58   foreach cmd in Logs[P][e].cmds {
59     if !execMap[cmd.id] {
60       execute(cmd) // execute the command and update the RSM's state
61       execMap[cmd.id] = true
62       if this.isPilot { send reply to client }
63     }
64   }
65 }
```

Figure 5: Pseudocode for Copilot's execution protocol.

```
1 Pilot
2 func takeover(P, i) {
3   // get higher ballot number and prepare P.i
4   Logs[P][i].ballot = make_higher_ballot(Logs[P][i].ballot)
5   Logs[P][i].status = PREPARED
6   send Prepare(P, i, Logs[P][i].ballot) to all replicas
7   wait for at least f other replies; beyond f replies, wait up to a timeout
8   if ≥ f+1 replies (including from itself) are PrepareOk {
9     Q := set of all PrepareOk replies
10    choose_value(Q)
11  } else {
12    # retry prepare phase with a higher ballot number
13  }
14 }
15
16 Replica
17 on Receiving Prepare(P, i, ballot)
18   inst := Logs[P][i]
19   if ballot > inst.ballot {
20     inst.ballot = ballot
21     inst.status = PREPARED
22     send PrepareOk(inst.status, inst.cmd, inst.dep, inst.accept_ballot)
23   } else {
24     send PrepareReject(inst.ballot)
25   }
```

Figure 6: Pseudocode for Copilot's fast takeover mechanism. The logic for picking the values of uncommitted entries is performed by the `choose_value()` procedure, detailed in Figures 7 and 8.

indicates the progress of *P.i* as seen by the replica—e.g., as a result of receiving a FastAccept or Accept message—which is used by the pilot in the `choose_value()` procedure to determine the correct value of *P.i*. If the proposed ballot number is not higher than the set ballot number for *P.i*, the replica replies with a PrepareReject message containing the highest ballot number it has set (lines 23–24).

`choose_value()` procedure: Figure 7 shows the pseudocode for the `choose_value()` procedure, which determines the value (commands and dependency) for an uncommitted entry *P.i* based on the set of PrepareOk replies. It first calls an auxiliary sub-procedure called `choose_value_common()` that helps resolve *P.i* in simple cases when there are no concurrent uncommitted entries in the other pilot's log that need to be resolved in order to resolve *P.i*. If `choose_value_common()` can decide the value of *P.i*, the `choose_value()` procedure returns. Otherwise, `choose_value_common()` returns the set of entries from the other pilot's log that are potentially concurrent with *P.i* and whose committed values are unknown. These tricky cases may arise, for example, when a pilot fails after it commits on the fast path but before it sends out Commit messages, and a sufficient number of replicas also fail concurrently. In these situations, `choose_value()` may need to resolve the value of each concurrent entry in order to decide the value of *P.i*. It does so by simultaneously preparing *P.i* and each concurrent entry $P'.k$, and then resolving *P.i* and $P'.k$ using the same quorum of replicas $Q_{i,k}$ that is returned by the `simultaneous_prepare()` (lines 7–32). If the value of *P.i* can be resolved with the new quorum $Q_{i,k}$, `choose_value()` returns (lines 9–10). If each of the concurrent entries either commits with a no-op or has a dependency $\geq$ *P.i*, *P.i* can be committed with the commands and dependency initially proposed by the failed pilot (lines 13–14, 18, and 35). We use `accept_and_commit()` as a shorthand for running the Accept phase followed by the Commit phase. If at least one entry commits with a dependency $<$ *P.i*, this implies that *P.i* could not have committed on the fast path, and thus `choose_value()` commits *P.i* with a no-op (line 15). If *P.i* and a concurrent entry $P'.k$ are potentially conflicting (i.e., $P'.k$'s initial dependency is $<$ *P.i*), and they cannot be conclusively resolved by `choose_value_common()`, `choose_value()` determines their values based on the replies from the quorum $Q_{i,k}$, as follows. (Note that $Q_{i,k}$ does not contain either of the original pilots who proposed *P.i* and $P'.k$.) For an entry to have potentially committed on the fast path, it should have $> \lfloor \frac{f+1}{2} \rfloor$ replies with a `FAST_ACCEPTED` status. Hence, if *P.i* and $P'.k$ both have fewer than $\lfloor \frac{f+1}{2} \rfloor$ replies with a `FAST_ACCEPTED` status, `choose_value()` can safely commit a no-op for *P.i* and $P'.k$. If *P.i* has $> \lfloor \frac{f+1}{2} \rfloor$ replies with a `FAST_ACCEPTED` status, `choose_value()` first commits a no-op for $P'.k$ (lines 22–25). If $P'.k$ has $> \lfloor \frac{f+1}{2} \rfloor$ replies with a `FAST_ACCEPTED` status, `choose_value()` commits a no-op for *P.i* and returns (lines 26–29).

`choose_value_common()` sub-procedure: Figure 8 shows the pseudocode for the `choose_value_common()` sub-procedure, which handles the simple cases for resolving *P.i* based on the quorum *Q* of PrepareOk replies. If the sub-procedure can directly

```
1  Pilot
2  func choose_value(P, i,  Q) {
3    Pi_resolved, Pi_initial_cmds, Pi_initial_dep, P'_uncommitted_entries := choose_value_common(P, i, Q)
4    if Pi_resolved { return }
5    foreach k in P'_uncommitted_entries {
6      // P.i and P'.k are both unresolved, so prepare both
7      Q_{i,k} = simultaneous_prepare(P, i, P', k)
8      // Check again if P.i can be resolved with the new quorum Q_{i,k}[P]
9      Pi_resolved, _, _, _ = choose_value_common(P, i, Q_{i,k}[P])
10     if Pi_resolved { return }
11
12     P'k_resolved, _, P'k_initial_dep, _ := choose_value_common(P', k, Q_{i,k}[P'])
13     if P'k_resolved {
14       if Logs[P'][k].cmds == no-op || Logs[P'][k].dep ≥ P.i { continue }
15       accept_and_commit(P, i, cmds=no-op, dep=∅)
16       return
17     }
18     if P'k_initial_dep ≥ P.i { continue } // P.i and P'.k do not conflict
19
20     // The new quorum Q_{i,k} does not contain either of the original pilots
21     // P.i and P'.k conflict and both have < f and ≥ ⌊(f+1)/2⌋ fast accepts
22     if P.i has > ⌊(f+1)/2⌋ FAST_ACCEPTED in Q_{i,k} {
23       accept_and_commit(P', k, cmds=no-op, dep=∅)
24       continue
25     }
26     if P'.k has > ⌊(f+1)/2⌋ FAST_ACCEPTED in Q_{i,k} {
27       accept_and_commit(P, i, cmds=no-op, dep=∅)
28       return
29     }
30     accept_and_commit(P, i, cmds=no-op, dep=∅)
31     accept_and_commit(P', k, cmds=no-op, dep=∅)
32     return
33   } // end of for loop
34   // each concurrent P'.k is either no-op or has dep ≥ P.i; safe to commit P.i with its initial value
35   accept_and_commit(P, i, cmds=Pi_initial_cmds, dep=Pi_initial_dep)
36 }
37
38 func simultaneous_prepare(P, i, P', j) {
39   // get higher ballot numbers and prepare both P.i and P'.j
40   Logs[P][i].ballot = make_higher_ballot(Logs[P][i].ballot)
41   Logs[P'][j].ballot = make_higher_ballot(Logs[P'][j].ballot)
42   Logs[P][i].status = PREPARED
43   Logs[P'][j].status = PREPARED
44   send SimultaneousPrepare(P, i, Logs[P][i].ballot, P', j, Logs[P'][j].ballot) to all replicas
45   wait for at least f other replies; beyond f replies, wait up to a timeout
46   Q := map[{P, P'} → []]
47   if ≥ f+1 replies (including from itself) are SimultaneousPrepareOk {
48     foreach SimultaneousPrepareOk reply {
49       Q[P].add(reply[P]); Q[P'].add(reply[P'])
50     }
51   } else { # retry prepare phase with higher ballot numbers }
52   return Q
53 }
54
55 Replica
56 on Receiving SimultaneousPrepare(P, i, P_ballot, P', j, P'_ballot)
57   Pinst := Logs[P][i]
58   P'inst := Logs[P'][j]
59   if P_ballot > Pinst.ballot && P'_ballot > P'inst.ballot {
60     Pinst.ballot = P_ballot
61     P'inst.ballot = P'_ballot
62     Pinst.status = PREPARED
63     P'inst.status = PREPARED
64     send SimultaneousPrepareOk(map[P → (Pinst.status, Pinst.cmds, Pinst.dep, Pinst.accept_ballot),
65                                   P' → (P'inst.status, P'inst.cmds, P'inst.dep, P'inst.accept_ballot)])
66
67   } else { send SimultaneousPrepareReject(Pinst.ballot, P'inst.ballot) }
```

Figure 7: Pseudocode for `choose_value()` procedure.

```
1  func choose_value_common(P, i, Q) {
2    # let Pi_proposer be the (pilot) replica that proposed in P.i
3    # let S be the set of replies r with r ∈ Q and r.status != NONE
4    // initial commands and dependency proposed in P.i
5    Pi_initial_cmds := nil; Pi_initial_dep := nil
6    if ∃ a reply r with r.status == FAST_ACCEPTED {
7      Pi_initial_cmds = r.cmds; Pi_initial_dep = r.dep
8    }
9    if ∃ a reply r with r.status == COMMITTED || r.status == EXECUTED { // Case 1
10     Logs[P][i].cmds = r.cmds; Logs[P][i].dep = r.dep; commit(P, i)
11   } else if ∃ a reply r with r.status == ACCEPTED { // Case 2
12     # pick r with the highest accepted ballot r.accept_ballot
13     accept_and_commit(P, i, cmds=r.cmds, dep=r.dep)
14   } else if |reply r with r.status == FAST_ACCEPTED| ≥ f+1 ||
15     (|reply r with r.status == FAST_ACCEPTED| == f && Pi_proposer's reply ∉ Q) { // Case 3
16     accept_and_commit(P, i, cmds=Pi_initial_cmds, dep=Pi_initial_dep)
17   } else if Pi_proposer's reply ∈ Q ||
18     |reply r with r.status == FAST_ACCEPTED| < ⌊(f+1)/2⌋ { // Case 4
19     accept_and_commit(P, i, cmds=no-op, dep=∅)
20   } else { // Case 5: ⌊(f+1)/2⌋ ≤ |reply r with r.status == FAST_ACCEPTED| < f && Pi_proposer's reply ∉ Q
21     if |S| < f+1 {
22       send FastAccept(Pi_initial_cmds, P, i, Pi_initial_dep) to the replicas in Q\S; wait for at least
23       f+1-|S| replies; add FAST_ACCEPTED/NOT_ACCEPTED replies to S; recheck cases 1&3; retry if |S|<f+1
24     }
25     max_suggested_dep := max{r.dep.e | reply r ∈ S and r.status == NOT_ACCEPTED}
26     P'_uncommitted_entries := [] // concurrent entries from the other pilot that are uncommitted
27     // examine each concurrent entry to determine if P.i could commit on fast path
28     for e' := Pi_initial_dep.e + 1; e' ≤ max_suggested_dep; e'++ {
29       if Logs[P'][e'].status == COMMITTED || Logs[P'][e'].status == EXECUTED {
30         if Logs[P'][e'].cmds == no-op || Logs[P'][e'].dep ≥ P.i { continue }
31         // Entry committed with a dependency < P.i; implies P.i could not commit on fast path
32         accept_and_commit(P, i, cmds=no-op, dep=∅)
33         return true, nil, nil, nil
34       } else { // found an uncommitted entry
35         P'_uncommitted_entries.add(e')
36       }
37     } // end of for loop
38     if len(P'_uncommitted_entries) > 0 { // need to resolve these entries to resolve P.i
39       return false, Pi_initial_cmds, Pi_initial_dep, P'_uncommitted_entries
40     }
41     // concurrent entries either are no-op or have dep ≥ P.i; safe to commit P.i with its initial value
42     accept_and_commit(P, i, cmds=Pi_initial_cmds, dep=Pi_initial_dep)
43   }
44   return true, nil, nil, nil
45 }
```

Figure 8: Pseudocode for `choose_value_common()` sub-procedure.

resolve *P.i* based on Q, it returns true. Otherwise, it returns false, the commands and the dependency initially proposed in *P.i*, and `P'_uncommitted_entries`, the set of entries from the other pilot that are potentially concurrent with *P.i* and have not been committed.

The sub-procedure handles five cases. *P.i* can always be definitively resolved in cases 1–4, so the sub-procedure returns true in these cases. Cases 1 and 2 are similar to how the canonical Paxos protocol chooses a value for its Accept phase. Case 3 implies that at least a majority of replicas have fast accepted *P.i*'s value, which means the proposing pilot of this value fast accepted its commands and initial dependency. Thus, `choose_value_common()` tries to get the same value committed.

In case 4, it is safe to commit a no-op for *P.i* if either (1) fewer than $\lfloor \frac{f+1}{2} \rfloor$ replicas have fast accepted the initial value proposed in *P.i* or (2) the proposing pilot is in Q. For (1), since at most f replicas outside of Q could have fast accepted *P.i*, fewer than $f + \lfloor \frac{f+1}{2} \rfloor$ could have fast accepted *P.i*. Hence, the proposing pilot of *P.i* could not have committed *P.i*. For (2), since the proposing pilot of *P.i* has not committed *P.i* (otherwise, *P.i* would have been resolved in case 1) and no longer has ownership of *P.i*, it is safe to commit a no-op for *P.i*.

In the last case (case 5), $\geq \lfloor \frac{f+1}{2} \rfloor$ and $< f$ replicas have fast accepted *P.i* and the proposing pilot of *P.i* is not in Q. If $|S| \geq (f+1)$, need to examine the entries from the other pilot that may have ordering conflicts with *P.i* (lines 25–42). These entries are the dependencies suggested by the replicas in Q which fast accepted *P.i* with a different dependency. If any $P'.k$ commits with a dependency $< P.i$, this implies that *P.i* could not have committed on the fast path and we can commit a no-op for *P.i*. If all $P'.k$ commit with a no-op or have a dependency $\geq P.i$, we can commit *P.i* with its original dependency since all concurrent entries from the other pilot have compatible ordering with *P.i*. If any $P'.k$ has not committed, it is added to the set of uncommitted entries `P'_uncommitted_entries`. In this case, `choose_value_common()` returns false, the commands and the dependency initially proposed in *P.i*, and `P'_uncommitted_entries`.

In case 5, if $|S| < (f+1)$, the recovering replica first tries to get more replicas to fast accept the initial value proposed by the failed pilot (lines 21–24). It does so by sending a FastAccept for the initial value to the replicas in $Q \backslash S$, which have not received it. It waits for at least $(f+1-|S|)$ replies; beyond $(f+1-|S|)$ replies, it waits up to a timeout. If one of the replicas rejects the FastAccept—e.g., because the entry has been prepared with a higher ballot number by another recovering replica—but indicates that the entry has been committed, the recovering replica commits the same committed value and the recovery procedure is completed. If at least $(f+1-|S|)$ replicas reply with a FastAcceptOk (i.e., they accept the initial dependency and set the entry's status to `FAST_ACCEPTED`) or a FastAcceptReply (i.e., they suggest a different dependency and set the entry's status to `NOT_ACCEPTED`), then the recovering replica adds these replies to S. It can now form a majority quorum of replicas that have fast accepted/not accepted the initial value proposed in *P.i*. If at least a majority of replicas have fast accepted the initial value, this is the same as case 3 and the recovering replica tries to get the same value accepted and committed by a majority of replicas. Otherwise, the recovering replica needs to examine the potentially concurrent entries of the other pilot's log to determine *P.i*, as we have described above. (If the recovering replica cannot gather enough FastAcceptOk/FastAcceptReply replies to form a majority, it applies a standard randomized exponential backoff before retrying the takeover.)

3.3 View Change (Pilot Election)

Figure 9 shows the pseudocode for the pilot election protocol. Copilot elects a new pilot by initiating a view change on the pilot's log. This view change protocol is based on the one used by canonical Paxos [21], and runs in three phases: it first proposes a new view ID, then it proposes a new view, and finally it commits the new view.

When a replica suspects that a pilot may have failed, it initiates a view change to elect a new pilot. It does this by assuming the role of the view manager and proposing a view ID that is higher than the highest view ID it has seen for this pilot (lines 4–5). Note that the view manager is not necessarily the pilot in the new view: it may or may not become the pilot once the new view is formed. The view manager also keeps track the latest entry in the pilot's log that is known by at least a majority of replicas. This identifies the next entry in the log where the new pilot should start proposing its commands. In particular, the view change protocol ensures that the new pilot does not propose new commands in an entry that has potentially been committed by the failed pilot (discussed further below). The view manager sends a ViewChange message containing the new view ID and the current view state of the pilot to all replicas. It waits for at least $f+1$ ViewChangeOk replies (including from itself) before forming the new view; if it does not gather enough ViewChangeOk replies, it retries with a higher view ID (line 9).

When a replica receives a ViewChange message, it compares the new view ID and current view ID of the proposer (the view manager) with its local view state for the pilot. If the manager's current view ID is less than replica's current view ID—implying that the manager's current view is stale, since a new view has been successfully formed that comes after the view the manager wants to change—or if the manager's new view ID is less than the highest view ID the replica has seen, the replica replies with a ViewChangeReject message containing its current view ID and the highest view ID it has seen (lines 45–47). Otherwise, the replica accepts the ViewChange message by becoming a follower, updating its current view to match the manager's current view (if needed), and replying with a ViewChangeOk message (lines 48–51). Since there may be concurrent view change

```
1 View Manager (any replica)
2 vs := views[P] // current view state for pilot P at this replica
3 func startViewChange(P) {
4   vs.mode = MANAGER
5   vs.proposed_vid = make_higher_viewid(vs.proposed_vid)
6   send ViewChange(vs.view, vs.proposed_vid) to all replicas
7   wait for at least f other replies; beyond f replies, wait up to a timeout
8   if ≥ f+1 replies (including from itself) are ViewChangeOk { acceptView() }
9   else { # retry startViewChange() with a higher proposed_vid }
10 }
11
12 func acceptView() {
13   if ∃ a ViewChangeOk reply r with r.accepted_view != nil {
14     // some replicas have already accepted a view (proposed by a concurrent view manager)
15     // select r with the highest accepted view ID; use r's accepted view as the new view
16     vs.accepted_view = r.accepted_view
17     vs.accepted_latest_entry = r.accepted_latest_entry
18   } else {
19     // no existing accepted views; create a new view with this replica as the pilot
20     vs.accepted_view = form_view(vs.proposed_vid, my_replica_id)
21     vs.accepted_latest_entry = latestEntry[P]
22     foreach reply r in ViewChangeOk replies {
23       vs.accepted_latest_entry = max(vs.accepted_latest_entry, r.latest_entry)
24     }
25   }
26   send AcceptView(vs.accepted_view, vs.accepted_latest_entry) to all replicas
27   wait for at least f other replies; beyond f replies, wait up to a timeout
28   if ≥ f+1 replies (including from itself) are AcceptViewOk { startView() }
29   else { # retry startViewChange() with a higher proposed_vid }
30 }
31
32 func startView() {
33   vs.view = vs.accepted_view
34   latestEntry[P] = vs.accepted_latest_entry
35   vs.mode = ACTIVE
36   vs.accepted_view = nil
37   send StartView(vs.view, latestEntry[P]) to all replicas
38   // fill holes in pilot P's log
39   for i := latestCommit[P] + 1; i <= latestEntry[P]; i++ { takeover(P, i) }
40 }
41
42 Replica
43 vs := views[P] // current view state for pilot P at this replica
44 on Receiving ViewChange(manager_curr_view, manager_proposed_vid) {
45   if manager_proposed_vid ≤ vs.proposed_vid || manager_curr_view.vid < vs.view.vid {
46     send ViewChangeReject(vs.view, vs.proposed_vid); return
47   }
48   if manager_curr_view.vid > vs.view.vid { # update local view of pilot P to match manager_curr_view }
49   vs.mode = FOLLOWER
50   vs.proposed_vid = manager_proposed_vid
51   send ViewChangeOk(latestCommit[P], latestEntry[P], vs.accepted_view, vs.accepted_latest_entry)
52 }
53
54 on Receiving AcceptView(view, latest_entry) {
55   if vs.proposed_vid == view.vid {
56     vs.accepted_view = view
57     vs.accepted_latest_entry = latest_entry
58     send AcceptViewOk()
59   } else { send AcceptReject(vs.proposed_vid) }
60 }
```

Figure 9: Pseudocode for view change protocol.

proposals, it is possible that the replica has already accepted a new view from a concurrent view manager at a lower view ID (which is why it is accepting the current manager's higher view ID). If this is the case, the replica includes the accepted view and the latest entry in the accepted view in its ViewChangeOk reply. It also includes the latest entry number from the pilot's log that it is aware of. This information (shown in line 51) is used by the view manager when forming the new view.

Once a view manager receives at least $f+1$ ViewChangeOk replies (including from itself), it starts forming the new view. If one or more replica have already accepted a new view (from a concurrent view manager, as indicated in their ViewChangeOk replies), the manager selects the accepted view with the highest view ID and uses this as the new view configuration (line 16). In particular, it uses the `accepted_latest_entry` value from this accepted view (line 17). If no replicas have accepted any views, the manager selects itself as the new pilot of the new view and sets `accepted_latest_entry` to the latest entry number among all of the ViewChangeOk replies. The manager then sends an AcceptView message containing the new view configuration to all replicas and waits for at least $f+1$ AcceptViewOk replies (including from itself); if it does not gather enough AcceptViewOk replies, it retries `startViewChange()` with a higher view ID.

Once the view manager receives at least $f+1$ AcceptViewOk replies, it commits the new view and starts operating in the new view. It does this by updating the view of the pilot to the new view configuration, setting the latest entry for proposing new commands, and setting the new default ballot number for future entries. It then sets the view mode to `ACTIVE`, indicating that it can start sending and receiving ordering protocol messages that operate on this pilot's log. (Note that when the view is in any mode other than `ACTIVE`, the replica will ignore or reject any ordering protocol messages pertaining to this pilot's log.) The manager sends a StartView message to all replicas indicating that they too can commit and start the new view; upon receiving this message, each replica performs the same steps above to update the pilot's view. A replica becomes the new pilot if its ID matches the replica ID of the new pilot specified in the new view configuration, and steps down from being the pilot otherwise. The new pilot will resolve and finalize any uncommitted entries in the log by invoking the fast takeover procedure (§3.2).

3.4 Other code

The pseudocode above covers all components of Copilot's implementation. However, some details of our implementation are not included, such as null dependency elimination, ping-pong batching, and metadata maintenance. These details, and our complete implementation, can be found in Copilot's code repository [8].

4 Proof of Correctness

We prove that Copilot replication is both safe (§4.1), i.e., it provides linearizability, and live (§4.2), i.e., all client commands eventually complete.

4.1 Safety

To prove linearizable semantics, we must show that client commands are (1) executed in some total order, and (2) this order is consistent with the real-time ordering of client commands, i.e., if command *a* completes in real-time before command *b* begins, then *a* must be ordered before *b* [12]. We begin by proving the real-time ordering requirement (Lemma 1) and then prove the total order requirement (Lemma 9).

Lemma 1 *If command β is proposed by a client after command α is committed, β will be executed after α.*

Proof. Assume command α from client c_1 has been committed by some replica at time t_1, and command β is proposed by client c_2 at time $t_2 > t_1$. We will show that β will be executed after α by any replica.

Assume that α is included in log entry $P.i$ of pilot P and log entry $P'.m$ of pilot P' (since a client sends a command to both pilots). Since α was committed, either $P.i$ or $P'.m$ must have been committed (this is true even if only one of the pilots received the command). Without loss of generality, assume that entry $P.i$ was committed by P at time t_0, and $P'.m$ was committed or will be committed by P' at time $t_3 > t_0$.

We first show that β will be proposed in a log entry that is later than $P.i$ at pilot P, and in a log entry later than $P'.m$ at pilot P'. Since a replica can commit α only after it receives a Commit message from either $P.i$ or $P'.m$, the commit time $t_1 > min\{t_0, t_3\} = t_0$, which implies that $t_2 > t_1 > t_0$. Since client c_2 proposes β at time $t_2 > t_0$, β will arrive at pilot P at some time after t_0, and hence will be included in a later log entry $P.j$: i.e., $\beta \in Logs[P.j].cmds$ with $P.j > P.i$. Similarly, β is included in a later log entry $P'.n$ by pilot P': i.e., $\beta \in Logs[P'.n].cmds$ with $P'.n > P'.m$.

We next show that $Logs[P'.n].dep \geq P.i$. Since $P'.n$ is constructed and proposed by P' after t_0, $P'.n$ will be committed with a dependency $\geq P.i$. To see why this is the case, observe that entry $P.i$ was committed at time t_0, which implies that $P.i$ was accepted by at least a majority of replicas at some time before t_0. Since $P'.n$ will be accepted by a majority of replicas after t_0, the two quorums will intersect and force $P'.n$ to acquire a dependency on some log entry of P that is $\geq P.i$. Hence, we have $Logs[P'.n].dep \geq P.i$.

We now show that no cycle can occur between $P.i$ and $P'.n$ by showing that all entries $P.e \leq P.i$ commit with a dependency $< P'.n$. Since $P.i$ was committed at time t_0, an entry $P.e \leq P.i$ must have completed the Fast Accept phase at a time $\leq t_0$. Since the final dependency of $P.e$ was selected from dependencies that were specified by the end of the Fast Accept phase (and hence by time t_0), and since $P'.n$ was constructed and proposed by P' after t_0, the final dependency of $P.e$ must be $< P'.n$. Hence, $Logs[P.e].dep < P'.n \; \forall e \leq i$. Since $Logs[P'.n].dep \geq P.i$, a cycle cannot exist between $P.i$ and $P'.n$, and thus $P.i$ will execute before $P'.n$.

$P.i$ also executes before $P.j$ since both entries appear in the same pilot's log and $P.j > P.i$. Since we already showed that $P.i$ executes before $P'.n$, and β appears in both entries $P.j$ and $P'.n$, at least one appearance of α (which is in $P.i$ in this case) is ordered before *both* appearances of β.

We now enumerate the possible execution orders between $P.i$ and $P'.m$ and show that α is executed before β in all cases. We use '$\rightarrow$' to indicate execution order, i.e., $x \rightarrow y$ indicates that x is executed before y.

Case 1: $P'.m \rightarrow P.i$.

This implies $P'.m \rightarrow P.j$ and $P'.m \rightarrow P'.n$. Since $\alpha \in Logs[P'.m].cmds$, α is executed first and before β. (The repeated command α in $P.i$ is ignored due to execution deduplication (§2.1).)

Case 2: $P.i \rightarrow P'.m$.

This implies the first appearance of α is in $P.i$. Hence the command $\alpha \in Logs[P.i].cmds$ is executed first and before β. (The repeated command α in $P'.m$ is ignored due to the deduplication.)

In all cases, we have shown that α is executed before β. Thus, Copilot replication satisfies the real-time ordering requirement of linearizability. ■

We now turn to the total order requirement of linearizability. We begin with some helper lemmas and definitions. First, we formally define *concurrent* entries (Definition 2), which characterizes the relationship between an entry in the pilot log and an entry in the copilot log. Then, we show that the log entries from different pilots cannot commit with incompatible dependencies (Lemma 3), and that two different values cannot be committed in the same log entry (Lemma 5). Finally, we show that Copilot provides a total order (Lemma 9) by showing that its execution (without the null dependency elimination optimization) provides a total order (Lemma 6), and then showing that null dependency elimination preserves the total order (Lemma 8).

Definition 2 *Two log entries $P.i$ and $P'.j$ are concurrent if $P.i$'s initial dependency $< P'.j$ and $P'.j$'s initial dependency $< P.i$.*

Lemma 3 *If two log entries $P.i$ and $P'.j$ from different pilots are committed, either $Logs[P.i].dep \geq P'.j$ or $Logs[P'.j].dep \geq P.i$.*

Proof. The two entries were either concurrent or they were not. Consider the case where $P.i$ and $P'.j$ were not concurrent, i.e., $P.i$'s proposed dependency $\geq P'.j$ or $P'.j$'s proposed dependency $\geq P.i$. Without loss of generality, assume that $P'.j$'s proposed dependency $\geq P.i$. This implies that entry $P'.j$ will have a dependency $\geq P.i$ when it commits, which means that a replica will either fast accept the proposed dependency or suggests a new dependency $> P.i$ (lines 17–23, Figure 4). Hence, $Logs[P'.j].dep \geq P.i$.

Now consider the case where $P.i$ and $P'.j$ were concurrent, i.e., $P.i$'s proposed dependency $< P'.j$ and $P'.j$'s proposed dependency $< P.i$. Since $P.i$ and $P'.j$ were committed and the Commit phase happens only after the FastAccept phase completes, $P.i$ and $P'.j$ must have received a FastAcceptReply from at least a majority $(f+1)$ of replicas (including the proposer). There are several subcases based on which phase the entries commit in.

Case 1: $P.i$ commits after the FastAccept phase, and $P'.j$ commits after the Accept phase (lines 17–21, Figure 3).

This means at least a fast path quorum of size $f + \lfloor (f+1)/2 \rfloor$ accepted the initially proposed dependency of $P.i$. This implies that at most $\lceil (f+1)/2 \rceil$ replicas accepted the initially proposed dependency of $P'.j$, and the remaining replicas replied to $P'.j$ with a suggested dependency $\geq P.i$ (since they had already accepted $P.i$'s proposed dependency). Hence, the new dependency chosen for $P'.j$ for the Accept phase must be $\geq P.i$ (lines 25–27, Figure 3).

Case 2: $P'.j$ commits after the FastAccept phase, and $P.i$ commits after the Accept phase (lines 17–21, Figure 3).

This is similar to case 1.

Case 3: Both $P.i$ and $P'.j$ commit after the Accept phase (lines 19–21, Figure 3).

Let $P'.j_0$ be the initially proposed dependency of $P.i$, and $P.i_0$ be the initially proposed dependency of $P'.j$. Since $P.i$ and $P'.j$ are concurrent, $P'.j_0 < P'.j$ and $P.i_0 < P.i$.

Let $\{P'.j_0, \ldots, P'.j_f, \ldots\}$ be the sorted set of dependencies $P.i$ received at the end of the FastAccept phase. Similarly, let $\{P.i_0, \ldots, P.i_f, \ldots\}$ be the sorted set of dependencies $P'.j$ received at the end of the FastAccept phase. We have: $P'.j_0 \leq \ldots \leq P'.j_f$ and $P.i_0 \leq \ldots \leq P.i_f$. Hence, the chosen dependency for the Accept and Commit phases of $P.i$ is $P'.j_f$ (lines 25–27, Figure 3). Similarly, the chosen dependency of $P'.j$ is $P.i_f$. Thus we have: $Logs[P.i].dep = P'.j_f$ and $Logs[P'.j].dep = P.i_f$.

By comparing these chosen dependencies to the proposed entries $P.i$ and $P'.j$, we show that either $Logs[P.i].dep \geq P'.j$ or $Logs[P'.j].dep \geq P.i$ in all cases.
Case 3.1: $P'.j \leq P'.j_f$. This implies that $Logs[P.i].dep = P'.j_f \geq P'.j$. The case where $P.i \leq P.i_f$ is analogous to this case.
Case 3.2: $P'.j > P'.j_f$. Since $Logs[P.i].dep = P'.j_f < P'.j$, we must show that $Logs[P'.j].dep \geq P.i$, i.e., $P.i_f \geq P.i$.

Since $P'.j > P'.j_f$, it follows that $\geq (f+1)$ replicas replied to the FastAccept of $P.i$ before the FastAccept of $P'.j$. This means $\geq (f+1)$ replicas replied to the FastAccept of $P'.j$ with a new dependency $\geq P.i$ (lines 17–23, Figure 4), which means that $\leq f$ replicas suggested dependencies $< P.i$. Hence, $P.i_f \geq P.i$ by contradiction, since if $P.i_f < P.i$, then $P.i_0 \leq \ldots \leq P.i_f < P.i$, which would imply that $> f$ replicas suggested dependencies for $P'.j$ that are $< P.i$. The case where $P.i > P.i_f$ is analogous to this case.

Case 4: Both $P.i$ and $P'.j$ commit after the FastAccept phase (lines 17–19, Figure 3).

This case is not possible: both entries cannot commit on the fast path without including each other in their dependencies. We prove this by contradiction: for $P.i$ and $P'.j$ to commit on the fast path without including each other, their fast path quorums must have no intersection. This would require a total of $\geq f + \lfloor (f+1)/2 \rfloor + f + \lfloor (f+1)/2 \rfloor = 3f+1$ replicas.

We have shown that in all cases, for two committed entries $P.i$ and $P'.j$, either $Logs[P.i].dep \geq P'.j$ or $Logs[P'.j].dep \geq P.i$. ■

Lemma 3 shows that it is impossible for two entries from different pilots to commit with incompatible dependencies. This prevents two entries from independently committing and executing without being aware of each other.

Lemma 4 *In a quorum Q of $f+1$ PrepareOk replies from Prepare(P, i, ballot), if the proposing pilot of some concurrent $P'.k$ is not in Q, P.i can commit on the fast path only if $\geq (\lfloor \frac{f+1}{2} \rfloor + 1)$ replicas in Q have fast accepted P.i.*

Proof. Assume that *(cmds, dep)* in $P.i$ has been committed on the fast path by a pilot P. This means at least $f + \lfloor \frac{f+1}{2} \rfloor$ replicas have fast accepted $P.i$. Since the pilot P' that proposed the conflicting $P'.k$ is not in Q, and P' could not fast accept $P.i$, at most $f-1$ replicas outside Q could have fast accepted $P.i$. Hence, there must be at least $(f + \lfloor \frac{f+1}{2} \rfloor) - (f-1) = \lfloor \frac{f+1}{2} \rfloor + 1$ replicas in Q that have fast accepted $P.i$. ■

Lemma 5 *For a committed log entry, all replicas will execute the same commands with the same dependency for that entry.*

Proof. Assume that $(cmds, dep)$ was committed in log entry $P.i$. In the normal case where there is no slowdown or failure, it is trivially true that only $(cmds, dep)$ can be committed in $P.i$, for the following reasons. (1) Pilot P will never propose a different value in $P.i$ because it advances its `latestEntry[P]` to be $P.i$ after proposing $(cmds, dep)$ in $P.i$ and only proposes new commands in the entries $> P.i$. (2) Only pilot P, which is the valid pilot in the current view and has ownership of $P.i$, can successfully get its proposed value of $(cmds, dep)$ in $P.i$ accepted and committed by a majority of replicas. This is because the replicas only accept proposals from the pilot in the current view and will reject proposals from any pilots in older views. If pilot P becomes slow or fails, a fast takeover is invoked by another pilot—either when a takeover timeout fires or during a pilot election—to complete the ordering work for entry $P.i$. We will show that the fast takeover procedure for $P.i$ correctly chooses the value of $(cmds, dep)$ and commits it in $P.i$.

When a pilot invokes a fast takeover of $P.i$, it runs the Prepare phase by sending Prepare messages to all replicas containing a ballot number that is higher than any ballot number it has seen for $P.i$. When it gathers at least $f+1$ PrepareOk replies, it calls the `choose_value()` procedure to pick the correct value of $P.i$ based on these replies. Let Q represent the quorum of PrepareOk replies. Let $P'.j$ be the initial dependency that the slow/failed pilot P proposed for $P.i$. (Note that the initial dependency $P'.j$ may or may not be the same as the committed dependency dep.) We examine the following cases based on which path (either the fast path or regular path) $P.i$ was committed on and based on when pilot P failed relative to the ordering phases.

Case 1: $(cmds, dep)$ was committed after the Accept phase (lines 19–21, Figure 3).

This means at least $f+1$ replicas have accepted $(cmds, dep)$ or $(cmds, dep)$ in $P.i$. Hence, Q must contain at least one PrepareOk reply that indicates the value of $(cmds, dep)$ as `ACCEPTED` or `COMMITTED` in $P.i$. If Q contains at least one reply with `COMMITTED` status, the fast takeover will run the Commit phase to commit $(cmds, dep)$ in log entry $P.i$, which is consistent with the earlier commit of $P.i$. Otherwise, Q contains at least one reply with `ACCEPTED` status, and the fast takeover procedure will run the Accept and Commit phases to commit $(cmds, dep)$ in log entry $P.i$, which is also consistent with the earlier commit of $P.i$.

Case 2: $(cmds, dep)$ was accepted by a majority in the Accept phase but the original pilot that proposed $(cmds, dep)$ failed before running the Commit phase (lines 19–21, Figure 3).

This is similar to case 1: another pilot will run the Accept and Commit phases to commit $(cmds, dep)$ in $P.i$ during its fast takeover of $P.i$. If the proposing pilot committed $(cmds, dep)$ and failed before sending out Commit messages, the recovered

value of $(cmds, dep)$ is consistent. If the proposing pilot did not commit *P.i* before it failed, the recovered value of $(cmds, dep)$ does not violate consistency since nothing had been committed in *P.i* by the proposing pilot.

Case 3: $(cmds, dep)$ was accepted by a fast path quorum in the FastAccept phase and the pilot committed locally but failed before broadcasting Commit messages (lines 17–19, Figure 3).

That means at least $f + \lfloor \frac{f+1}{2} \rfloor$ replicas (including the failed pilot) fast accepted $(cmds, dep)$. There should be at least $\lfloor \frac{f+1}{2} \rfloor$ replicas replying PrepareOk with the tuple $(cmds, dep)$ being `FAST_ACCEPTED`. In this case, we can infer that the PrepareOk reply from the original proposing pilot of *(cmds, dep)* is not in Q. To see why, assume that the PrepareOk reply is in Q. We show that this leads to a contradiction with the premise of this case, which states that there are no replies with the `COMMITTED` status. The original proposing pilot must have replied to the Prepare message after it committed *(cmds, dep)* in *P.i*. Otherwise (if it replied to the Prepare message before the commit), it could not commit *P.i* locally. Hence, the PrepareOk reply from the original proposing pilot must indicate that *(cmds, dep)* is committed, and there should be at least one reply with the `COMMITTED` status in Q. This is a contradiction. For the following sub-cases, we consider that the original proposing pilot is not in Q.

Case 3.1: There are at least $f+1$ replies with *(cmds, dep)* being fast accepted. It is safe to accept and commit *(cmds, dep)* in *P.i* because: (1) The committed value of *(cmds, dep)* is consistent with the value that could have been committed by the failed pilot. (The failed pilot either committed *(cmds, dep)* in *P.i* before it failed or did not commit any value in *P.i* before it failed.) (2) Any entry $P'.k > P'.j$ from P' would commit with a dependency $\geq$ *P.i* since *P.i* was fast accepted before $P'.k$ by at least a majority of replicas.

Case 3.2: There are f replies with *(cmds, dep)* being fast accepted. Since the original proposing pilot of *(cmds, dep)* is not in Q and a proposing pilot always fast accepts its entry, we can infer that at least $f+1$ replicas (including the failed pilot) have fast accepted *(cmds, dep)*. Hence, it is safe to commit *(cmds, dep)* in *P.i* for the same reasons stated in case 3.1.

Case 3.3: There are $< f$ and $\geq \lfloor \frac{f+1}{2} \rfloor$ replies with *(cmds, dep)* being fast accepted. The takeover process examines the status and value (i.e. commands and dependency) of the entries from pilot P' that are potentially concurrent with *P.i*. Let C be the set of such entries. C includes all the entries starting from $j+1$, the entry after the initial dependency $P'.j$, to `max_suggested_dep`, the largest suggested dependency among the replies in Q. (The entries after `max_suggested_dep` would have a dependency $\geq$ *P.i* when they commit since they would be accepted after *P.i* at a majority of replicas. Hence, we only need to examine the entries in C to infer the status of *P.i*.) Since *P.i* was committed on the fast path with a dependency $P'.j$, any entry in C will be committed with a dependency $\geq$ *P.i* (Lemma 3). We check whether all entries in C have been committed, which leads to two subcases.

Case 3.3.1: Every entry in C is committed with either a no-op or has a dependency $\geq$ *P.i*. Hence, the fast takeover can safely commit *(cmds, dep=* $P'.j$*)* in *P.i*. This committed value of *(cmds, dep=* $P'.j$*)* is consistent with that committed by the failed pilot. (Note that even if the failed pilot had not committed any value in *P.i* before it failed, the committed value of *(cmds, dep=* $P'.j$*)* does not violate consistency since nothing has been committed in *P.i*.

Case 3.3.2: Some entries in C have not been committed. We need to resolve the status of such entries first. Once their status is resolved, we can show that *P.i* can be correctly recovered similar to case 3.3.1. Let $P'.k$ be an unresolved entry in C. `choose_value()` will simultaneously prepare *P.i* and $P'.k$ and return the new quorum of replies Q_{ik}. Since *P.i* was committed on the fast path, we will show that $P'.k$ ($P'.k > P'.j$) will either be committed with no-op or acquire a dependency $\geq$ *P.i*. In either case, it is safe to commit *(cmds, dep=*$P'.j$*)* in *P.i*. If $P'.k$ can be conclusively resolved using the quorum Q_{ik}, $P'.k$ can only either commit with a no-op or have a dependency $\geq$ *P.i* because *P.i* was committed on the fast path with dependency $P'.j$, which is $< P'.k$ (Lemma 3).

Now we consider the case that $P'.k$ cannot be conclusively resolved using the quorum Q_{ik}. Examining whether *P.i* and $P'.k$ are concurrent leads to two subcases below, but first we establish some premises for the subcases. (1) The failed pilot that proposed *P.i* is not in Q_{ik} as we have shown earlier for case 3. (2) The proposing pilot of $P'.k$ is not in Q_{ik} either, otherwise $P'.k$ would have been conclusively resolved by `choose_value_common()` (lines 9–19, Figure 8). (3) *P.i* and $P'.k$ each have $\geq \lfloor \frac{f+1}{2} \rfloor$ and $< f$ replies with `FAST_ACCEPTED` status because neither *P.i* nor $P'.k$ can be conclusively resolved using the new quorum Q_{ik} (line 20, Figure 8). (4) At least $f+1$ replicas in Q_{ik} have received and replied to the FastAccept for *P.i* and $P'.k$ (lines 21–24, Figure 8). We now examine the two subcases based on whether *P.i* and $P'.k$ are concurrent.

Case 3.3.2.1: *P.i* and $P'.k$ are concurrent. Q_{ik} must have $> \lfloor \frac{f+1}{2} \rfloor$ replies from the replicas that have fast accepted *P.i* because the original proposing pilot of $P'.k$ is not in Q_{ik} and the failed pilot P had committed *P.i* on the fast path before it failed (Lemma 4). Since $P'.k$ and *P.i* are concurrent and $> \lfloor \frac{f+1}{2} \rfloor$ replicas in Q_{ik} have fast accepted *P.i*, there must be $< f+1-\lfloor \frac{f+1}{2} \rfloor = \lceil \frac{f+1}{2} \rceil \leq \lfloor \frac{f+1}{2} \rfloor + 1$ replicas in Q_{ik} that have fast accepted $P'.k$. Hence, $P'.k$ could not have committed on the fast path (Lemma 4) and it is safe to commit a no-op in $P'.k$.

Case 3.3.2.2: *P.i* and $P'.k$ are not concurrent (i.e., the initial dependency of $P'.k \geq$ *P.i*). $P'.k$ will eventually be committed with a no-op or have a dependency $\geq$ *P.i*, since $P'.k$'s initially proposed value has been received by a majority of replicas, each of which either accepted $P'.k$'s initial dependency or suggested a new dependency that is $> P'.k$'s initial dependency.

We have shown that in all cases, the fast takeover procedure safely and correctly recovers the value that has been committed in *P.i* by the failed pilot.

Next we consider the case where the recovering replica fails while doing the fast takeover of *P.i*. In this case, another replica will eventually invoke a fast takeover of *P.i*. Since the fast takeover procedure recovers the correct value and commits it using the regular Accept and Commit phases of canonical Paxos, an argument similar to Paxos's correctness shows that only the correct value will ever be committed in *P.i* despite repeated failures and recoveries. Specifically, if a recovering replica commits the correct value in *P.i* and then fails, another recovering replica will only choose this correct value and commit it in *P.i*. If no recovering replica has committed the correct value in *P.i*, another recovering replica is still able to choose the correct value that has potentially been committed by the failed pilot and commit it in *P.i*, as we have shown above.

We have shown that the fast takeover procedure can correctly recover the value that has been committed in *P.i* by the failed pilot. We have also shown that if a recovering replica fails during its fast takeover of *P.i*, another recovering replica can also recover the correct value that has potentially been committed in *P.i*. Hence, only one value—commands and dependency—can be committed in an entry. ■

Lemma 6 *The execution algorithm executes log entries in the same order across all replicas.*

Proof. The execution algorithm always preserves the implicit ordering between entries from the same pilot. This is trivially true since a pilot/replica processes the log of a pilot in increasing order of log entry number. In other words: $P.0 \rightarrow P.1 \rightarrow \ldots$, and similarly, $P'.0 \rightarrow P'.1 \rightarrow \ldots$.

We now show that the execution algorithm executes the log entries of pilots' logs in the same total order. We assume that the null dependency elimination optimization is not being used; later in Lemma 8, we will show that null dependency elimination preserves the total ordering of commands. (Note that the ping-pong batching optimization is not discussed as it does not affect correctness.) The proof is by induction. Let i be the next log entry number of P that has not been executed, and let j be the next log entry number of P' that has not been executed. Without loss of generality, assume that P has higher execution priority than P' (recall that this is used to deterministically break cycles between two entries from different pilots).

Case 0 (base case): $i = 0$, $j = 0$.

If $P.0$ has no dependency and $Logs[P'.0].dep = P.0$, $P.0$ is executed first. Similarly, If $P'.0$ has no dependency and $Logs[P.0].dep = P'.0$, $P'.0$ is executed first.

If $Logs[P.0].dep = P'.0$ and $Logs[P'.0].dep = P.0$, this implies a cycle exists between $P.0$ and $P'.0$. Since P has higher priority than P', $P.0$ is executed first.

Note that the case where both $P.0$ and $P'.0$ have no dependencies cannot happen because it must be the case that either $Logs[P.0].dep \geq P'.0$ or $Logs[P'.0].dep \geq P.0$ (Lemma 3).

Assume the execution has executed up to entry $P.i_0$ of P's log and up to entry $P'.j_0$ of P''s log, and there is a total ordering between the log entries of P and P' thus far. We will show that regardless of whether the execution algorithm starts with $i = i_0 + 1$ or $j = j_0 + 1$ next, the execution history is expanded and maintains a total ordering across replicas. We enumerate all possible cases based on the execution status—i.e., whether an entry has been executed—of $P.i$'s dependency ($Logs[P.i].dep$) and the execution status of $P'.j$'s dependency ($Logs[P'.j].dep$). In each case, we will show that whether the execution starts with $P.i$ or $P'.j$, it will result in the same entry being executed next. There are three main cases.

Case 1: $Logs[P.i].dep$ has been executed.

This implies $Logs[P.i].dep \leq P'.j_0$ since the execution has executed up to the entry $P'.j_0$ of P''s log. If the execution starts with $P.i$, then $P.i$ can be executed first because $Logs[P.i].dep$ has been executed.

Now consider the case where the execution starts with $P'.j$. By Lemma 3, we know that $Logs[P'.j].dep \geq P.i$ since $Logs[P.i].dep \leq P'.j_0 < P'.j$. If no cycle exists between $P.i$ and $P'.j$, the execution algorithm cannot execute $P'.j$ yet and will switch to the next entry of P, which is $P.i$. $P.i$ can be executed immediately since its dependency, $Logs[P.i].dep$, has been executed. If a cycle exists between $P.i$ and $P'.j$, this implies that $\exists e \in [0, i_0]$ s.t. $Logs[P.e].dep \geq P'.j$, which implies that a cycle exists between $P'.j$ and $Logs[P'.j].dep$. Since P' has lower priority than P, the execution algorithm will execute $Logs[P'.j].dep$ first, which requires executing $P.i$ first. Hence, whether a cycle exists between $P.i$ and $P'.j$ or not, $P.i$ will be executed first.

In case 1, we have shown that $P.i$ is always executed first regardless of whether the execution starts with $P.i$ or $P'.j$.

Case 2: $Logs[P.i].dep$ has not been executed and $Logs[P'.j].dep$ has not been executed.

This implies $Logs[P.i].dep \geq j$ and $Logs[P'.j].dep \geq i$, which implies a cycle exists between $P.i$ and $P'.j$. Since P has higher priority than P', $P.i$ is executed first regardless of whether the execution algorithm starts with $P.i$ or $P'.j$.

Case 3: $Logs[P.i].dep$ has not been executed and $Logs[P'.j].dep$ has been executed.

This implies $Logs[P.i].dep \geq j$. We show that a cycle cannot exist between $P.i$ and $P'.j$. We prove this by contradiction. Assume that a cycle exists between $P.i$ and $P'.j$. This implies that $\exists e' \in [0, j_0]$ s.t. $Logs[P'.e'].dep \geq P.i$, which implies a cycle

exists between $P.i$ and $P'.e'$. Since P has higher priority than P', $P.i$ must be executed before $P'.e'$. We know that $P'.e'$ has been executed because $P'.e' \leq P'.j_0$. Since $P.i$ must be executed before $P'.e'$ and $P'.e'$ has been executed, $P.i$ must have been executed. This contradicts the assumption that $P.i$ has not been executed. Hence, a cycle cannot exist between $P.i$ and $P'.j$.

If the execution starts with $P'.j$, then $P'.j$ can be executed first since $Logs[P'.j].dep$ has been executed. If the execution starts with $P.i$, then $P.i$ cannot be executed since $Logs[P.i].dep$ has not been executed. Hence, the execution algorithm switches to the next entry of P''s log, which is $P'.j$. $P'.j$ can be executed since its dependency, $Logs[P'.j].dep$, has been executed.

In case 3, we have shown that $P'.j$ is always executed first regardless of whether the execution starts with $P.i$ or $P'.j$.

Cases 1, 2, and 3 are exhaustive. Hence, the execution algorithm executes log entries in the same total order across all replicas. ■

The ping-pong batching optimization used by Copilot does not affect correctness. We now show that the null dependency elimination optimization preserves the correctness of Copilot. We start with a helper lemma that shows that the commands in a nullified dependency either are no-ops or will not change (Lemma 7). Then, we use this lemma to show that null dependency elimination preserves a total ordering (Lemma 8).

Lemma 7 *If cmds are the commands that are nullified at $P'.j$, then either cmds or a no-op will eventually be committed in $P'.j$.*

Proof. Assume that $P.i$ is the committed entry we consider executing next and $P'.j$ is its dependency, which we try to nullify. Let *cmds* be the commands that pilot P' proposes for its entry $P'.j$. Let V_m be the current view of pilot P'. A dependency $P'.j$ satisfies null dependency elimination if (1) at least a majority of replicas have the same current view V_m as pilot P', (2) at least a majority of replicas have advanced their `latestEntry[P']` to be $\geq P'.j$, and (3) the commands with the same ID as those in *cmds* have already been executed. Assume that entry $P'.j$ satisfies null dependency elimination. We will show that in all cases—both normal operation and slowdown/failure cases—either *cmds* or a no-op will be committed in $P'.j$.

After pilot P' proposes its *cmds* in log entry $P'.j$, it advances its `latestEntry[P']` to be $P'.j$ and will only propose new commands in the next entry, which is after $P'.j$. Hence, it will never propose any commands other than *cmds* in $P'.j$. If there are no failures, P' will commit its initially proposed *cmds* in $P'.j$ either on the fast path or the regular path. If P' becomes slow or fails, a fast takeover and/or a view change may occur. We consider each case.

If a fast takeover of $P'.j$ occurs, the fast takeover procedure can only ever commit the commands that have been proposed for entry $P'.j$, or a no-op—i.e., it will never introduce any new commands.

We now show that if a view change for pilot P' occurs, the newly elected pilot in the new view $V_n > V_m$ will only propose its commands starting from a log entry that is $> P'.j$. Since $P'.j$ satisfies null dependency elimination, at least a majority quorum of replicas Q_1 are in the same view V_m and have `latestEntry[P']` $\geq P'.j$. In order to succeed in the view change, the new pilot of the new view V_n must gather at least $f+1$ ViewChangeOk replies from at least a majority quorum Q_2 of replicas. Due to quorum intersection, at least one replica in Q_1 must be in Q_2, and this replica has `latestEntry[P']` $\geq P'.j$. Since the new pilot keeps track of the latest entry in pilot P''s log when receiving a ViewChangeOk reply, it knows that the latest entry is $\geq P'.j$ after receiving ViewChangeOk replies from the replicas in Q_2. Hence, the new pilot sets its `latestEntry[P']` to be $\geq P'.j$ when it starts the new view, and will only propose its commands starting from a log entry $> P'.j$.

We have shown that if *cmds* are the commands that are nullified at $P'.j$, then either *cmds* or a no-op (and nothing else) will eventually be committed in $P'.j$. ■

Lemma 8 *Copilot with the null dependency elimination optimization preserves the total ordering of commands.*

Proof. Null dependency elimination allows the execution of a committed entry $P.i$ if all entries $\leq Logs[P.i].dep$ that have not been executed from P''s log can be nullified. This results in $P.i$ being executed before the nullified entries.

Assume that all entries $P.e < P.i$ have been executed. Let $P'.j$ be $Logs[P.i].dep$ and $P'.e'$ be the next entry of P''s log that has not been executed.

In this proof, we focus on the case where the null dependency elimination is invoked and all entries in $[P'.e', P'.j]$ can be nullified. With the optimization, $P.i$ is executed before $P'.e'$ regardless of the committed dependencies of the entries in $[P'.e', P'.j]$. Executing $P.i$ before $P'.e'$ also implies executing $P.i$ before all entries $> P'.j$ from P''s log. However, another replica may order and execute some entries $> P'.j$ before $P.i$. For example, this can happen due to null dependency elimination if the dependency of these entries (e.g., $P.i$) can be nullified. Hence, to show that null dependency elimination preserves the total execution order, we need to show that all possible relative orderings between $P.i$ and the nullified entries in $[P'.e', P'.j]$ and the entries $> P'.j$ that are potentially executed before $P.i$ will lead to an execution history which is equivalent to the execution history where $P.i$ is executed first. We examine two cases based on whether there exists an execution where some entry $> P'.j$ is ordered and executed before $P.i$.

Case 1: There exists no execution where some entry $> P'.j$ is executed before $P.i$.

This implies $P.i$ is executed before all entries $> P'.j$ in all executions. Thus, we only need to examine the possible orderings among $P.i$, $P'.e'$ and $P'.j$. If $P.i$ is ordered before $P'.e'$, this implies that $P.i \rightarrow P'.e' \rightarrow P'.j$. Since the commands in the nullified entries in $[P'.e', P'.j]$ either are no-ops or will not change (Lemma 7), and since the null dependency elimination assumes they have been executed, they will not be executed again. Hence, the command execution history will not contain any commands from $[P'.e', P'.j]$. Thus, the resulting execution is identical to executing $P.i$ before all entries in $[P'.e', P'.j]$ as a result of null dependency elimination.

The case where $P.i$ is ordered after $P'.j$ and the case where $P.i$ is ordered after $P'.e'$ but before $P'.j$ can be argued similarly. In all cases, since all commands from the entries in $[P'.e', P'.j]$ have been executed and will not be executed again, the resulting execution is the same regardless of the relative orderings between $P.i$ and the entries in $[P'.e', P'.j]$.

In case 1, we prove that executing $P.i$ first without knowing the committed dependencies of the entries in $[P'.e', P'.j]$ does not violate the total order of commands.

Case 2: There exists at least one execution where some entry $> P'.j$ is executed before $P.i$.

Assume $P'.k$ is the entry that is $> P'.j$ in P''s log and is executed before $P.i$. By Lemma 3, we can infer that $P'.k$ commits with a dependency $\geq P.i$, since $Logs[P.i].dep = P'.j < P'.k$. There are two possible scenarios where case 2 may occur: (1) a cycle exists between $P.i$ and $P'.k$ and P' has higher priority than P; and (2) $P'.k$ can successfully nullify all entries $\leq Logs[P'.k].dep$ in P's log that have not been executed. We first show that (1) cannot happen and then focus on (2).

We show that (1) cannot happen by contradiction. Assume that a cycle exists between $P.i$ and $P'.k$ and P' has higher priority than P. Since $Logs[P.i].dep = P'.j < P'.k$ and $Logs[P'.k].dep \geq P.i$, there must exist an entry $P.i_0 < P.i$ such that $Logs[P.i_0].dep \geq P'.k$ (in order for a cycle to exist between $P.i$ and $P'.k$). This implies that a cycle also exists between $P.i_0$ and $P'.k$. Since P' has higher priority than P, $P'.k$ must be executed before $P.i_0$. But $P.i_0$ has already been executed since all entries $< P.i$ have been executed, which means $P'.k$ must have been executed since it must be executed before $P.i_0$. This contradicts our assumption that $P'.k$ has not been executed.

We now focus on scenario (2). Since $P.i$ is ordered after $P'.k$, we have $P'.e' \rightarrow P'.j \rightarrow P'.k \rightarrow P.i$. Since all the entries $\leq Logs[P'.k].dep$ can be nullified, all commands having the same ID as the commands in those entries have already been executed. Execution deduplication will thus avoid re-executing the commands in those entries, which include $P.i$ since $Logs[P'.k].dep \geq P.i$ (by Lemma 3, as argued above). Hence, the execution history will not contain $Logs[P.i].cmds$, which implies that the execution leads to the same total order as the execution where $P.i$ is ordered before all entries in $[P'.e', P'.j]$ due to null dependency elimination: $P.i \rightarrow P'.e' \rightarrow P'.j \rightarrow P'.k$.

In all cases, we have shown that the null dependency elimination optimization preserves the total order of commands. ■

Combining the above lemmas, we can now show that Copilot satisfies the total order property.

Lemma 9 *All replicas execute commands in the same total order.*

Proof. Since replicas execute the log entries of both pilots in the same order (Lemma 6), and the commands in a log entry are the same across all replicas (Lemma 5), replicas will execute the same exact sequence of commands. For duplicate commands, only the first appearance in the sequence is executed (remaining appearances are ignored). Hence, all commands are executed only once and in the same total order. We further prove that null dependency elimination preserves the total ordering of commands (Lemma 8). Hence, Copilot guarantees a total ordering of commands across replicas. ■

Lemma 10 *Copilot provides linearizability.*

Proof. Copilot satisfies the real-time order (Lemma 1) and total order (Lemma 9) requirements of linearizability. ■

4.2 Liveness

We now show that Copilot replication satisfies liveness—i.e., all client commands eventually complete—and specify the conditions necessary for this to hold.

Due to FLP [11], no protocol can be both safe and live in a completely asynchronous setting. Since we provide safety in an asynchronous setting, we must make assumptions about synchrony in order to guarantee liveness. Copilot guarantees liveness under the following assumptions:

- No more than f replicas are faulty.
- A majority of replicas can communicate with each other within a timeout, and messages eventually arrive at their destination before their receiver times out.
- A client keeps resending its command after a timeout until its command is successfully completed.

We start by showing that the fast takeover procedure and view change protocol make progress and eventually complete. (Lemmas 11 and 12). Then we use these facts to show that Copilot makes progress by eventually committing a client command, executing it, and replying to the client (Lemma 13).

Lemma 11 *The fast takeover procedure makes progress and eventually terminates.*

Proof. We will show that the fast takeover procedure for an entry will eventually make progress by choosing a value and committing it in the entry. Recall that a fast takeover is invoked when a pilot tries to take over an entry from the other pilot's log, or during a view change to fill in the gaps in a pilot's log.

We focus on the steps in the fast takeover procedure that may prevent Copilot from making progress—specifically, those that require sending messages and waiting for responses from the replicas. (It is obvious that the other steps of the fast takeover procedure—e.g., an iteration over a finite set of entries—terminate.) The fast takeover procedure uses Paxos's Prepare and Accept phases to prepare an entry and get a value accepted (and then committed) by at least a majority of replicas. Since canonical Paxos guarantees liveness under the partial synchrony assumptions made by Copilot, these phases in Copilot also make progress by the same arguments. The simultaneous prepare phase in Copilot is similar to the Prepare phase and also needs replies from at least a majority of replicas to make progress. By the same argument, this phase will also eventually complete.

Note that as in Paxos, the dueling proposers problem may occur and prevent the fast takeover procedure from successfully preparing and/or getting a value accepted by a majority of replicas—e.g., because the replicas have promised to another proposal with a higher ballot number. In such cases, Copilot retries and applies the common technique of random exponential backoff. The retries will not repeat indefinitely and the fast takeover procedure eventually completes its Prepare/Accept phases by relying on the partial synchrony assumptions and arguments that Paxos [17] makes to ensure progress.

When a fast takeover sends a FastAccept of the initial value to the replicas that have not received the FastAccept from the failed pilot (in order to gather at least $(f+1)$ replies to the FastAccept message) (lines 21–24, Figure 8), it will succeed if there is no dueling proposer and this step will not be repeated. In the presence of dueling proposers, some replicas may not fast accept the initial value because they have promised to another proposal with a higher ballot number in the recovered entry. In this case, the recovering replica may have to retry the fast takeover. By the same arguments regarding dueling proposers above, a fast takeover will eventually make progress: either it gets more replies to its FastAccept message, or it learns that at least a majority of replicas have replied to another FastAccept and thus avoids repeating this step (which has been completed by another fast takeover of the same entry).

We have shown that the fast takeover procedure makes progress and eventually terminates. ■

Lemma 12 *Copilot's view change protocol makes progress and eventually terminates.*

Proof. We will show that Copilot's view change protocol makes progress by successfully forming a new view. Our view change protocol (§3.3) is similar to the view change protocol described for canonical Paxos [21].

The first two phases of our view change protocol, `startViewChange` and `acceptView`, require waiting for at least $f+1$ replies before taking the next step. By our assumptions, at least $f+1$ replicas are alive and can communicate with each other, so these phases will eventually make progress. Without any concurrent view managers (other replicas) trying to invoke view changes, a view manager will be able to get at least a majority of replicas to agree with its view change request (lines 7–8, Figure 9) and accept its new view (lines 27–28, Figure 9). Once a majorit of replicas have accepted the new view, the view manager can inform the other replicas to start the new view and a new pilot is successfully elected.

In the presence of concurrent view managers that initiate a view change at the same time, a view manager may not be able to successfully form a new view, similar to the dueling proposers problem. In such cases, a view manager may have to retry until a new view or a new pilot is elected. The retries will not repeat indefinitely by relying on the same partial synchrony assumptions and arguments that Multi-Paxos [21] makes to ensure progress. (As before, the common techniques of random exponential backoff helps increase the likelihood that a view manager eventually succeeds in forming a new view.)

Copilot's view change protocol uses the fast takeover procedure (§3.2) to resolve the holes in a pilot's log. Lemma 11 shows that fast takeovers make progress and will eventually complete.

Hence, Copilot's view change protocol makes progress and eventually forms a new view. ■

Lemma 13 *Copilot makes progress by eventually committing and executing a client command, and replying to the client.*

Proof. Assume that a client sends its command α to both pilots and at least one pilot receives the command α. (At least one pilot is available in the system because if one or both pilots fail, Copilot will initiate a view change to elect new pilots, which will eventually complete as shown in Lemma 12.) Without loss of generality, assume that pilot P puts command α in its log entry $P.i$ and proposes its entry $P.i$.

First, we show that command α will eventually be committed, and then we show that it will be executed and a client will receive a reply. To show that command α will eventually be committed, we first consider the failure-free case and then discuss failures or network partitions.

If there are no failures, a pilot can make progress in its FastAccept phase and Accept phase as long as it receives replies from at least a majority of replicas. (A pilot always resorts to the regular path if it cannot gather a fast path quorum of replies—the timeout prevents a pilot from waiting forever for more replies beyond $f+1$.) By the same partial synchrony assumptions and arguments as Paxos [17], a pilot is guaranteed to gather a majority of replies and complete its FastAccept phase and Accept phase, and commit the entry *P.i*.

Now consider the case of failures. If a pilot fails while committing *P.i*, a fast takeover of *P.i* is invoked or a view change occurs to elect a new pilot, which will call the fast takeover procedure to resolve *P.i*. The fast takeover procedure and view change will eventually complete as we have shown in Lemmas 11 and 12. If the failed pilot commits a value in *P.i* before it fails, the fast takeover guarantees that only this value, which contains the command α, can ever be committed in *P.i*. If the failed pilot does not commit any value in *P.i* before it fails, the fast takeover may commit a no-op in *P.i*. In this case, the client may not receive a reply for its command α and will resend the command to the pilots (including the newly elected pilot) if it has not received a reply from either pilot within a timeout. At least one pilot will eventually commit its entry (which contains the command α) without failing—the newly elected pilots cannot keep failing because we assume that at most f replicas (including the pilots) can fail for Copilot to guarantee liveness.

If a network partition occurs and the pilot is in the minority side of the partition, then it may not be able to commit its entry *P.i* because it cannot gather a majority quorum of replies during the FastAccept/Accept phase. In this case, a replica in the majority side will eventually detect and initiate a view change to elect a new pilot (which will also be in the majority side). The client will eventually resend its command to both pilots (including the newly elected pilot) if it has not received a reply within a timeout. As we have shown above, at least one pilot eventually commit its entry which contains the command α.

We now have that command α is committed in entry *P.i*. We show that command α will eventually be executed by showing that either *P.i* is executed or the same command α, appearing in some other log entry, has been executed.

Consider the case where execution switches to pilot *P*'s log and tries to execute *P.i*. If executing *P.i* requires executing some entries that have not been committed in the other pilot's log—e.g., because the other pilot has failed, then those entries will be resolved when a timeout triggers a fast takeover and/or a view change. A fast takeover or view change will eventually complete as we have shown in Lemmas 11 and 12. Once those entries are committed and executed, *P.i* can be safely executed. If the command α has not been executed, it will be executed for the first time when *P.i* is executed. If the command α has already been executed, then execution deduplication will avoid executing command α in *P.i* (§2.1). (Recall that the same command α can appear in a committed entry in the other pilot's log because the client sends command α to both pilots and the other pilot may have successfully committed command α.) In both cases, the command α is executed.

Consider the case where execution never switches to pilot *P*'s log to execute *P.i*. This case can only happen if the entries in the other pilot's log can always nullify their dependencies on the entries in *P*'s log, which include *P.i*. An entry in the other pilot's log can nullify *P.i* only if all the commands in *P.i* (including command α) have been executed. In this case, command α has also been executed.

We have shown that the command α will eventually be executed. When it is executed for the first time, Copilot will send a reply to the client.

We have shown that Copilot guarantee liveness by showing that a client command submitted to Copilot will eventually be committed and executed, and a client will eventually receive a reply for its command. ■

5 Conclusion

Copilot replication is the first 1-slowdown-tolerant consensus protocol. Its pilot and copilot both receive, order, execute, and reply to all client commands. It uses this proactive redundancy and a fast takeover mechanism to provide slowdown tolerance. Despite its redundancy, Copilot replication's performance is competitive with existing consensus protocols when no replicas are slow. When a replica is slow, Copilot is the only consensus protocol that avoids high latencies.

Acknowledgements. We thank our shepherd, Allen Clement, and the anonymous reviewers for their insights and help in refining the ideas of this work. We are grateful to Christopher Hodsdon and Jeffrey Helt for their feedback. This work was supported by the National Science Foundation under grant number CNS-1827977.

References

[1] M. K. Aguilera and M. Walfish. No time for asynchrony. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2009.

[2] P. Ajoux, N. Bronson, S. Kumar, W. Lloyd, and K. Veeraraghavan. Challenges to adopting stronger consistency at scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] L. Alvisi, A. Clement, M. Dahlin, M. Marchetti, and E. Wong. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[4] M. Brooker, T. Chen, and F. Ping. Millions of tiny databases. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

[5] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, et al. Windows Azure Storage: A highly available cloud storage service with strong consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[7] Cockroach DB. `https://www.cockroachlabs.com/product/`, 2021.

[8] Copilot. `https://github.com/princeton-sns/copilot`, 2021.

[9] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[10] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), 1985.

[12] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1990.

[13] P. Huang, C. Guo, L. Zhou, J. R. Lorch, Y. Dang, M. Chintalapati, and R. Yao. Gray failure: The achilles' heel of cloud-scale systems. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[14] P. Huang, C. Guo, J. R. Lorch, L. Zhou, and Y. Dang. Capturing and enhancing in situ system observability for failure detection. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[15] M. Isard. Autopilot: Automatic data center management. *Operating Systems Review*, 41(2), 2007.

[16] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2), 1998.

[17] L. Lamport. Paxos made simple. *ACM Sigact News*, 32, 2001.

[18] J. B. Leners, H. Wu, W.-L. Hung, M. K. Aguilera, and M. Walfish. Detecting failures in distributed systems with the falcon spy network. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[19] C. Lou, P. Huang, and S. Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[20] Y. Mao, F. P. Junqueira, and K. Marzullo. Mencius: Building efficient replicated state machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[21] D. Mazières. Paxos made practical. `http://www.scs.stanford.edu/~dm/home/papers/paxos.pdf`, 2007.

[22] Y. Mei, L. Cheng, V. Talwar, M. Levin, G. Jacques-Silva, N. Simha, A. Banerjee, B. Smith, T. Williamson, S. Yilmaz, W. Chen, and G. J. Chen. Turbine: Facebook's Service Management Platform for Stream Processing. In *International Conference on Data Engineering (ICDE)*, 2020.

[23] I. Moraru, D. G. Andersen, and M. Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[24] K. Ngo, S. Sen, and W. Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[25] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computer Surveys*, 22(4), 1990.