# Tolerating Slowdowns in Replicated State Machines

Quang Minh Khiem Ngo

A Dissertation
Presented to the Faculty
of Princeton University
in Candidacy for the Degree
of Doctor of Philosophy

Recommended for Acceptance
by the Department of
Computer Science
Adviser: Wyatt Andrew Lloyd

September 2021

# Abstract

Replicated state machines (RSMs) are linearizable, fault-tolerant groups of replicas that are coordinated using a consensus algorithm. RSMs are used throughout large-scale systems, such as distributed databases, cloud storage, and service managers. At such scale, it is common for some machines to be slow. In this dissertation, we address the problem of developing slowdown-tolerant RSMs that continue to operate as fast RSMs despite the presence of slow replicas. We define *s*-slowdown-tolerance and identify why existing consensus protocols are not slowdown-tolerant. As the first and pragmatic step toward slowdown-tolerant RSMs, we design, implement, and evaluate two 1-slowdown-tolerant consensus protocols, Copilot and Latent Copilot.

Copilot replication is the first 1-slowdown-tolerant consensus protocol: it delivers normal latency despite the slowdown of any 1 replica. Copilot uses two distinguished replicas—the pilot and copilot—to proactively add redundancy to all stages of processing a client's command. Copilot uses dependencies and deduplication to resolve potentially differing orderings proposed by the pilots. To avoid dependencies leading to either pilot being able to slow down the group, Copilot uses fast takeovers that allow a fast pilot to complete the ongoing work of a slow pilot. Copilot includes two optimizations—ping-pong batching and null dependency elimination—that improve its performance when there are 0 and 1 slow pilots respectively. Our evaluation of Copilot shows its performance is lower but competitive with Multi-Paxos and EPaxos when no replicas are slow. When a replica is slow, Copilot is the only protocol that avoids high latencies.

Latent Copilot, a variant of Copilot, is another design and implementation of a 1-slowdown-tolerant consensus protocol. Latent Copilot operates with one active pilot, which actively proposes commands, and one latent pilot, which proposes commands only when necessary. In this way, Latent Copilot achieves an intermediate tradeoff between Multi-Paxos and Copilot in terms of throughput and slowdown tolerance. Our evaluation of La-

tent Copilot shows that it achieves 1-slowdown-tolerance and that its performance with no slowdowns is comparable to Multi-Paxos.

Finally, we generalize the design of Copilot replication to handle more than one slowdown. We provide the design of an $s$-slowdown-tolerant protocol that achieves slowdown tolerance despite $s$ slow replicas.

# Acknowledgements

I would like to thank my adviser, Wyatt Lloyd, for his unflagging support, guidance, and encouragement during my PhD. Wyatt cares about the growth of his students. He invested a significant amount of time and effort to help me build a strong research background and important skills to become a good researcher. It all started with our weekly 1-hour research meetings and countless whiteboard sessions where I learnt from him about designing systems, back-of-the-envelope calculations, running experiments on a whiteboard and drawing expected graphs. Wyatt was always generous with his time and help whenever I needed it. He gave insightful feedback and advice to help me improve my research work and research skills. My research interests in designing and building practical distributed systems that are rooted in a strong theoretical background bear his influence. I also learnt from him his positive attitude. He showed encouragement and excitement in every research finding I shared. I am very grateful to Wyatt for my rewarding and memorable PhD experience.

I had a great opportunity to work with Siddhartha Sen when we collaborated on the slowdown tolerance project. Sid also served as unofficial secondary adviser during my time at Princeton. During our collaboration, Sid often showed encouragement and support, patiently gave me insightful feedback on my research, and shared useful advice to help improve my research skills. I learnt from Sid about writing a well-structured and coherent proof when we worked together on the correctness proof of Copilot. I would also like to thank the other members of my dissertation committee—Amit Levy, Ravi Netravali, and Michael Freedman—for their advice that helped significantly improve my thesis.

I would like to thank my undergraduate thesis adviser, Wei Tsang Ooi, and Mun Choon Chan for the research guidance during my time at the National University of Singapore, which helped discover my interest in research and set me on the path to graduate school.

During my PhD, I spent much time in the research labs—USC NSL and Princeton SNS—where I found a vibrant intellectual and supportive community. I would like to thank the faculty members—Ramesh Govindan, Ethan Katz-Bassett, Minlan Yu, Wyatt Lloyd,

Amit Levy, and Michael Freedman—for their commitment to maintaining a collaborative and supportive lab environment. I made many supportive friends along the way: Chris Hodsdon, Andrew Or, Jeffrey Helt, Zhenyu Song, Theano Stavrinos, Haonan Lu, Jennifer Lam, Ashwini Raina, David Liu, and Yi-Ching Chiu. I want to thank Chris for being my lunch buddy at USC and Princeton, and for the conversations about research and countless random topics—e.g., his bread baking experience. I would like to thank Chris, Haonan, Jeff, Theano, Ethan, and Matt Burke for reading my paper drafts and giving extensive feedback to improve them. I would also like to thank my intern managers, Suresh Pasala and Bertan Ari, for their guidance and support during my summer internship at Facebook.

The research in this dissertation was supported by the National Science Foundation under a CNS Award (#1827977).

Finally, I would like to thank my family for their support. I am grateful to my parents for their unconditional love and hard work to give me and my brothers what is necessary to pursue our career paths. My brother Khoi deserves special thanks for being my role model and giving me guidance and help whenever I need it.

To my family.

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Replicated state machines (RSMs) are linearizable, fault-tolerant groups of replicas coordinated by a consensus algorithm [48]. Linearizability gives the RSM the illusion of being a single machine that responds to client commands one by one [22]. Fault-tolerance enables the RSM to continue operating despite the failure of a minority of replicas. Together, these make RSMs operate as single machines that do not fail.

RSMs are used to implement small services that require strong consistency and fault tolerance, whose work can be handled by a single machine. They are used throughout large-scale systems, such as distributed databases [15, 14], cloud storage [7, 10], and service managers [26, 40]. While each RSM is individually small, their pervasive use at scale means that they collectively use many machines. At such scale, it is common for some machines to be slow [16, 2]. These slowdowns arise for a myriad of reasons, including misconfigurations, host-side network problems, partial hardware failures, garbage collection events, and many others. The slowdowns manifest as machines whose latency for responding to other machines is higher than usual.

Thus, RSMs should also be slowdown-tolerant, i.e., provide similar performance despite the presence of slow replicas. Unfortunately, no existing consensus protocol is slowdown-tolerant: a single slow replica can sharply increase their latency. This increased

latency decreases availability because a service that does not respond in time is not meaningfully available [21, 6, 7, 50].

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 should eventually detect long-term slowdowns and reconfigure the slow replica out of the RSM to restore normal performance [3, 25, 24, 36, 1, 33]. What remains unsolved is how to tolerate transient slowdowns in general and how to tolerate long-term slowdowns in the time between their onset, their eventual detection, and the end of reconfiguration.

Our ultimate goal is to develop slowdown-tolerant RSMs that continue to operate as fast RSMs despite the presence of slow replicas. Given the general rarity of slowdowns, however, it is unlikely that a single RSM will contain multiple slow replicas at the same time. Thus, we target the first and most pragmatic step toward slowdown-tolerant RSMs: *1-slowdown-tolerant* RSMs that continue to provide normal performance despite the presence of 1 slow replica.

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 [29, 11, 3] or rely on the collaboration of multiple replicas [41, 37]. 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.

Copilot replication is the first 1-slowdown-tolerant consensus protocol. It avoids slowdowns using two distinguished replicas, the pilot and copilot. The two pilots do all stages of processing a client's command in parallel. This ensures all steps happen quickly even if one pilot is slow. Clients send commands to both pilots, and both pilots order, execute, and

reply to the client. This proactive redundancy protects against a slowdown but also makes it more challenging to preserve consistency and efficiency.

The key challenge for Copilot replication is making its ordering stage slowdown tolerant. To provide linearizability, it needs to ensure the pilots agree on the ordering of client commands, but that in turn would naively require each to wait on the other if it is slow. Copilot instead allows a pilot to *fast takeover* the ordering work of a slow pilot. It does so by persisting its takeover and subsequent ordering to the replicas.

Each pilot has a separate log where it orders client commands. Copilot combines the logs using dependencies, e.g., pilot log entry 9 is after copilot log entry 8. Copilot's ordering protocol has two phases—FastAccept, Accept—that commit commands to the pilots' logs along with their dependencies. In the FastAccept round, a pilot proposes an initial dependency for a log entry. If a sufficient number of replicas agree to this ordering, then this entry has committed on the *fast path* and the pilot moves on to execution. Otherwise—if the replicas have already agreed to a different ordering proposed by the other pilot—then the pilot adopts a dependency suggested by the replicas that it persists in the Accept round.

Copilot provides crash fault tolerance using similar mechanisms to Multi-Paxos [29, 38] that are applied independently to the log of each pilot. Copilot combines the logs of the two pilots using mechanisms inspired by EPaxos [41]. As such, it provides the same safety and liveness guarantees as Multi-Paxos and EPaxos. It is safe under any number of crash faults, and it is live as long as a majority of replicas can communicate in a timely manner. In addition, Copilot provides slowdown tolerance even if one replica is slow or failed.

The core Copilot protocol provides slowdown tolerance. However, it would naively go to the Accept round often as the two pilot's ordering commands continuously interleave and prevent one or both from taking the fast path. This additional round of messages would increase latency and decrease throughput relative to traditional consensus protocols like Multi-Paxos, which need only 1 round in the normal case.

Copilot replication includes two optimizations that keep it on the fast path almost all the time. When both pilots are fast, *ping-pong batching* coordinates them so that they alternate their proposals, allowing both pilots to commit on the fast path. When one pilot is slow, *null dependency elimination* allows the fast pilot to avoid waiting on commits from the slow pilot. With null dependency elimination, a fast pilot only needs to fast takeover the ordering work of the slow pilot that is in-progress when the slowdown begins.

Copilot replication is implemented in Go and our evaluation compares it to Multi-Paxos and EPaxos in a datacenter setting. When no replicas are slow, Copilot's performance is competitive with Multi-Paxos and EPaxos. When there is a slow replica, Copilot is the only consensus protocol that avoids high latencies for client requests.

Latent Copilot is another design and implementation of a 1-slowdown-tolerant consensus protocol. It is a variant of Copilot and introduces additional mechanisms. In Latent Copilot, a client also sends its commands to both pilots, but to reduce the overhead of having two pilots, only one of the pilots is in *active* mode and actively proposes commands. The other pilot operates in *latent* mode and only proposes commands when they have not been committed by the active pilot in a timely manner. In this way, Latent Copilot provides an intermediate tradeoff between Multi-Paxos and Copilot in terms of throughput and slowdown tolerance.

Latent Copilot has different mechanisms to determine when the pilots should switch their modes if it suspects the active pilot is constantly slow. To learn about the progress of the other pilot, a pilot uses additional metadata embedded in the ordering messages to learn about the status of the commands from the replicas. With this mechanism, the latent pilot will become active when it receives a sufficient number of signals indicating that the active pilot is potentially slow. Similarly, the active pilot will become latent when it gathers a sufficient number of signals indicating that the latent pilot is fast or the active pilot itself is potentially slow. The promoting and demoting mechanisms together help Latent Copilot

operate with one fast active pilot most of the time, while keeping the other (potentially slow) pilot in latent mode.

Our evaluation shows that Latent Copilot provides 1-slowdown-tolerance and that its performance with no slowdowns is comparable to Multi-Paxos.

As the next crucial step toward developing slowdown-tolerant RSMs, we also propose the design of an *s*-slowdown-tolerant consensus protocol that can tolerate more than one slowdown. Our *s*-slowdown-tolerant protocol generalizes the desing of Copilot replication including its ordering protocol, execution protocol, fast takeover procedure, and optimizations—ping-pong batching and null dependency elimination.

In summary, the contributions of this dissertation are as follows:

- Defining slowdown-tolerance and identifying why existing consensus protocols are not slowdown-tolerant (§2).
- Copilot replication, the first 1-slowdown-tolerant consensus protocol. Copilot replication uses two pilots to ensure the RSM stays fast, by using proactive redundancy in all stages of processing a client command (§3).
- Ping-pong batching and null dependency elimination, which make Copilot's performance with no slowdowns or one slowdown competitive with traditional protocols (§5).
- Latent Copilot replication, another design and implementation of a 1-slowdown-tolerant protocol. Latent Copilot achieves an intermediate tradeoff between Multi-Paxos and Copilot in terms of throughput and slowdown tolerance by operating with one active pilot, which actively propose commands, and one latent pilot, which proposes commands only when necessary (§6).
- The design of an *s*-slowdown-tolerant consensus protocol that extends Copilot's design to tolerate any $s$ slow replicas ($s \geq 1$) (§7).

# Chapter 2

# Slowdown Tolerance

This section explains RSMs, defines slowdown tolerance, and explains why existing protocols do not tolerate slowdowns.

## 2.1 Replicated State Machine Primer

RSMs are linearizable, fault-tolerant groups of machines. They implement a state machine that atomically applies deterministic commands to stored state and returns any output [48]. The machines within an RSM are *replicas*. The RSM provides fault tolerance by starting the replicas in the same initial state and then moving them through the same sequence of states by executing commands in the same order. Then, if one of the replicas fails, the remaining replicas still have the state and can continue providing the service.

RSMs provide linearizability for client commands. *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* [22].

RSMs are coordinated by *consensus protocols* that determine a consistent order of client commands that are then applied across the replicas. An RSM goes through four stages to process a client command: it receives the command, it orders the command using the

consensus protocol, it executes the command, and it replies to the client with any output. Each replica executes commands in the agreed-upon order. A common way to implement and think about RSMs is that they agree to put commands in sequentially increasing log entries, and then execute them in that log order.

## 2.2 Defining Slowdown Tolerance

We define a slow replica, clarify the relationship between slow and failed, and then define *s*-slowdown-tolerance.

**Defining a slow replica.** We reason about the speed of a replica based on the time it takes between when the machine receives a request over the network and sends a response back out over the network. This includes the replica's RSM processing and its host-side network processing. It does not include the time it takes messages to traverse network links.

We say a replica is *slow* when its responses to messages take more than a threshold time $t$ over its normal response time. For example, if a replica typically replies to messages within 1 ms, and we consider a slowdown threshold of $t = 10$ ms, then a replica is slow if it takes more than 11 ms to send responses. The precise setting of $t$ will depend on the scenario and may even vary over time. For example, if an OS upgrade increases the processing speed of all replicas, then what was considered normal performance in the past may now be considered slow. We assume the term "slow" reflects the current definition and build our notion of slowdown tolerance on top of this term—that is, our notion of slowdown tolerance is robust to changes in what is considered slow.

**Failed versus slow replicas.** Replicas that have failed are also slow because they will not reply to messages within the slowdown threshold time. Thus, all failed replicas are slow. However, not all slow replicas are failed. Replicas can be slow but not failed for many reasons, e.g., misconfigurations, host-side network problems, or garbage collection

events. It is these slow-but-not-failed replicas that we care about most because existing fault-tolerance mechanisms do not necessarily tolerate them.

**Defining s-slowdown-tolerance.** Traditionally, clients use RSMs because they provide a service that does not fail despite $f$ replicas failing. Our definition of slowdown tolerance mirrors this traditional definition of fault tolerance while accounting for the dynamic nature of what is considered "slow." An RSM is *s-slowdown-tolerant* if it provides a service that is not slow despite $s$ replicas being slow. 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—i.e., its response time properties based on the current definition of "slow"—and let $T'$ 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.

In this dissertation, we focus on the practical case of 1-slowdown-tolerance. Designing RSMs that are $s$-slowdown-tolerant for $s > 1$ is discussed in Chapter 7. Implementing and evaluating an $s$-slowdown-tolerant protocol is an interesting avenue of future work.

## 2.3 Why Existing Protocols Slowdown

We explain why existing protocols are not slowdown tolerant using Multi-Paxos, EPaxos, and Aardvark as examples.

**Multi-Paxos.** Multi-Paxos [29, 30, 38, 27] is the canonical consensus protocol. It uses the replicas to elect a *leader*. The leader receives client commands and orders them by assigning them to the next available position in its log. It persists that order by sending Accept messages to the replicas and waiting for a majority quorum (including itself) to

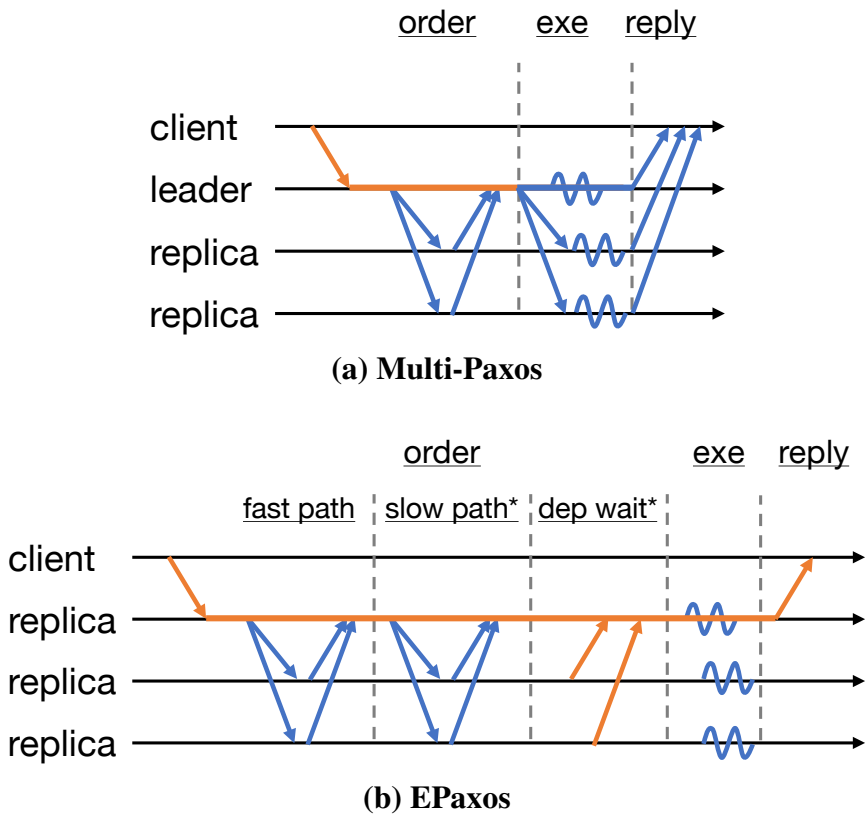



**Figure 2.1: Message diagrams with execution for Multi-Paxos (a) and EPaxos (b). Orange components indicate parts of each protocol that are not slowdown tolerant because they lack redundancy. Blue components indicate parts with redundancy. EPaxos ordering phases that are sometimes necessary are marked with asterisks (*).**

reply, which commits the command in that log position. It notifies other replicas of the commit using a Commit message. The replicas execute commands in the accepted prefix of the log in order, i.e., they only execute a command once its log position is committed and all previous log positions have been executed. After executing the command, the replicas reply to the client with any output. (We describe a variant of Multi-Paxos that has all replicas reply to the client, similar to PBFT [11], because it provides more redundancy.)

Figure 2.1a shows these steps and identifies parts of the protocol that are not slowdown tolerant. Receiving the client's command and running the ordering protocol are not slowdown tolerant because they are only done by the leader. If the leader is slow, it slows these stages. In turn, this is evident to clients whose commands see much higher latency.

Several parts of Multi-Paxos are individually slowdown tolerant—notably, the Accept messages sent to the replicas to persist the leader's ordering of a command. These messages are sent to all replicas with the leader only needing to hear back from a majority (including itself). For instance, with 5 replicas the leader sends the messages to the 4 other replicas and can proceed once it hears back from 2. This makes Multi-Paxos resilient to a non-leader replica being slow.

**EPaxos.** EPaxos [41] avoids the single leader of Multi-Paxos with a more egalitarian approach that distributes the work of receiving, ordering, executing, and replying across all replicas. Each replica in EPaxos receives commands from a subset of clients and runs the ordering protocol. We call this specific replica the command's *designated replica*. EPaxos's ordering protocol uses fine-grained dependencies between commands to dynamically determine an ordering using FastAccept and SlowAccept phases. Once a replica knows the dependencies of its commands, it waits for the final dependencies of its dependencies to arrive in the DependencyWait phase. Then a replica totally orders the commands and executes them in the resulting order. When a replica executes a command for which it is the designated replica, it sends the reply to the client. EPaxos can sometimes avoid the SlowAccept and DependencyWait phases.

Figure 2.1b shows these steps and identifies the parts of the protocol that are not slowdown tolerant. Receiving the client's command, running the ordering protocol, and replying to the client are all not slowdown tolerant because they are only done by a command's designated replica. If the designated replica is slow, it will slow down all of these stages, and thus the RSM, for its subset of clients.

DependencyWait can lead to slowdowns for all clients if any replica is slow. This is because DependencyWait requires a replica to wait until it learns the dependencies of the dependencies of a command. These transitive dependencies are necessary for EPaxos to consistently order commands at different replicas. But they are only determined and then

sent from a command's designated replica. Thus, a slow replica will be slow to finalize and send out the dependencies for its designated commands to other replicas. This in turn slows commands that acquire dependencies on commands ordered by the slow replica, in addition to commands that use the slow replica as their designated replica.

**Leader election.** Consensus protocols with leaders include a leader election sub-protocol that provides fault tolerance in case a leader fails. In this sub-protocol, replicas detect when they think a leader may have failed, elect a new leader, ensure that the new leader's log includes all the commands that have been accepted by a majority quorum, and then have the new leader start processing new commands.

Some protocols, like Aardvark [3] and SDPaxos [55], have proposed using leader election to mitigate slowdowns as well, by having replicas detect when they think a leader is slow and then trigger the leader election sub-protocol. Unfortunately, this approach does not provide slowdown tolerance for two reasons. First, leader election is a heavy-weight process that makes an RSM unavailable while it is ongoing: no new commands can be processed until a new leader is elected and brought up to date. Second, leader election is only triggered when a replica thinks the leader is slow (or failed). Thus, only the subset of slowdowns detected by the replicas will be mitigated, and only after they have been detected. In contrast, 1-slowdown-tolerance requires an RSM to deliver performance as if the slowdown did not exist.

Consider the case of Aardvark. Aardvark employs two mechanisms to detect slowdowns in the leader: the first enforces a gradually increasing lower bound on the leader's throughput based on past peak performance; the second starts a heartbeat timer between each batch to ensure the leader is proposing new batches quickly enough. If the leader's throughput drops below the lower bound or if the heartbeat timeout expires, Aardvark initiates a view change to rotate the leader among the replicas. These mechanisms provide only partial slowdown tolerance because each limits the effects of only the subset of slow-

downs it detects. For example, they do not protect against a replica whose processing path is slow for client requests but fast for replicas; or a replica whose responses become gradually slower over time while maintaining a small gap between successive responses. Such replicas would still be able to slow down the RSM during their turn as leader.

Further, using view changes to react to slowdowns can itself cause slowdowns and become costly. In practice, leader election timeouts are generally on the order of hundreds [45, 44] or thousands [9, 15, 18] of milliseconds to prevent the excess load, unavailability, and instability that occurs when leader elections are easily triggered. Thus, any leader slowdown whose severity is less than these timeouts will go undetected, as will any slowdown that is not covered by the detection mechanisms.

## 2.4 Summary and Insights

The fundamental problem with existing protocols is that they are *detection based*. Detection-based approaches do not protect against slowdowns until they are detected and never protect against slowdowns that are not detected. As a result, a consensus protocol cannot be 1-slowdown-tolerant if the path of a client's command includes at least one point where it goes through a single replica. If that replica is slow, the RSM will be slow (until and if the slowdown is detected). Thus, to design a 1-slowdown-tolerant replication protocol, we must *proactively* ensure there are at least 2 disjoint paths that a client's command can take at every stage. If one of these paths gets stuck at a slow replica, the other path can continue because we assume only 1 replica becomes slow.

# Chapter 3

# Design

The core idea behind Copilot [42] is to use two distinguished replicas, the pilot ($P$) and the copilot ($P'$), to redundantly process every client command. Figure 3.1 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.

This section describes the basic design of Copilot, and Section 5 describes optimizations that complete its design. This section first defines our model and then details each major part of the protocol—ordering, execution, and fast takeovers. Finally, it covers additional design details and summarizes why Copilot provides 1-slowdown-tolerance.

## 3.1 Model

Copilot assumes the *crash failure model*: a failed process stops executing and stops responding to messages. Copilot assumes an *asynchronous system*: there is no bound on the relative speed at which processes execute instructions, and there is no bound on the time it takes to deliver a message. Copilot requires $2f+1$ replicas to tolerate at most $f$ failures, and guarantees linearizability as a correctness condition despite any number of failures. Copilot provides 1-slowdown-tolerance in the presence of any one slow replica.

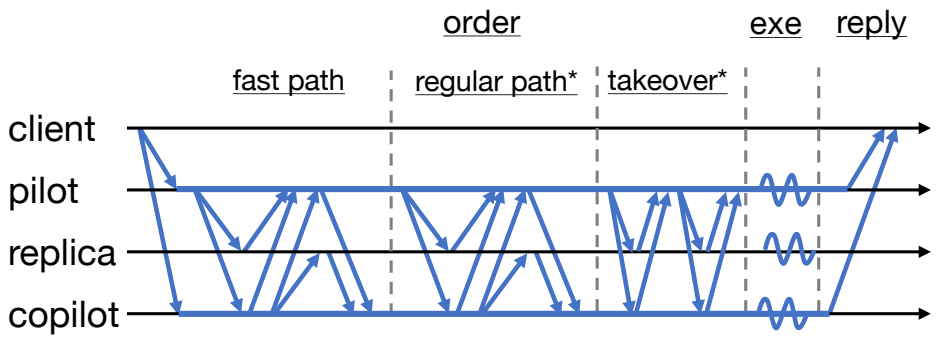



**Figure 3.1: Message diagram with execution for Copilot. All components are in blue because all have the necessary redundancy to avoid any slow replica. Phases that are only sometimes necessary are marked with asterisks (*). The takeover phase only executes when it is necessary to prevent one pilot from waiting too long on the other pilot. Copilot's optimizations (§5) keep it on the fast path when both pilots are fast and mostly avoid the need for fast takeovers when one pilot is slow.**

## 3.2 Ordering

Copilot's ordering protocol places client commands into the *pilot log* and the *copilot log*, which are coordinated by the pilot and copilot, 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. Pilots propose *initial dependencies* for log entries. Replicas either agree to that ordering or reply with a *suggested dependency*. Ultimately, each entry has a *final dependency* that is used by the execution protocol. The final dependencies between the pilot and copilot log may form cycles. Copilot's execution protocol constructs a single *combined log* using the final dependencies between the pilot and copilot logs and a priority rule that orders pilot entries in a cycle ahead of copilot entries. Figure 3.2 shows an example of how dependencies are used to order the entries in the combined log.

Copilot's ordering protocol persists the command and final dependency for a log entry to the replicas to ensure they can be recovered if up to $f$ replicas (including both pilots) fail. The ordering protocol always includes a FastAccept phase and sometimes includes an Accept phase. The protocol completes after the FastAccept phase if enough of the replicas have agreed with the initial dependency to ensure it will always be recovered as the final

dependency. Otherwise, the pilot selects a suggested dependency that orders an entry after enough of the other pilot's log to ensure linearizability.

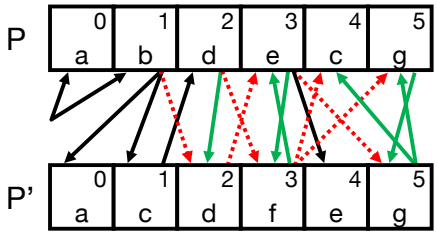
The remainder of this subsection follows the ordering protocol in order, starting with the client sending a command to the replicas. Our description assumes no fast takeovers (§3.4) or view-changes (§3.5) for simplicity; with fast takeovers and view-changes, replicas reject messages when entries are taken over by another pilot, and entries can be committed with a no-op as a command.

**Clients submit commands to both pilots.** Each client has a unique client ID *cliid*. Clients assign commands a unique, increasing command ID *cid*. Clients send each command, its client ID, and its command ID to both pilots. The $\langle cliid,\ cid \rangle$ tuple uniquely identifies commands and enables the replicas to deduplicate them during execution.

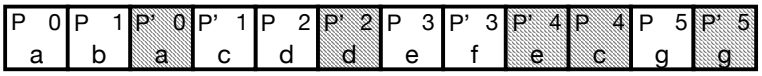
**Pilots propose commands and an initial dependency.** Upon receiving a command from a client, a pilot puts the command into its next available log entry. It also assigns the *initial dependency* for this entry, which is the most recent entry from the other pilot it has seen. It then proposes this assignment of command and initial dependency for this entry to the other replicas by sending them FastAccept messages.

**Replicas reply to FastAccepts.** When a replica receives a FastAccept message it checks if the initial dependency for this entry is compatible with all previously accepted dependencies. If it is, the replica fast accepts the initial dependency. If it is not, the replica rejects the initial dependency and replies with a new suggested dependency.

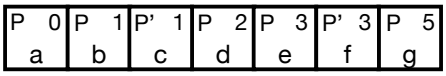
A pair of dependencies are *compatible* if at least one orders its entry after the other. Figure 3.2a shows examples of compatible and incompatible dependencies. $P'.1$ with dependency $P.1$, and $P.2$ with dependency $P'.1$ are compatible because $P.2$ is ordered after $P'.1$. $P'.3$ with dependency $P.2$ and $P.3$ with dependency $P'.2$ are *incompatible* because neither is ordered after the other. Incompatible dependencies must be avoided because they




**(a) Dependencies join the two logs.**




**(b) Combined log with duplicates.**




**(c) Execution order from the combined log.**

**Figure 3.2: 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.**

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 $P.3$.

A replica uses the compatibility check to determine if an initial dependency, $P.i$ with dependency $P'.j$, is compatible with all previously accepted dependencies. $P.i$ is ordered after all previous entries in the $P$ log automatically and after all entries $P'.j$ or earlier by its dependency. Thus, the check only needs to look at later entries in the other pilot's log. The *compatibility check* passes unless the replica has 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 $< P.i$.

If it has not accepted a later entry, then this same check will prevent the replica from fast accepting any incompatible dependencies from the other pilot in the future. If it has accepted a later entry, but that entry's dependency is on $P.i$ or a later entry, then that entry, call it $P'.k$, is ordered after this one, i.e., $P'.j, P.i, \ldots, P'.k$. Thus, in either of these cases the replica fast accepts the initial dependency and replies with a FastAcceptOk message to the pilot. Otherwise, it sends a FastAcceptReply message to the pilot with its latest entry for the other pilot, $P'.k$, as its *suggested dependency*.

**Pilots try to commit on the fast path.** A pilot tries to gather a *fast quorum* of $f + \lfloor \frac{f+1}{2} \rfloor$ FastAcceptOk replies (including from itself).[1] If a pilot gathers a fast quorum, then enough replicas have agreed to its initial dependency that it will always be recovered from any majority quorum of replicas. Thus, it is safe for the pilot to commit this entry on the *fast path* and continue to execution. The entry's initial dependency is now its *final dependency* that is used during execution. The pilot also sends a Commit message to the other replicas to inform them of the final dependency for this entry. (It does not wait for responses for the Commit messages.)

A pilot might be unable to gather a fast quorum of FastAcceptOks for two reasons. First, it might receive FastAcceptReplys because replicas rejected the initial dependency as incompatible. Second, it might only receive as few as $f+1$ replies instead of the necessary $f + \lfloor \frac{f+1}{2} \rfloor$ because up to $f$ of the $2f+1$ replicas have failed. In either case, the pilot waits until it receives at least $f+1$ FastAcceptOks and FastAcceptReplys and then continues to the Accept phase.

**Pilots persist the final dependency in the Accept phase.** A pilot selects the final dependency 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

[1] This size is 2/3, 3/5, 5/7, and 6/9 for common RSM sizes.

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.

Then the pilot persists this final dependency by sending it in an Accept message to all the other replicas. The ordering determined by final dependencies in Accept messages can create cycles at replicas. These cycles are acceptable because replicas will learn about them and then execute the commands in the cycles in the same order using the execution protocol. Thus, the other replicas accept this final dependency and reply with AcceptOk messages. When the pilot receives $f+1$ AcceptOks (including from itself) it has committed the entry on the *regular path*. It then sends Commit messages to the other replicas and proceeds to execution.

## 3.3 Execution

Replicas execute commands in the combined log order. The combined log contains each client command twice. A replica only executes a command in its first position in the combined order. After executing a command, the pilot and copilot reply to the command's client. Figure 3.2 shows an example of a combined log and its executed subset.

**Copilot's total order of commands.** 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. There are three rules that define the total order. (1) The total order includes the partial order of each pilot's log, e.g., $P.0 < P.1 < P.2$ in Figure 3.2a. The dependencies between the logs sometimes create cycles. (2) When the dependencies are acyclic, the total

order follows the dependency order, e.g., $P.1 < P'.0 < P'.1 < P.2$ in Figure 3.2a. (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 3.2a.

**Executing in order.** Replicas learn the final dependencies for each entry and thus use the same total order. A replica executes a command once its entry is committed and all preceding entries in the total order have been executed. The following rules determine when it is safe for a replica to execute a command in entry $P.i$ with dependency $P'.j$: (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. The rules 1–3 correspond to the rules that define the total order above.

Replicas can learn of committed entries out of order, e.g., a pilot can learn that their entries have committed before they learn of the commits for their dependencies. To ensure commands are executed in the total order, a replica must wait for the commit of all potentially preceding entries. For example, an entry in the pilot log $P.i$ must wait for the commit of all entries $< i$ in the pilot log, the commit of its dependency $P'.j$ in the copilot log, and the commit of all entries $< j$ in the copilot log. Copilot's fast takeover protocol ensures a fast pilot need not wait long (§3.4) before executing this entry.

**Deduplicating execution and replying.** In the absence of failures, each command will be in the combined log twice. A replica executes each command only once in its first position. It tracks the commands from each client that have already been executed using the $\langle cliid,\ cid \rangle$ tuple. The first time it sees a command, it executes it. If the replica is the current pilot or copilot, it replies to the client with any output. The second time it sees a command, it simply marks it as executed and moves on. A client thus receives a response from each pilot for each command; it ignores the second response.

## 3.4 Fast Takeover

To execute commands in the total order determined by the ordering protocol, a pilot sometimes waits on commits from the other pilot. Waiting on the other pilot for a long time would not be slowdown tolerant. Copilot's fast takeover mechanism avoids a fast pilot waiting too long for a slow pilot by completing the necessary ordering work for that 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 [30]. These ballot numbers allow a fast pilot to safely takeover the work of a slow pilot using Paxos's two phases of prepare and accept. When a replica is elected as either pilot or copilot, that sets a ballot number for all entries in the corresponding log to be $b$. Replicas only (fast) accept entries if the included ballot number is $\geq$ the ballot number set for that entry. When a pilot is not slow, its included ballot numbers are exactly those set for each entry, and the protocol proceeds as described above.

When a pilot is slow, the other pilot can safely takeover its work by setting higher ballot numbers on the relevant entries in the slow pilot's log. The fast pilot does this by sending Prepare messages with a higher ballot number $b'$ for the entry to all replicas. If $b'$ is higher than the set ballot number for that entry, the replicas reply with PrepareOk messages and update their prepared ballot number for that entry. The PrepareOk messages indicate the progress of an entry at a replica, which is one of: not-accepted, fast-accepted, accepted, or committed. The PrepareOk messages include the highest ballot number for which a replica has fast or regular accepted an entry, the command and dependency associated with that entry, and an 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 of the PrepareOk messages indicate 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.

**Recovery value picking procedure.** We use *value* to indicate the command and dependency for a log entry. The fast takeover mechanism and view-change mechanism use the recovery value picking procedure to correctly recover a command and dependency for any entry that could have been committed and thus executed. This ensures all replicas execute all commands in the same combined log order.

The recovery value picking procedure is complex and its full details are described in Appendix A.2. The procedure examines 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. In turn, this ensures that any incompatible entries from the other pilot's log will be ordered after this entry. Thus, it is 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. Examining these entries leads to three subcases:

- If each of these entries either commits with a no-op or has a dependency $\geq$ the recovered entry, the procedure picks the initial value proposed by the failed pilot. It is safe to commit the initial value in the recovered entry because it is compatible with the entries from the other pilot's log.
- If at least one of these entries commits with a dependency $<$ the recovered entry, the procedure picks a no-op. It is safe to commit a no-op in the recovered entry because the failed pilot could not have committed its proposed value on the fast path before it failed in this case.
- If at least one of these entries is not yet committed (e.g., due to the other pilot failing before sending out the Commit messages), the value picking procedure needs to recover that entry in order to determine the correct value for the recovered entry. It does so by simultaneously preparing these two entries and then recovering them using the same quorum of replicas that reply to the SimultaneousPrepare messages. (SimultaneousPrepare is similar to Prepare except that it prepares two different entries in a single message.)

**Triggering a fast takeover.** A pilot sets a takeover-timeout when it has a committed command but does not know the final dependencies of all potentially preceding entries, i.e., it has not seen a commit for this entry's final dependency. If the takeover-timeout fires, the

pilot stops waiting and does the necessary ordering work itself. It starts the fast takeover of all entries in the slow pilot's log that potentially precede this entry. Our implementation does this in a parallel batch for all entries. Setting the takeover-timeout too low could result in spurious fast takeovers that could lead to dueling proposers. We avoid dueling proposers using the standard technique of randomized exponential backoff. We avoid spurious fast takeovers by setting a medium takeover-timeout in our implementation (10 ms). This medium timeout is fine because null dependency elimination (§5.2) avoids needing to wait when a pilot is continually slow.

Fast takeovers have a superficial resemblance to leader elections because both are triggered by one replica timing out while waiting to hear from another replica. Leader elections are triggered when one replica does not hear *something* from another replica—e.g., a heartbeat or a new proposal. But a leader can still send something regularly and/or quickly while being slow in other ways (§2.3). Fast takeovers, on the other hand, are triggered when one pilot is waiting to execute a specific client command. This puts them on the processing path of every request. When combined with the proactive redundancy of having both pilots process each client command, this bounds the latency of client commands to that of the faster pilot. If one pilot is slow, the other will process any given command up until execution and then, if necessary, wait for the takeover-timeout before completing the specific ordering work of the other pilot needed to unblock execution.

## 3.5 Additional Design

The additional parts of Copilot's design not described in this section all are similar to normal RSM designs. At-most-once semantics for client requests are handled using $\langle$*cliid, cid*$\rangle$ tuples and caching the output associated with a command. Non-deterministic commands can be handled by having pilots make the commands deterministic by doing the non-deterministic work (e.g., selecting a random number) and including it as input to the

command. There will be two different non-deterministic versions of the command in the combined total order, but deduplication will ensure only the first is executed. State used for deduplication is garbage collected once a command is encountered in the log a second time.

Pilot and copilot election uses *view-changes*, analogous to Multi-Paxos's leader election [38], on the pilot and copilot logs, respectively. The view-change process has a newly elected pilot or copilot use the recovery value picking procedure described above while committing all unresolved entries in the log. The two separate logs of the pilots allow Copilot to elect a new pilot to replace a failed one while the other pilot continues to order and commit commands in its own log. While this is happening, the active pilot will acquire no new dependencies. Thus, the active pilot will be able to commit on the fast path and execute commands without waiting on any entries in the other log while a new pilot is elected. The full details of Copilot's view-change protocol are described in Appendix A.3.

## 3.6 Why Copilot is 1-Slowdown-Tolerant

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 other (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. Thus, the fast pilot is able to continue executing its own entries. Shortly after a slowdown,

the fast pilot stops acquiring dependencies on uncommitted entries (or acquires only null dependencies (§5.2)), eliminating the need for any fast takeovers. Thus, the performance of the RSM reduces to that of the faster pilot, satisfying 1-slowdown-tolerance.

# Chapter 4

# Correctness

We prove that Copilot replication is both safe, i.e., it provides linearizability (4.1), and live, i.e., all client commands eventually complete (4.2). The full proofs can be found in Appendix B; we summarize the intuition for each proof below.

## 4.1 Safety

To prove linearizability, 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 operations, i.e., if command $a$ completes in real-time before command $b$ begins, then $a$ must be ordered before $b$.

Let $P$ and $P'$ represent the two pilots. To prove the real-time ordering property, consider a command $a$ that completes before a command $b$ begins. Since $a$ completes, it must be committed in at least one pilot's log; suppose w.l.o.g. it commits in $P$'s log at entry $P.i$. Within $P$'s log, $a$ is trivially ordered before $b$, because $b$ is issued only after $a$ has been committed. In $P'$'s log, $a$ and $b$ may commit in either order, but the key observation is that $b$'s entry, call it $P'.j$, cannot have a dependency that precedes $P.i$, because this would be deemed incompatible during the FastAccept phase (cf. §3.2). Since $P'.j$'s dependency is

$\geq P.i$ and $P.i$'s dependency is $< P'.j$, there are no cycles between $P.i$ and $P'.j$. Thus, $P.i$ is executed before $P'.j$, which implies that $a$ is executed before $b$.

To prove the total ordering property, we first prove the following invariant: if two log entries $P.i$ and $P'.j$ commit at different pilots, either $P.i$ has a dependency $\geq P'.j$ or $P'.j$ has a dependency $\geq P.i$. This ensures that a dependency path exists from one entry to the other, preventing them from being ordered differently at different replicas. We then show that each entry in a pilot's log commits with the same commands and dependency across all replicas, even in the presence of failures (including failures of both pilots). This relies on the recovery value picking procedure from §3.4. When an entry commits on either the fast path or regular path, it is persisted to at least a majority of replicas. During a fast takeover or view change—which occur when one or both pilots are slow (or failed)—the prepare phase will see the entry due to majority quorum intersection, and will reuse it when committing. If the replies from the prepare phase do not show a committed entry, then we must look at them more carefully. If any reply shows the entry is accepted, or if $\geq f$ replies show it is fast-accepted, then we commit the entry with its accepted dependency because it might have committed. If $< \lfloor \frac{f+1}{2} \rfloor$ replies show it is fast-accepted, then we can safely commit a no-op because the entry did not have enough fast accepts to commit. The final case occurs when the number of replies that show fast-accepted is in the range $[\lfloor \frac{f+1}{2} \rfloor, f)$. In this case, the entry may or may not have committed, depending on whether there was an incompatible entry in the other pilot's log. The recovery value picking procedure resolves this by examining and, if needed, recovering the possible incompatible entries in the other pilot's log. Note that this procedure does not rely on replies from either pilot, and instead reasons about any $f+1$ possible replies received during the prepare phase.

Since each pilot's log is consistent across a majority of the replicas, the entries and their dependencies are also consistent, so the commands are executed in the same total order.

## 4.2 Liveness

To prove liveness, we must show that a command issued by a client eventually receives a response. Due to FLP [19], we assume the system is eventually partially-synchronous [17] and that all messages are eventually delivered.

Our proof uses a double induction. Assume a replica has executed all entries in $P$'s log up to $P.i$ and all entries in $P'$'s log up to $P'.k$. We show that the replica eventually executes either $P.(i+1)$ or $P'.(k+1)$, or a fast takeover occurs, or a view change occurs. Consider the failure-free case first.

If the dependency of $P.(i+1)$ is null or points to an entry $P'.j \leq P'.k$, then $P.(i+1)$ can be executed immediately. If $P'.j > P'.k$ (i.e., $P'.j$ has not been executed), then Copilot checks if a cycle exists between $P.(i+1)$ and $P'.j$. If no cycle exists, then execution switches to the next entry in $P'$'s log, $P'.(k+1)$. $P'.(k+1)$ can be executed because its dependency must be $\leq P_i$ (otherwise there would have been a cycle between $P.(i+1)$ and $P'.j$), which by our inductive assumption has been executed.

If a cycle exists between $P.(i+1)$ and $P'.j$ and $P$ has higher priority, Copilot checks if $P.(i+1)$ can be executed before $P'.(k+1)$, which is the next entry in $P'$'s log that has not been executed. If a cycle also exists between $P.(i+1)$ and $P'.(k+1)$, Copilot breaks the cycle in favor of $P$ and executes $P'.(i+1)$. If no cycle exists between $P.(i+1)$ and $P'.(k+1)$, execution switches to the next entry in $P'$'s log, $P'.(k+1)$. $P'.(k+1)$ can be executed immediately because its dependency must be $\leq P.i$ (otherwise there would have been a cycle between $P.(i+1)$ and $P'.(k+1)$), which by our assumption has been executed.

If a cycle exists between $P.(i+1)$ and $P'.j$ and $P'$ has higher priority, execution switches to $P'$'s log. Entry $P'.(k+1)$ can execute immediately if its dependency is $\leq P.i$ (by our inductive assumption), or after Copilot breaks the cycle in favor of $P'$. In all cases, either $P.(i+1)$ or $P'.(k+1)$ is executed.

Now consider the case of failures. If only non-pilots fail, this reduces to the failure-free case. If $P'$ is slow/failed, then $P.(i+1)$ may not be able to execute because its dependency

$P'.j$ and/or some entries $< P'.j$ in $P'$'s log may not have committed. In this case, $P$ eventually does fast takeovers of the entries $\leq P'.j$ in $P'$'s log that have not been committed. If both pilots are slow/failed, then neither $P.(i+1)$ nor $P'.(k+1)$ may be able to execute. In this case, a replica eventually initiates a view change to elect new pilots. Fast takeovers and view changes cannot repeat indefinitely by the same argument that basic Paxos and Multi-Paxos use to ensure progress, by relying on partial synchrony.

# Chapter 5

# Optimizations

This section covers ping-pong batching and null dependency elimination, which improve Copilot's performance. Ping-pong batching coordinates the pilots so they propose compatible orderings when both are fast. Null dependency elimination allows a fast pilot to safely avoid waiting on commits from a slow pilot. Copilot includes both optimizations.

## 5.1 Ping-Pong Batching

Ping-pong batching coordinates the pilots so they propose compatible orderings to the replicas. The replicas fast accept these compatible orderings and thus the pilots commit on the fast path. With ping-pong batching, each pilot accumulates a batch of client commands. It assigns each command to its next available entry, so each batch is a growing assignment of client commands to consecutive entries. A pilot closes a batch and tries to FastAccept the batch when either it receives a FastAccept message from the other pilot or its ping-pong-wait timeout fires.

When both pilots are fast, they will close batches when they receive a FastAccept from the other pilot. This causes FastAccepts to ping-pong back and forth between the two pilots. The pilot closes its first batch and sends out its FastAccepts. When the copilot receives that

FastAccept, it closes its first batch and sends out its FastAccepts. When the pilot receives that FastAccept, it closes its second batch, and so on.

This ping-ponging ensures that the pilots agree on the ordering of their entries. Before a pilot sends out a batch it hears about the latest batch from the copilot; and the copilot will not send out another batch until it hears about this batch from the pilot. Because the pilots agree on the ordering of their entries, the replicas can always fast accept their proposed orderings. If the replicas receive the proposed orderings in the same order that the pilots ping-pong propose them, then they agree to this ordering. Even when replicas receive the proposed ordering in a different order, they can still accept them because the dependencies will be compatible.

If one pilot is slow, the other will close its batches when the ping-pong-wait timeout fires. This timeout helps provide slowdown tolerance: even if one pilot is slow, the other need not wait on it for long.

## 5.2 Null Dependency Elimination

Null dependency elimination allows a fast pilot to avoid waiting on commits from a slow pilot. It looks inside a dependency to see the command it contains. If the contained command has already been executed, then execution deduplication (§3.3) will avoid executing it. We call these *null dependencies* because their execution will have no effect.

Sometimes a pilot must wait on the commit of the other pilot's earlier entries because it needs to know the finalized dependency of that entry to know the agreed-upon total order. This is unnecessary for null dependencies because they are not executed. Thus, their final ordering information is irrelevant: a pilot need not determine when to execute them because it will not execute them. Instead, the pilot marks the null dependency as executed and continues.

When there is a continually slow pilot, null dependency elimination allows the fast pilot to avoid fast takeovers. A continually slow pilot will propose entries with a given command $c$ after the fast pilot has already proposed an entry with that command $c$. Thus, the continually slow pilot's entries will be null dependencies for the fast pilot that can be safely skipped. This allows the fast pilot to never wait on commits from the slow pilot and thus avoids needing to fast takeover its entries. Fast takeovers are still necessary, however, for the cases when a pilot becomes slow *after* it proposes its ordering. Thus, when a pilot becomes slow, the other pilot does a fast takeover of the slow pilot's ongoing entries to provide 1-slowdown-tolerance. Thereafter, the fast pilot uses null dependency elimination to provide 1-slowdown-tolerance.

Naively nullifying a dependency entry by simply checking if its commands have been executed, however, may violate the total order property of linearizability and thus compromise the safety of Copilot. For example, if the commands that are nullified are not the same as the commands that are eventually committed in the dependency entry, different replicas may end up executing different sequences of commands, and thus violate the correctness of the RSM. This scenario may happen if the pilot $P'$ fails shortly after proposing its entry $P'.j$ and only a minority of replicas have received the proposed commands for $P'.j$ from the failed pilot. When a new pilot $P'$ is elected, it may propose new commands in $P'.j$ because no replicas that participate in the pilot election know about the empty log entry $P'.j$. (The minority of replicas that receive the commands from the failed pilot may have failed and do not participate in the view change.) If any of the replicas in the minority has nullified the commands in $P'.j$ when it executes an entry, it leads to inconsistent states of a RSM by executing an incorrect ordering of the commands.

To safely nullify a dependency entry, a replica performs two additional checks before it can actually nullify the commands inside the dependency. First, it checks if the *null-dep-safe* flag—an additional field about the dependency maintained by each log entry—is true. Second, it checks if its view about the dependency's pilot is sufficiently up-to-date. A pilot

is in charge of determining when it is safe to consider nullifying the dependencies that the entries in its log have on the other pilot's log. It does so by updating an entry's null-dep-safe flag during its ordering and committing of that entry. For a committed entry *P.i* with a dependency $P'.j$, $P'.j$ is safe to be considered for nullifying—i.e., *P.i*'s null-dep-safe flag is true—if at least a majority of replicas that have the same (current) view about $P'$ have advanced the latest entry in their $P'$'s log to be $\geq P'.j$. This ensures that if the pilot $P'$ fails and a new pilot is elected, the new pilot will propose its commands in the log entries after the highest latest entry from a majority, which is after $P'.j$. Hence, the new pilot will not propose any new commands in $P'.j$; and the fast takeover procedure can only ever commit the commands that have been proposed in $P'.j$ or a no-op in $P'.j$. This condition prevents the scenario where the commands that are eventually committed in $P'.j$ are different from the commands that are unsafely nullified, which leads to violating the total order property of linearizability and compromising Copilot's safety. Lemmas 7 and 8 in Appendix B.1 show that Copilot with the null dependency elimination optimization is safe and preserves the total order of commands.

**Updating the null-dep-safe flag.** Each replica maintains two view states, $P$-view and $P'$-view, one for each pilot role (pilot $P$ and copilot $P'$). Each view state includes the fields such as view ID and replica ID of a pilot, etc. (For the same pilot role, each view ID uniquely identifies a view.). Copilot updates the null-dep-safe flag of an entry during the ordering phase. When the pilot $P$ sends a FastAccept or Accept message for an entry *P.i* that includes a dependency $P'.j$, it also includes a *dep-view-ID* which is the view ID of its current $P'$-view. When a replica receives this message from the pilot $P$, it sets the *dep-seen* flag to true if it has the same $P'$-view (i.e., the view ID of its $P'$-view matches the dep-view-ID) and its latest entry in $P'$'s log is $\geq P'.j$. (In the case a replica has the same $P'$-view but its latest entry in $P'$'s log is $< P'.j$, if it is not participating in any view-change for $P'$-view, it can also set the dep-seen flag to true after updating the latest entry that it is

aware of in $P'$'s log to be $P'.j$.) A replica then includes the dep-seen flag in its reply to the FastAccept/Accept message. Upon receiving the replies from the replicas, the pilot $P$ will set the null-dep-safe flag to true if at least a majority of replicas (including itself) have set their dep-seen flag to true.

The pilot $P$ also includes the null-dep-safe flag and the dep-view-ID when it sends the Commit messages for the entry $P.i$ to other replicas. A replica records the additional metadata about the dependency $P'.j$, null-dep-safe and dep-view-ID, in the log entry $P.i$ when it receives the Commit message for $P.i$. When a replica tries to execute a committed entry $P.i$ with a dependency $P'.j$, it can safely nullify $P'.j$ if (1) $P.i$'s null-dep-safe flag is true, (2) the view ID of its current $P'$-view is $\geq$ dep-view-ID, and (3) the commands that $P'.j$ contains have been committed.

# Chapter 6

# Latent Copilot

## 6.1 Introduction

Copilot provides 1-slowdown-tolerance via proactive redundancy by having clients send their commands to both pilots and both pilots actively propose them. This results in Copilot incurring some throughput overhead in the normal case where there are no slowdowns: its peak throughput is about 8% lower than Multi-Paxos. In the normal case, the redundant work of a pilot actively proposing the same commands that have been proposed by the other pilot would become unnecessary if the other pilot commits them quickly. The redundant work, however, is necessary for providing 1-slowdown-tolerance when a slowdown occurs.

We introduce Latent Copilot, another design and implementation of a 1-slowdown-tolerant protocol. Latent Copilot achieves an intermediate tradeoff between Multi-Paxos and Copilot in terms of throughput and slowdown tolerance. Latent Copilot trades off some proactiveness for better performance (e.g., throughput) by avoiding proposing the commands that have been committed and proposing only when the commands have not been committed for some time (or when it suspects the other pilot is slow).

Like Copilot, Latent Copilot has two pilots that can receive and propose client commands, and a client sends its commands to both pilots. To reduce the overhead of having

two pilots actively proposing commands, Latent Copilot tries to operate with one active pilot, which is in active mode and actively proposes the commands, and one latent pilot, which is in latent mode and only proposes the commands if they have not been committed by the active pilot after a timeout. Operating with one active pilot helps Latent Copilot avoid the overhead of Copilot in the normal case while the timeout and fast takeover mechanisms help Latent Copilot achieve 1-slowdown-tolerance.

Latent Copilot introduces different mechanisms to determine when the pilots should switch their modes if it suspects the current active pilot is (constantly) slow while ensuring the stability of the system (i.e., minimizing or avoiding unnecessary mode switching). To reliably infer if a pilot is potentially slow or fast, Latent Copilot uses additional metadata embedded in the ordering messages to learn about the status of the commands from the replicas and learn about the progress of the other pilot. With this mechanism, the latent pilot will become active if a sufficient number of signals indicate that the active pilot is potentially slow. Similarly, the active pilot will become latent if a sufficient number of signals indicate that the latent pilot is fast and the active pilot itself is potentially slow. The promoting and demoting mechanisms together help Latent Copilot operate with one fast active pilot most of the time, while the other (potentially slow) pilot stays latent. To avoid two pilots from quickly demoting each other and both entering latent mode, Latent Copilot uses a min-active-window mechanism which allows a pilot to stay active for at least a duration of the min-active-window once it enters active mode.

## 6.2 Design Goals

Latent Copilot introduces various design decisions and optimizations that together provide 1-slowdown-tolerance while ensuring stability (i.e., minimizing unnecessary/wrong mode switching) and reducing overhead (e.g., reducing the overhead of running two active pilots

during the normal case as in Copilot). Specifically, Latent Copilot tries to achieve the following goals:

- When there are no slowdowns, one pilot should remain active for as long as possible while the other pilot should remain inactive/latent.
- When there is long-term slowdown on a pilot, the fast pilot should be the active pilot while the slow pilot should be the latent pilot.
- When there is transient slowdown on the active pilot, the latent pilot should react quickly and propose the commands if they have not been committed for some timeout.
- The slow latent pilot should not be able to destabilize the system by incorrectly demoting the fast active pilot and promoting itself to active mode.

## 6.3 Basic Design

Latent Copilot has two pilots that can receive and propose client commands, and a client sends its commands to both pilots. Each pilot in Latent Copilot can operate in either active mode or latent mode. In active mode, an active pilot assigns a command to its next log entry and proposes this command immediately when it receives a client command. In latent mode, a latent pilot also receives the commands from clients. Instead of proposing a command immediately upon receiving it from a client, a latent pilot waits for some timeout before it decides the next step. When the timeout fires, it checks whether the command has been committed by the other pilot, which it assumes to be active. If the command has been committed, it will not propose the same command. Otherwise, it will assign the command to its next log entry and propose this command with a dependency on the most recent entry from the other pilot by sending FastAccept messages to all replicas. (Latent Copilot's ordering protocol is the same as Copilot.) It then waits for the replies from the replicas before it determines whether it should enter active mode or remain in latent mode.

When a replica replies to the FastAccept message, it also indicates in its reply whether it has committed the same command in the FastAccept. If a replica replies to the FastAccept and indicates that the command has been committed, the latent pilot will not enter active mode. (It still proceeds and completes the ordering for the proposed entry.) If no replicas indicate that the command has been committed, the latent pilot suspects that the active may have become slow. It tries to do the fast takeover of all uncommitted entries from the other pilot on which its proposed entry has a dependency, and at the same time completes the ordering of the proposed entry.

**Rationale of using FastAccept for inference.** If the latent pilot detects a command has not been committed when a timeout fires, it cannot reliably infer whether the other pilot is slow or this pilot itself is slow (for instance, by being slow at receiving or processing messages). Thus, taking additional actions such as fast takeover and becoming active immediately after a timeout fires may result in unnecessary overhead or even slowdown the system if the latent pilot itself is indeed slow. The idea behind this mechanism is to use additional information that is embedded in the ordering messages by other (fast) replicas in the systems to more reliably infer if the other pilot is potentially slow and if it should enter active mode. It is based on the 1-slowdown assumption that at most one pilot is slow (for achieving slowdown-tolerance guarantee) and a majority of replicas in the system (including both pilots) are fast. If the current active pilot is fast, it should have committed the commands within the timeout and some of the replicas should have learnt about the committed status of these commands. Hence, the latent pilot would learn about the committed status of those commands from the replicas via the FastAcceptReplies. (Even if the latent pilot is slow, it will be slow in learning the committed status of those commands and will only affect itself. It will not slow down the system since the commands have already been committed, and the fast active pilot will be able to nullify this entry from the slow latent pilot on which its future entries have a dependency.) If the current active pilot is indeed

slow and is not able to commit the commands within the timeout, the fast latent pilot would quickly learn the uncommitted status from the replicas.

The basic design of Latent Copilot we described so far can reduce the overhead of operating with two active pilots by having only one active pilot actively propose the commands. It provides 1-slowdown-tolerance by timing out and proposing uncommitted commands when the timeout fires, and thus bounds the maximum latency of a command, which is controlled by the timeout value. If the active pilot is constantly slow or the duration of a slowdown at the active pilot is much longer than a transient slowdown, the system would have better performance if two pilots switch their roles: the slow active pilot becomes latent and the fast latent pilot becomes active. This would provide a command latency much lower than the timeout value and reduce the overhead incurred by the fast latent pilot due to it frequently invoking the fast takeover. Latent Copilot introduces the promoting and demoting mechanisms that help determine when a pilot should become active or latent. They together will ensure that the fast pilot should stay active and and the other (potentially slow) pilot should stay latent.

## 6.4 Switching From Latent Mode To Active Mode

If the active pilot incurs a transient slowdown or network hiccup and then becomes fast again shortly after that, it may not be necessary for the other (latent) pilot to enter active mode to replace the current active pilot. (The latent pilot still proposes the uncommitted commands when the timeout fires.) To further improve the stability of Latent Copilot, our design and implementation supports a configurable min-new-proposals. With min-new-proposals, a pilot becomes active only after min-new-proposals of its consecutive proposed entries receive the indications from the replicas that the commands in those entries have not been committed. A sufficient min-new-proposals larger than 1 would be a stronger signal that shows the other pilot is likely experiencing more than a transient slowdown.

Once in active mode, a pilot remains active for at least a duration of min-active-window: it cannot be demoted to latent mode during this min-active-window. This ensures that the system operates with at least one active pilot most of the time.

## 6.5 Switching From Active Mode To Latent Mode

An active pilot demotes itself to be a latent pilot when it suspects the other pilot is fast and/or itself is not as fast as the other pilot. The challenge is how to reliably infer if the other pilot is fast and when it should demote itself. Latent Copilot overcomes this challenge by using the Commit messages from the other pilot. Specifically, if an active pilot receives a Commit message containing the commands that have not been committed by this active pilot, it starts suspecting that the other pilot is fast and itself is slow. An active pilot becomes latent if it receives at least min-new-commits of such Commit messages outside the min-active-window. Enforcing min-new-commits threshold, which is configurable in our design and implementation, helps prevent an active pilot from unnecessarily switching to latent mode if it is just experiencing a transient slowdown. (Even if there is a small time window when both pilots are in latent mode, the timeout mechanism of a latent pilot ensures the commands during this window will be proposed once the timeout fires. Existing mechanisms of Latent Copilot are sufficient to bound the maximum latency for a command. In other words, even if such a window occurs, Latent Copilot is still 1-slowdown-tolerant.)

**Rationale of using Commit for inference.** Using the new commit is a good signal for the receiving active pilot to make inference since a commit of new (uncommitted) commands is a proof that the other pilot is able to complete useful work. In addition, the other (latent) pilot has been waiting for some timeout before proposing and committing the new commands which this active pilot has not even committed. This suggests that the receiving active pilot might not be as fast as the other pilot. Combining the Commit signal with the min-new-commits threshold leads to a reliable signal for the active pilot to become latent.

# Chapter 7

# s-Slowdown-Tolerant Protocol Design

This section describes the design of $s$-slowdown-tolerant Copilot, which extends the design of 1-slowdown-tolerant Copilot to tolerate any $s$ slow replicas ($s \geq 1$). This section covers the main parts of the protocol—model, ordering, execution, fast takeovers, and optimizations. Finally, it covers the correctness arguments for our $s$-slowdown-tolerant Copilot.

## 7.1 Model

$s$-slowdown-tolerant Copilot assumes the crash failure model and an asynchronous system. It requires $2f+1$ replicas to tolerate at most $f$ failures, and guarantees linearizability as a correctness condition despite any number of failures. It provides $s$-slowdown-tolerance (cf. §2.2) in the presence of any $s$ slow replicas ($s \leq f$). (Note that since the set of slow replicas is a superset of failed replicas, we cannot tolerate more than $f$ slowdowns or failures while still guaranteeing liveness. If there are more than $f$ slow replicas, then any majority quorum will intersect with at least one slow replica, and so the protocol will be slowed down.)

## 7.2 Ordering

To tolerate $s$ slowdowns ($s \leq f$), $s$-slowdown-tolerant Copilot needs $s+1$ pilots, $\{P_0, P_1, \ldots, P_s\}$, and clients send each command to all pilots. The ordering protocol places client commands in pilots' logs each coordinated by a pilot. These separate logs are ordered together using dependencies. Like 1-slowdown-tolerant Copilot, the ordering protocol of $s$-slowdown-tolerant Copilot always includes a FastAccept phase and sometimes includes an Accept phase. In the FastAccept phase, a pilot proposes its log entry with an initial dependency for that log entry. If it can gather a fast quorum of $f + \lfloor \frac{f+1}{2} \rfloor$ FastAcceptOk replies (including from itself), a pilot can commit its entry with the initial dependency on the fast path. Otherwise, it proceeds to the regular path by selecting the final dependency and persisting it in the Accept phase.

**Dependency.** Dependency in $s$-slowdown-tolerant Copilot is a vector $D$ of size $s+1$, $D = [e_0, e_1, \ldots, e_s]$, where each sub-dependency, $e_l$, indicates the prefix of the pilot $P_l$'s log that should be executed before a given entry. In other words, $e_l$ indicates a dependency on the entry $P_l.e_l$ of the pilot $P_l$.

**Pilots propose commands and an initial dependency.** Upon receiving a command from a client, a pilot puts the command into its next available log entry. It also assigns the initial dependency $D = [e_0, e_1, \ldots, e_s]$ for this entry. Each sub-dependency $e_l$ in $D$ is the most recent entry from the pilot $P_l$ this pilot has seen. (For the sub-dependency on this pilot itself, it can either put an empty dependency or the preceding entry of this pilot's log since we maintain the partial order of each pilot's log). It then proposes this assignment of command and initial dependency $D$ for this entry to other replicas by sending them FastAccept messages.

**Replicas reply to FastAccepts and Compatibility check.** When a replica receives a FastAccept message for an entry $P_k.i$ with an initial dependency $D$ from a pilot $P_k$ it checks

if the initial dependency vector $D$ is compatible with all previously accepted dependencies from each of the other pilots.

An entry $P_k.i$ with an initial dependency $D = [e_0, e_1, \ldots, e_s]$ is compatible at a replica if and only if $P_k.i$ with an initial dependency $e_l$ is compatible with all previously accepted dependencies from a pilot $P_l$ for all $0 \leq l \leq s$ and $l \neq k$. A replica does the compatibility check for an entry $P_k.i$ with its initial dependency $D = [e_0, e_1, \ldots, e_s]$ by doing the individual compatibility check for $P_k.i$ with an initial dependency $e_l$ against all previously accepted dependencies from the pilot $P_l$ for all $0 \leq l \leq s$ and $l \neq k$. (The individual compatibility check is the same as 1-slowdown-tolerant Copilot.) The compatibility check for $P_k.i$ with its initial dependency $D$ passes if and only if all individual compatibility checks for $P_k.i$ with an initial dependency $e_l$ pass.

**Replicas reply to FastAccepts.** If the initial dependency $D$ is compatible, a replica fast accepts it and replies with a FastAccepOk message to the proposing pilot $P_k$. Otherwise it replies with a FastAcceptReply message that includes the suggested sub-dependencies for the initial sub-dependencies whose individual compatibility checks do not pass. Specifically, a replica constructs a suggested dependency vector $D' = [e'_0, e'_1, ..., e'_s]$ of size $s+1$, where $e'_l$ $(0 \leq l \leq s)$ is the same as the initial dependency $e_l$ if the individual compatibility check for $e_l$ passes and the latest entry of pilot $P_l$ at this replica otherwise. It then includes $D'$ in its FastAcceptReply message to the proposing pilot $P_k$.

**Pilots try to commit on the fast path.** If the proposing pilot $P_k$ can gather a fast quorum of $f + \lfloor \frac{f+1}{2} \rfloor$ FastAcceptOk replies (including from itself), it can commit its entry with the initial dependency $D$, which now becomes the final dependency. It then sends a Commit message to the other replicas to inform them of the committed commands and the final dependency for this entry.

**Pilots persist the final dependency in the Accept phase.** If a pilot cannot gather a fast quorum of FastAcceptOks, it waits until it receives at least $f+1$ FastAcceptOks and FastAcceptReplys and then proceeds to the Accept phase. In the Accept phase, a pilot constructs the final dependency vector based on the suggested dependencies from the responses in the FastAccept round. All FastAcceptOk messages (including the pilot's) suggest the initial dependency vector. To construct the final dependency vector $D = [e_0, e_1, \ldots, e_s]$, we select each sub-dependency $e_l$, an entry from pilot $P_l$, in a manner similar to how 1-slowdown-tolerant Copilot selects the final dependency for its Accept phase. Specifically, to select the final sub-dependency $e_l$, we sort the $l$-th sub-dependencies from all suggested dependency vectors and then select the $(f+1)$-th as the final sub-dependency. Each sub-dependency $e_l$ of the final dependency vector $D$ is high enough to capture the necessary ordering constraint between the proposed entry $P_k.i$ of pilot $P_k$ and the entries that have already been committed and potentially executed by the pilot $P_l$. Any entry beyond $e_l$ of pilot $P_l$ will have its own dependency on this entry $P_k.i$ since $P_k.i$ arrived at a majority quorum first. After constructing the final dependency $D$, a pilot proceeds to the Accept phase to persist this final dependency in the same way as 1-slowdown-tolerant Copilot. It does so by sending an Accept message that includes the final dependency $D$ to all replicas. When it gathers $f+1$ AcceptOks replies from the replicas, it commits the entry and sends Commit messages to all replicas.

## 7.3 Execution

**Combining logs and execution deduplication.** Like 1-slowdown-tolerant Copilot, replicas in $s$-slowdown-tolerant Copilot execute commands in the combined log order, and the pilots' logs are combined using the dependencies. In the absence of failures, each command is in the combined log $s+1$ times. A replica also uses the execution deduplication

mechanism: it executes each command only once in its first position and ignores the later appearances. A replica replies to the client if the replica is a pilot.

**Total order of commands.** Like 1-slowdown-tolerant Copilot, $s$-slowdown-tolerant Copilot determines the total order of commands in the combined log by the partial order of each pilot's log, the dependencies between them, and a priority rule. When cycles exist between the logs, $s$-slowdown-tolerant Copilot breaks a cycle deterministically using the priority of the pilots: the entries of a pilot with smaller pilot index are ordered before the entries of a pilot with larger pilot index, i.e., $P_0$'s entries $< P_1$'s entries $< \ldots < P_s$'s entries in the presence of cycles. Note that cycles between a pilot $P_k$'s log and each sub-dependency are handled separately. That is, cycles between $P_k$'s entries and $P_l$'s entries can be broken independently of cycles between $P_k$'s entries and any other pilots' entries.

**Executing in order.** A replica executes a command once its entry is committed and all preceding entries in the total order have been executed. The following rules determine when it is safe for a replica to execute a command in entry $P_k.i$ with dependency vector $D = [e_0, e_1, \ldots, e_s]$: (0) $P_k.i$ is committed, and (1) it has executed $P_k.(i-1)$ and then one of the following condition holds: (2) it has executed all sub-dependencies $P_l.e_l$ in $D$, or (3) for each unexecuted sub-dependency $P_l.e_l$, a cycle exists between the entry $P_k.i$ and all unexecuted entries $\leq P_l.e_l$ of pilot $P_l$ and $k < l$ (i.e., pilot $P_k$ has higher priority than $P_l$). To ensure commands are executed in the total order, a replica must wait for the commit of all potentially preceding entries. The fast takeover protocol ensures a fast pilot need not wait long before executing an entry.

## 7.4 Fast Takeover

The fast takeover procedure for $s$-slowdown-tolerant Copilot is similar to 1-slowdown-tolerant Copilot. A pilot sets a takeover-timeout when it has a committed entry but has

not seen the commits for the entries in the other pilots' logs that potentially precede this entry. When the takeover-timeout fires, the pilot starts the fast takeovers of all uncommitted entries in the other pilots' logs that potentially precede this entry.

A pilot invokes fast takeover to complete the ordering work of a slow pilot using Paxos's two phases of prepare and accept. To safely takeover an entry $P_k.i$ of pilot $P_k$, it creates a new ballot number that is higher than any ballot number that has been prepared for $P_k.i$. It sends Prepare messages that include this new ballot number to all replicas and waits for at least $f+1$ PrepareOks (including from itself). The recovery value picking procedure examines the set of PrepareOks to correctly recover the command and dependency that could have been committed in the entry $P_k.i$. For the easy cases where the value of $P_k.i$ can be conclusively resolved, the value picking procedure of $s$-slowdown-tolerant Copilot is similar to 1-slowdown-tolerant Copilot (§3.4).

The tricky cases are when there are in the range of $[\lfloor\frac{f+1}{2}\rfloor, f)$ PrepareOk replies with fast-accepted as their progress. These 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/pilots also fail concurrently. In these scenarios, the value picking procedure examines the values of the concurrent entries from other pilots in order to determine the value of $P_k.i$. (The concurrent and potentially conflicting entries are included in PrepareOk reply if a replica did not fast accept the initial dependency of $P_k.i$ and it suggested a different dependency.) Three possible subcases are as follows:

1. If each of the concurrent entries either commits with a no-op or has a dependency $D$ where its sub-dependency on pilot $P_k$, $D[k]$, is $\geq i$, it is safe to commit the initial command and dependency in $P_k.i$.
2. If at least one of the concurrent entries commits with a dependency $D$ and $D[k] < i$, this implies the pilot $P_k$ could not have committed on the fast path and thus it is safe to commit a no-op with an empty dependency in $P_k.i$.

3. If the value of a concurrent entry, say $P_l.j$ is unknown (e.g., due to their proposing pilots failing before sending out Commit messages), the value picking procedure may need to resolve the value of $P_l.j$ in order to determine the value of $P_k.i$. It does so by simultaneously preparing $P_k.i$ and $P_l.j$, and then it resolves $P_k.i$ and $P_l.j$ using the same quorum of replicas that reply to the SimultaneousPrepare messages. These steps are similar to how 1-slowdown-tolerant Copilot handles this tricky case.

When the number of pilots is more than two, it is possible that multiple pilots try to simultaneously fast takeover the same entry of a slow pilot, potentially leading to dueling proposers. We avoid dueling proposers using two techniques: non-uniform takeover-timeout and randomized exponential backoff. Non-uniform takeover-timeout technique assigns different takeover-timeout values to different pilots. For instance, the pilots $P_0$, $P_1$, and $P_2$ use the takeover-timeout values of 10 ms, 10.5 ms, and 11 ms, respectively. (Alternatively, a pilot can randomly select a takeover-timeout value from a predefined set of values every time it sets a takeover-timeout.) Having different takeover-timeout values across the pilots minimizes the likelihood that the pilots do fast takeovers at the same time, and thus reduces the likelihood of dueling proposers. When a pilot incurs the dueling proposers problem, it applies the standard technique of randomized exponential backoff before retrying the fast takeover of the same entry.

## 7.5 Optimizations

### 7.5.1 Ping-Pong Batching

We explain how we extend ping-pong batching for $s$-slowdown-tolerant Copilot. Ping-pong batching coordinates the pilots to propose compatible ordering to replicas. A pilot $P_i$ closes a batch and tries to FastAccept the batch when either it receives a FastAccept message from the pilot $P_{i-1}$ or its ping-pong-wait timeout fires. (The pilot $P_0$ waits for the last pilot, $P_s$.) In order to avoid the premature timeout that may lead to a pilot proposing incompatible

ordering, ping-pong-wait timeout should be chosen carefully so that its value should be greater than $(s+1)\times$ one-way delay between replicas.

### 7.5.2 Null Dependency Elimination

Null dependency elimination allows a fast pilot to avoid waiting on commits from a slow pilot. Null dependency elimination optimization for 1-slowdown-tolerant Copilot can be generalized to work for $s$-slowdown-tolerant Copilot. It looks inside each sub-dependency of a dependency vector to see the commands it contains. If all commands in every sub-dependency have already been executed, we call the dependency vector a null dependency: the execution deduplication will avoid executing these commands and their final ordering information is irrelevant. Thus, the fast pilot can avoid waiting on commits from the slow pilots. Note that waiting for the commits of the sub-dependencies whose commands have been executed is still necessary if any of the other sub-dependencies contain unexecuted commands since the final dependency information of the former may affect the ordering between the entry and the sub-dependencies containing unexecuted commands.

To safely nullify a dependency vector, a replica performs two additional checks before it can actually nullify the commands inside the sub-dependencies. First, it checks if the *null-dep-safe* flag—an additional field about the dependency vector maintained by each log entry—is true. Second, it checks if its view about each sub-dependency's pilot is sufficiently up-to-date. A pilot is in charge of determining when it is safe to consider nullifying the dependencies that the entries in its log have on the other pilots' logs. It does so by updating an entry's null-dep-safe flag during its ordering and committing of that entry. For a committed entry $P_k.i$ with a dependency $D = [e_0, e_1, \ldots, e_s]$, $D$ is safe to be considered for nullifying—i.e., $P_k.i$'s null-dep-safe flag is true—if for every sub-dependency entry $P_l.e_l$ in $D$ ($l \neq k$), at least a majority of replicas that have the same (current) view about $P_l$ have advanced the latest entry in their $P_l$'s log to be $\geq P_l.e_l$. Replicas indicate if they have the same view about $P_l$ and have advanced the latest entry in their $P_l$'s log to be $\geq P_l.e_l$

using a *dep-seen* flag for each sub-dependency that is included in their replies to a Fast-Accept/Accept message for $P_k.i$. The mechanism to keep track and set the dep-seen flag for each sub-dependency is similar to that in 1-slowdown case (§5.2).

## 7.6 Correctness Arguments

We provide the correctness arguments for our $s$-slowdown-tolerant Copilot: why it is both safe, i.e., it provides linearizability, and live, i.e., all client commands eventually complete.

### 7.6.1 Safety

To prove linearizability, we need to show that client commands are (1) executed in some total order, and (2) this order is consistent with the real-time order of client operations.

**Real-time order.** To prove the real-time ordering property, we need to show if command $a$ completes in real-time before command $b$ begins, then $a$ must be ordered before $b$. Consider a command $a$ that completes before a command $b$ begins. Let $P_0, P_1, \ldots, P_s$ represent $s+1$ pilots. Since $a$ completes, it must be committed in at least one pilot's log; suppose w.l.o.g. it commits in $P_0$'s log at entry $P_0.i$. Within $P_0$'s log, $a$ is trivially ordered before $b$ because $b$ is issued only after $a$ has been committed. In any other $P_l$'s log ($1 \leq l \leq s$), $a$ and $b$ may be committed in either order. Consider any $P_l$'s log ($1 \leq l \leq s$). Assume $P_l.j_l$ is the entry of $P_l$'s log where command $b$ is placed. We will show that $P_0.i$ is executed before $P_l.j_l$ by using the reasoning similar to that in 4.1. The same key observation is applied: $P_l.j_l$ cannot have a dependency that precedes $P_0.i$ because this would be considered incompatible by the compatibility check for $P_l.j_l$ with its sub-dependency on pilot $P_0$ during the FastAccept phase (cf. §7.2). Since $P_0.i$'s dependency $< P_l.j_l$ and $P_l.j_l$'s dependency $\geq P_0.i$, no cycles exist between $P_0.i$ and $P_l.j_l$. Hence, $P_0.i$ is executed before $P_l.j_l$.

Since $P_0.i$ is executed before $P_l.j_l$ for all $1 \leq l \leq s$, the entry $P_0.i$, which contains command $a$, is executed before the entries that contain command $b$ in the logs of the other pilots. This implies $a$ is executed before $b$.

**Total order.** To prove the total ordering property, we first need to prove the following invariant: for any two log entries $P_k.i$ from pilot $P_k$ and $P_l.j$ from pilot $P_l$ ($0 \leq k, l \leq s$), either $P_k.i$ has a dependency $\geq P_l.j$ or $P_l.j$ has a dependency $\geq P_k.i$. Then we show the consistency property: the value of an entry—i.e., the commands and dependency—is consistent across replicas. The formal proof can be done similarly to the 1-slowdown-tolerant design. We provide some intuition and reasoning why the invariant and the consistency property are true for the $s$-slowdown-tolerant design.

Recall that an entry $P_k.i$ of pilot $P_k$ with its committed dependency is ensured to be compatible with other pilots' entries by the ordering protocol, and that they are compatible if and only if $P_k.i$ with its sub-dependency on each of the other pilots is compatible with the entries from that pilot. The mechanisms of checking compatibility and constructing each final sub-dependency is the same as those used between two pilots in 1-slowdown-tolerant Copilot. By applying the arguments similar to those in the 1-slowdown case (cf. §4.1) to each pair of pilots separately, we can show the following invariant for $s$-slowdown-tolerant Copilot: if two log entries $P_k.i$ and $P_l.j$ commit at the pilots $P_k$ and $P_l$, respectively, either $P_k.i$ has a dependency $\geq P_l.j$ or $P_l.j$ has a dependency $\geq P_k.i$. This ensures that a dependency path exists from one entry to the other, preventing them from being ordered differently at different replicas.

Next we intuitively explain that each entry in a pilot's log commits with the same commands and dependency across replicas, even in the presence of failures (including failures of $s$ pilots). We use the arguments similar to those in §4.1. The following trivial cases can be handled and proved similarly: (1) a reply from the prepare phase (of a fast takeover)

indicates the entry is committed, or (2) any reply indicates the entry is accepted, or (3) $\geq f$ replies show it is fast-accepted, or (4) $< \lfloor \frac{f+1}{2} \rfloor$ indicates it is fast-accepted.

The tricky cases occur when there are in the range of $[\lfloor \frac{f+1}{2} \rfloor, f)$ replies that indicate fast-accepted as their progress. In these cases, the entry may or may not have been committed. The recovery value picking procedure examines some or all concurrent and potentially conflicting entries from other pilots. If at least one of the concurrent entries commits with a dependency vector $D$ where its sub-dependency on the failed pilot precedes the recovered entry, this implies the failed pilot could not have committed on the fast path before it failed. Thus, it is safe to commit a no-op in the recovered entry. If each of the concurrent entries either commits with a no-op or has a sub-dependency on the failed pilot $\geq$ the recovered entry, this means the initial dependency proposed in the recovered entry is compatible with the concurrent entries from other pilots. Thus, it is safe to commit the commands and the dependency initially proposed by the failed pilot in the recovered entry. It is safe even if the failed pilot did not commit any value before it failed because no other values have ever been committed. Note that examining the set of concurrent entries that are gathered from at least $f+1$ replicas is sufficient. All entries that are later than these entries would have a sub-dependency $\geq$ the recovered entry since they arrive after the recovered entry at $\geq (f+1)$ replicas.

Since the commands and dependency in an entry are the same across replicas, and the entries of pilots' logs are executed in the same order, the commands are executed in the same total order.

### 7.6.2 Liveness

To prove liveness, we need to show that a command issued by a client eventually receives a response. Due to FLP [19], we assume the system is eventually partially-synchronous [17] and that all messages are eventually delivered.

Assume that for each pilot $P_l$'s log ($0 \leq l \leq s$) a replica has executed all entries in $P_l$'s log up to the entry $P_l.i_l$. We intuitively explain that the replica eventually executes the next entry $P_l.(i_l + 1)$ from one of the pilots' logs. We examine two cases: the failure-free case and the case of failures. We consider the failure-free case first.

A committed entry $P_l.(i_l + 1)$ with a dependency $D$ is ready for execution immediately if either (1) all sub-dependencies in $D$ have been executed, or (2) for every sub-dependency that has not been executed in $D$, pilot $P_l$ has higher priority than the sub-dependency's pilot, and cycles exist between $P_l.(i_l + 1)$ and all entries that are $\leq$ the sub-dependency and have not been executed in the log of the sub-dependency's pilot.

If a committed entry $P_l.(i_l + 1)$ cannot be executed because some sub-dependencies must be executed first, the execution switches to the log of one of the pilots that propose those unexecuted sub-dependencies, and checks if the next entry of that pilot's log is ready for execution. If the next entry of that pilot's log cannot be executed due to some unexecuted sub-dependencies that need to be executed first, the execution switches to another pilot's log. The execution keeps switching until it finds one of the next entry $P_l.(i_l + 1)$ from one of the pilots' logs that can be executed (i.e., it satisfies one of the conditions (1)–(2)). The key observation is that such an entry is eventually found since the number of pilots, $s + 1$, is finite. If every next entry of each pilot has a sub-dependency on unexecuted entries, then there must be a cycle among the next entries of the pilots (because the set of pilots is finite). In that case, the next entry $P_l.(i_l + 1)$ in the log of the pilot with the highest priority can be executed. Thus, the system can make progress by eventually executing the next entry $P_l.(i_l + 1)$ from one of the pilots' logs.

We now consider the case of failures. If only non-pilot replicas fail, this reduces to the failure-free case. If some pilots are slow/failed and at least one pilot, say $P_l$, is alive, the pilot $P_l$ may not be able to execute its next committed entry $P_l.(i_l + 1)$ since some sub-dependencies in $P_l.(i_l + 1)$'s dependency may not have been committed by the failed pilots. In this case, the pilot $P_l$ eventually does the fast takeovers of all uncommitted sub-

dependencies/entries from the slow/failed pilots that potentially precede $P_l.(i_l + 1)$. Once the fast takeovers complete (as argued below), this case reduces to the failure-free case where all sub-dependencies are committed. Thus, $P_l.(i_l + 1)$ can eventually be executed.

If all pilots are slow/failed, a replica eventually invokes a view change to elect new pilots. Fast takeovers and view changes cannot repeat indefinitely by relying on the same partial synchrony assumptions that basic Paxos and Multi-Paxos make to ensure progress. When a new replica is elected as the new pilot $P_l$, it will learn about the committed entry $P_l.(i_l + 1)$ from the majority of replicas (including itself) since the entry $P_l.(i_l + 1)$ has been committed. $P_l.(i_l + 1)$ can eventually be executed as shown in the previous cases where at least one pilot is alive.

In both failure-free case and failure case, we have shown that the system makes progress by eventually executing the next entry $P_l.(i_l + 1)$ from one of the pilots' logs.

# Chapter 8

# Evaluation

## 8.1 Implementation and Baseline

We implemented Copilot in Go using the framework of EPaxos [41] to enable a fair comparison with the baselines. We use the framework's implementations of *EPaxos* and *Multi-Paxos*. The Multi-Paxos implementation is representative of well optimized Multi-Paxos [38, 27, 12]. Clients send commands directly to the leader, the leader gets those commands accepted in a single round of messages to the replicas, it executes the commands in log order, and then it replies to the clients. Replicas execute commands in log order but do not reply to the client. Any performance improvement we made to Copilot's implementation we also applied to EPaxos and Multi-Paxos to ensure the comparison remains fair.

EPaxos and Multi-Paxos can use the *thrifty* optimization to send and receive messages only to the required number of other replicas. The thrifty optimization improves performance by decreasing load on all replicas in EPaxos and the leader in Multi-Paxos. It also harms slowdown-tolerance by eliminating redundancy from the ordering in these systems. Our latency slowdown experiments do not use the thrifty option for Multi-Paxos and

EPaxos to show them in their best possible setting. Our throughput and latency experiments without slowdowns compare to the baselines with and without the thrifty optimization.

The EPaxos and Multi-Paxos baselines send pings every 3 s to make sure each replica has not failed. An alternative that would make them more slowdown tolerant, though less stable and unable to use some optimizations, is to use a very short view-change timeout. *Fast-View-Change* is a baseline we use to represent this alternative. Our implementation builds on the view-change implementation for Multi-Paxos in the EPaxos framework. It differs from a faithful implementation in two ways that decrease the time to complete a view change. Thus, its performance is an upper bound on that of a more faithful implementation. The first difference is that view-changes are triggered by a master process that never fails or becomes slow. The master receives heartbeats from the current leader every 1 ms and triggers a view-change as soon as 10 ms have elapsed with no heartbeats. (This timeout matches the fast-takeover timeout for Copilot.) The second difference is that a view-change immediately identifies the next leader instead of running an election, making the view-change process similar to that for viewstamped replication [43, 35]. If a client has not received a response to its command after 10 ms, it contacts the master to learn the current leader and resubmits its command to that leader.

## 8.2 Experimental Setup

Experiments were run on the Emulab testbed [52], where we have exclusive bare-metal access to 21 machines. Each machine has one 2.4 GHz 64-bit 8-Core processor, 64 GB RAM, and is networked with 1 Gbps Ethernet. These machines are located in the same datacenter with an average network round-trip time of about 0.1 ms. Thus, our evaluation of Copilot is focused on a datacenter setting with small latencies between replicas. We also experiment with Copilot in a geo-replicated setting and discuss our results in subsection

§8.3.6. Optimization of Copilot for a geo-replicated setting is an interesting avenue of future work.

**Configuration and workloads.** We use 5 machines to create an RSM with 5 replicas that can tolerate at most 2 failures. We use 5-replica RSMs since they are a common setup for fault-tolerant services inside a datacenter [9]. Clients run on separate machines in the same facility. We use a simple workload with 8 byte commands that overwrite 4 bytes of data.

We run each experiment for 3 minutes and exclude the first and the last 30 seconds of each run to avoid experimental artifacts. To determine how to fairly configure our latency experiments, we probed the operation of each system under increasing load. For each system, we choose the number of closed-loop clients where the system operates at 50% of its peak load. This reduces the effect of queuing delays.

We enable batching for EPaxos and Multi-Paxos with a batching interval of 0.1 ms, which is similar to the effective length of Copilot's ping-pong batches. This choice of batching interval ensures all systems have similar median latency at low and moderate load. Copilot uses a ping-pong-wait timeout of 1 ms and a fast-takeover timeout of 10 ms.

For Multi-Paxos, clients send commands to the leader. For Copilot, clients send commands to both pilots. For EPaxos, each client has a designated replica it sends commands to.

EPaxos includes an interface that allows service builders to provide specialized logic in their implementation that identifies when two commands conflict. This allows EPaxos to avoid needing to determine an order between non-conflicting commands. We compare to EPaxos with 0%, 25%, and 100% conflicts. The 0% case is EPaxos's best case. The 100% case is EPaxos's worst case and also represents its performance when used as a generic RSM without its specialized interface. The 25% case is a middle ground.

**Severity and duration.** Slowdowns vary in their severity and their duration. The *severity* of a slowdown indicates its magnitude, e.g., a replica taking an extra 10 ms or an extra

80 ms to send responses. The *duration* of a slowdown indicates how long the slowdown lasts, e.g., 1 second or 10 minutes. For example, a replica could take an extra 10 ms to respond to every message it receives during a 1-second duration. We present experiments that evaluate tolerance of slowdowns of varying severity, duration, and manifestation.

## 8.3 Copilot

Copilot provides 1-slowdown-tolerant RSMs by using two pilots to provide redundancy at every stage of processing a command. This evaluation demonstrates the benefit and quantifies the overhead of Copilot. Specifically, it asks:

§8.3.1 Can Copilot tolerate transient slowdowns?

§8.3.2 Can Copilot tolerate slowdowns of varying severity?

§8.3.3 Can Copilot tolerate slowdowns of varying manifestations?

§8.3.4 How does the throughput and latency of Copilot compare to existing consensus protocols?

§8.3.5 Can Copilot maintain service availability when a pilot fails?

§8.3.6 How does the latency of Copilot compare to existing consensus protocols in a geo-replicated setting?

**Summary.** We find that Copilot tolerates *any* one replica slowdown regardless of the type of slowdown, the role of the slow replica, or how slow the slow replica becomes. Copilot's latency under slowdown scenarios is comparable to its normal case latency when no replicas are slow. Copilot tolerates slowdowns better than Multi-Paxos, EPaxos, and Multi-Paxos with fast view changes. All commands in Multi-Paxos see high latencies when the leader is slow. EPaxos incurs a partial slowdown when any of the replicas is slow, and a slow replica can slow down other normal replicas under high conflict rates. Multi-Paxos with fast view changes tolerates the slowdowns that its low timeout detects, but it does not tolerate slowdowns that go undetected. Copilot achieves slowdown tolerance through

redundancy. Although this incurs more messages and processing, we find that Copilot's throughput and latency are competitive with Multi-Paxos and EPaxos. We find that Copilot provides better service availability than Multi-Paxos and EPaxos during one pilot failure due to its proactive redundancy.

### 8.3.1 Transient Slowdowns

Figure 8.1 shows the latency of client commands for Copilot, Multi-Paxos, and EPaxos as transient slowdowns of increasing severity are injected. Transient slowdowns are injected every second starting at time 2 seconds. The injected slowdowns are pauses of increasing length, i.e., the severity and duration of the slowdown are both equal to the pause length. The pause lengths are 0.5 ms, 1 ms, 2 ms, 5 ms, 10 ms, 20 ms, 40 ms, and 80 ms. The pauses are injected by stopping all processing for the specified length inside the Go processes. The slowdowns are injected on a pilot for Copilot, on the leader for Multi-Paxos, and on a replica for EPaxos.

*Multi-Paxos and EPaxos slow down.* Multi-Paxos and EPaxos each have latency spikes that increase proportionally with the length of the injected pause. For instance, for pauses of 40 ms, Multi-Paxos and EPaxos have commands with 40.1 ms and 41.5 ms respectively.

*Fast-View-Change tolerates transient slowdowns.* Fast-View-Change limits the maximum latency by detecting the pause and switching to a new leader. Maximum latency is controlled by the client timeout and view-change timeout. We see a maximum latency around their sum of 20 ms when a client needs to retransmit its command twice because the view-change had not completed after its first timeout. For instance, Fast-View-Change has commands with 25 ms latency for a 40 ms pause.

*Copilot tolerates transient slowdowns.* The latency for Copilot remains low and close to its latency when there are no slowdowns. For very small pauses, e.g., 0.5 ms, Copilot simply waits out the pause. This does not mask the slowdown and does show up in client command latency, but its magnitude is small enough that latency remains similar. For

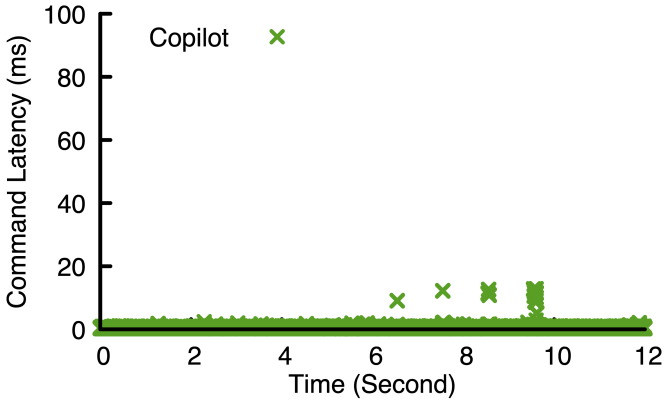
longer pauses, Copilot's fast-takeover timeout of 10 ms fires and the fast pilot completes the ordering work of the slow pilot. This keeps latency low and close to the timeout value. For instance, the maximum command latency is 12.6 ms for a 40 ms pause. The maximum latency during the onset of a slowdown is thus controlled by the fast-takeover timeout value. Latency as a slowdown continues, however, is even lower as our next experiment shows.

### 8.3.2 Slowdowns of Varying Severity

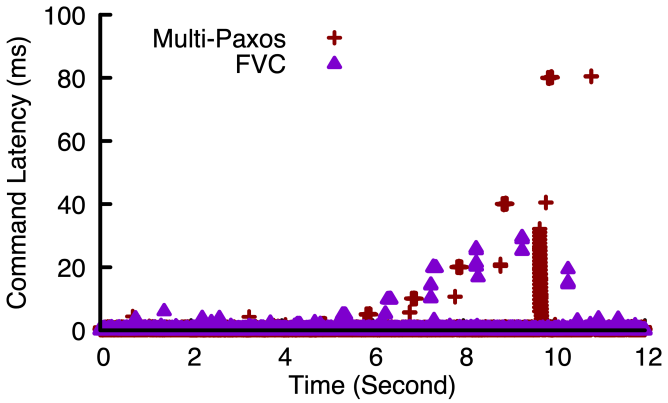
Figure 8.2 shows a CDF of latency for Copilot, Multi-Paxos, and EPaxos in the normal case (0 slowdown) and with slowdowns of varying severity that last for the duration of the experiment. A slowdown of the given severity is injected on one of the pilots for Copilot, the leader for Multi-Paxos, and a replica for EPaxos. The duration of these slowdowns is the length of the experiment (they last longer than the slowdowns evaluated in the previous subsection). The slowdowns are injected using Linux's traffic control (tc) to add delay corresponding to the severity on the slow replica. The severity ranges from 0.5 ms to 40 ms.

*Multi-Paxos and EPaxos slow down.* Figure 8.2b shows the CDF of latency for Multi-Paxos. The latency of client commands in Multi-Paxos is proportional to $2\times$ the severity of the slowdown. The slowdown affects latency twice because the leader appears twice on the path for client commands: the message path is client-to-leader-to-replicas-to-leader-to-client. Fast-View-Change has similar results to Multi-Paxos when the severity of the slowdown is less than the view-change timeout and it avoids the slowdown using a view-change when the severity is greater than the timeout.

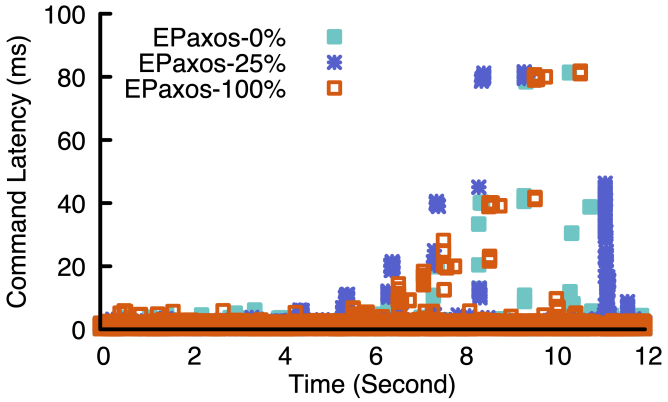
Figure 8.2c shows the CDF of latency for EPaxos with 25% conflicts. Normal case latency is higher than Multi-Paxos because EPaxos processes batches together, and if one command in a batch acquires a dependency then the entire batch goes to the slow path and does a dependency wait. With 25% conflicts, almost all batches have at least one command with a dependency and thus almost all have higher latency than Multi-Paxos. Slowdowns have two effects for EPaxos that result in two step functions in latency. First,




**(a) Copilot.**

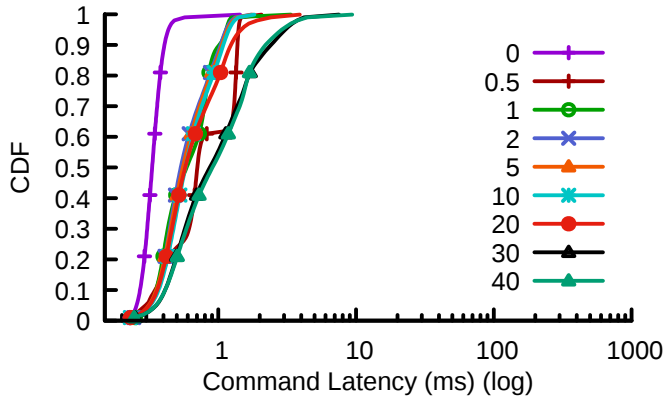



**(b) Multi-Paxos.**

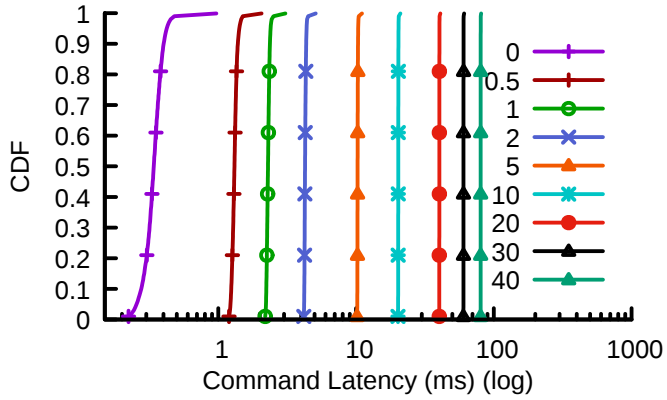



**(c) EPaxos.**

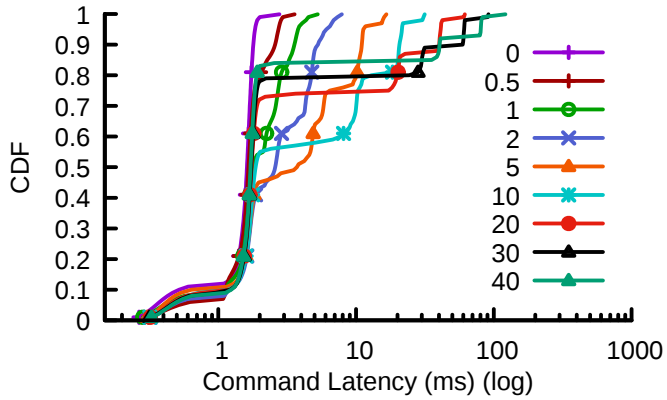
**Figure 8.1: Client command latency for Copilot, Multi-Paxos, Fast-View-Change, and EPaxos with transient slowdowns. Transient slowdowns are injected every second starting at time 2 seconds. The severity and duration of the slowdowns in order are 0.5 ms, 1 ms, 2 ms, 5 ms, 10 ms, 20 ms, 40 ms, and 80 ms. Multi-Paxos and EPaxos have spikes in latency proportional to the slowdowns. Fast-View-Change tolerates the slowdowns using view changes to limit the maximum latency. Copilot tolerates the transient slowdowns because fast takeovers limit maximum latency.**




**(a) Copilot.**




**(b) Multi-Paxos.**




**(c) EPaxos-25%.**

**Figure 8.2: CDF of command latency for Copilot, Multi-Paxos, and EPaxos in the normal case (0) and with slowdowns of varying severity in ms. Slowdowns are injected for the duration of the experiment. Multi-Paxos and EPaxos have latency that increases proportionally with the severity of the slowdown. Copilot's latency stays low during the slowdowns because the fast pilot completes all stages of processing commands. In addition, null dependency elimination avoids having the fast pilot either wait on or fast takeover the ordering work of the slow pilot during the duration of a slowdown.**

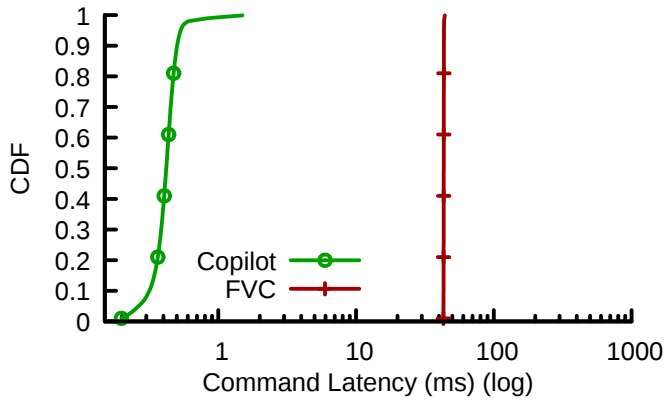
the upper percentiles show a slowdown proportional to 2× the severity of the slowdown. This is due to the increased latency for commands whose designated replica is the slow replica. Second, the middle percentiles show a slowdown proportional to 1× the severity of the slowdown. This is due to the increased latency for commands that are ordered by a fast replica but that acquire a dependency on a command ordered by the slow replica. These commands wait on commits from the slow replica (§2.3). The CDF of latency for EPaxos with 0% conflicts (not shown) shows only the first effect. The CDF of latency for EPaxos with 100% conflicts (not shown) shows both effects with the latency of nearly all commands affected.

*Copilot tolerates slowdowns of varying severity.* Figure 8.2a show the CDF of latency for Copilot. Normal case latency is similar to Multi-Paxos. Copilot's latency under these slowdowns is related to its ping-pong-wait timeout of 1 ms. The fast pilot forms batches when either it hears from the slow pilot or its ping-pong-wait timeout fires. The fast pilot orders client commands in earlier batches than the slow pilot. Thus, null dependency elimination enables the fast pilot to avoid waiting on the slow pilot or having to fast takeover its work. The larger batches result in an increase in the latency for Copilot compared to its normal case, but this increase is small and overall performance is similar. Even in the worst case during a slowdown, median, 90th, and 99th percentile latencies are within 0.6 ms, 2 ms, and 4 ms of their values when there is no slowdown, respectively. Thus, we conclude that Copilot's implementation is resilient to slowdowns.

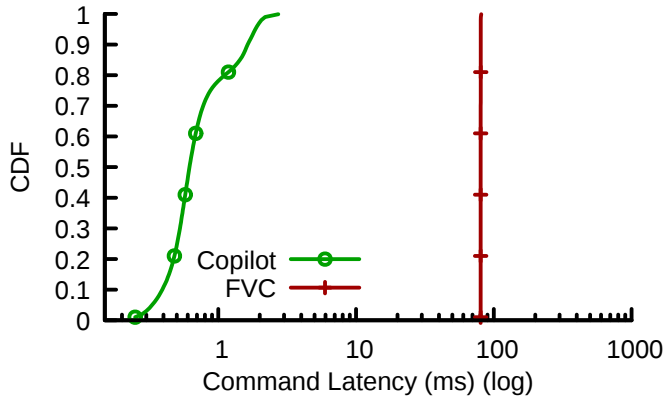
### 8.3.3 Slowdowns of Varying Manifestations

Figure 8.3 compares latency CDFs for Copilot and Fast-View-Change for three slowdowns with varying manifestations. The slowdowns are injected on the leader for Fast-View-Change and one of the pilots for Copilot.

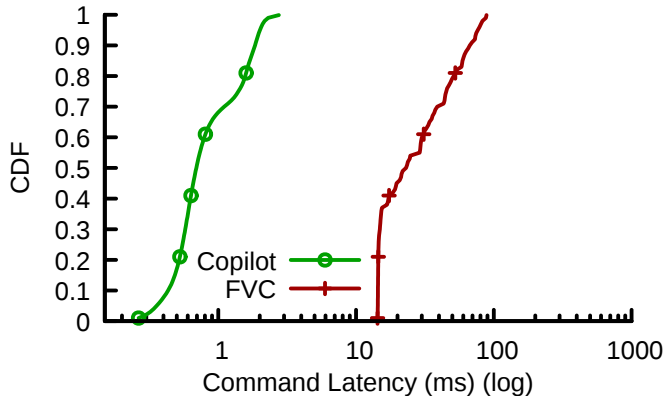
Figure 8.3a considers a slowdown manifested by a slowed processing path for client commands with a fast processing path for messages from replicas. This experiment uses




**(a) Slow for clients.**




**(b) Slow with fast heartbeats.**




**(c) Gradually slow.**

**Figure 8.3: CDF of client command latency for Copilot and Fast-View-Change with slowdowns of varying manifestations. Fast-View-Change's view changes are not triggered in these cases and latency spikes. Copilot's proactive redundancy tolerates these slowdowns and delivers latency similar to the normal case.**

tc to inject 40 ms of delay. Fast-View-Change slows down in this case with 40 ms higher latency than usual because the client command processing path on the leader is slow.

Figure 8.3b shows a CDF of latency when the leader is slow but still quickly replies to heartbeats. This experiment injects 40 ms of delay to non-heartbeat processing directly in the Go process. Fast-View-Change slows down in this case with 80 ms higher latency than usual because the slow leader appears twice on the processing path for client commands.

Figure 8.3c shows a CDF of latency when the leader becomes gradually slower over time. The leader's processing of all messages (including heartbeats) is delayed by $X$ ms, where $X$ starts at 5 ms and increases by 1 ms every 1 second. This delay is directly injected in the Go process. Fast-View-Change slows down in this case with a CDF of latency that mirrors the increasing slowness of its leader.

In each of these slowdowns Fast-View-Change's low view change timeout is not triggered because the replicas are still regularly receiving messages from the leader. Multi-Paxos and EPaxos's view changes similarly would not be triggered. In contrast, Copilot's proactive redundancy tolerates these slowdowns and delivers latency similar to the normal case.

### 8.3.4 Performance Without Slow Replicas

Figure 8.4 shows the throughput and latency of the systems without the thrifty optimization as we increase load. We find that Copilot's throughput is about 8% lower than Multi-Paxos's. Copilot's latency at low/moderate load is similar to Multi-Paxos's; at high load its latency is higher but still low.

EPaxos's best case of 0% conflicts achieves the same peak throughput as Multi-Paxos with slightly higher latency. Under moderate and high conflict rates, EPaxos incurs another round-trip to commit on the slow path more often, and hence has higher latency and lower throughput. EPaxos processes an entire batch on the slow path if any command in the batch has a conflict. With 25% conflicts, almost all batches have at least one command with a

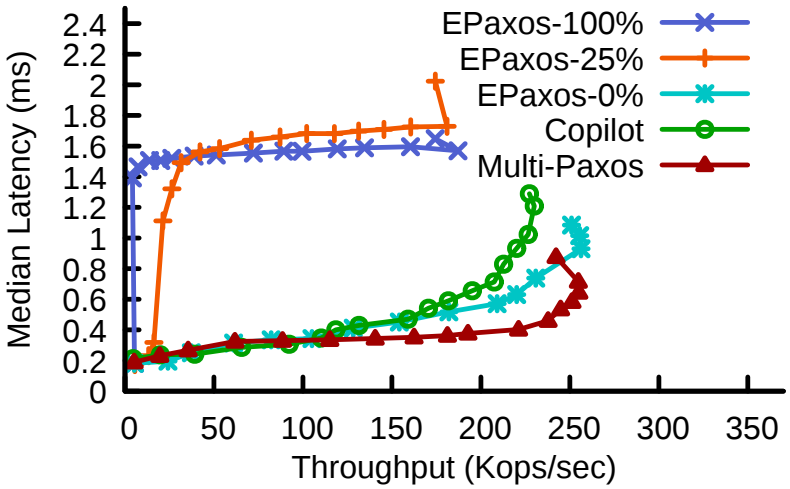



**Figure 8.4: Throughput and latency without the thrifty optimization of the systems when there are no slow replicas.**

conflict and thus almost all are processed on the slow path, resulting in similar performance to 100% conflicts. In contrast, Copilot and Multi-Paxos are not affected because they both totally order all commands.

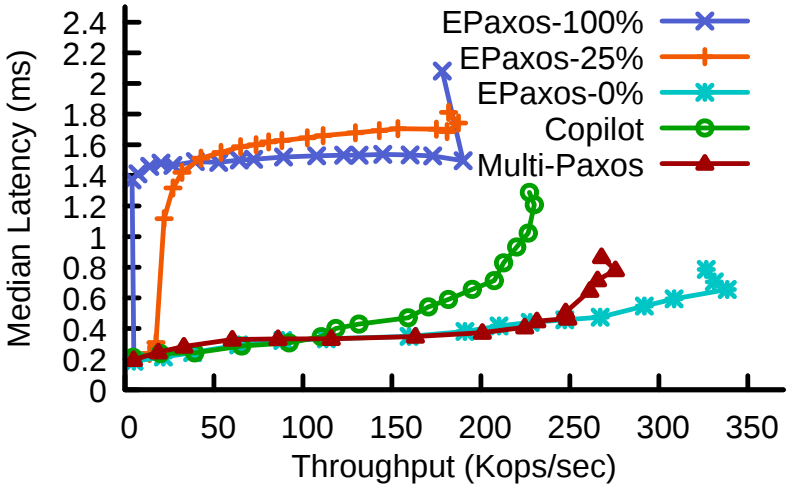



**Figure 8.5: Throughput and latency with the thrifty optimization of the systems when there are no slow replicas.**

Figure 8.5 shows the throughput and latency of all systems with the thrifty optimization as we increase load. Copilot does not use the thrifty optimization because its elimination of redundancy is not slowdown tolerant. Thus, Copilot's performance is the same. Multi-

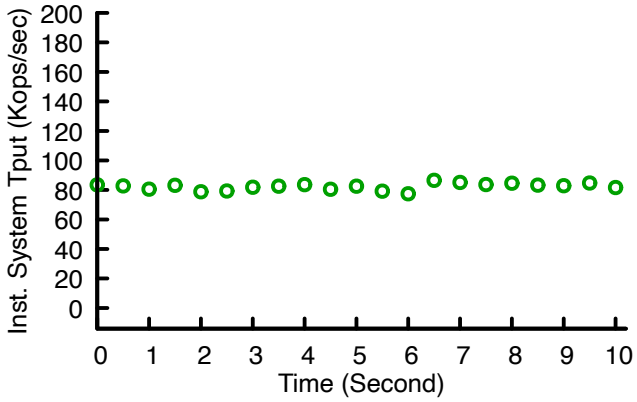
Paxos and EPaxos both see their maximum throughput increase. This makes EPaxos's best case (0% conflicts) provide clearly the highest throughput. With conflicts, however, its throughput is still lower than that of Copilot and Multi-Paxos. The thrifty optimization makes Multi-Paxos provide higher throughput than Copilot by about 35K commands/second, i.e., Copilot achieves 13% lower maximum throughput than Multi-Paxos. Multi-Paxos has higher throughput because it needs to send and receive fewer messages.

Copilot's low latency and high throughput when there are no slow replicas is due to ping-pong batching. The pilots coordinate with each other to ensure that replicas agree with their proposed ordering, allowing them to always commit on the fast path. Committing on the fast path keeps the amount of work each pilot needs to do for its own batches similar to that of a leader in Multi-Paxos. However, a pilot also needs to do the work of a replica for the other pilot's batches. Thus, Copilot's lower but competitive performance with Multi-Paxos is as we expect, because the pilots and leader are the throughput bottlenecks in each system respectively.

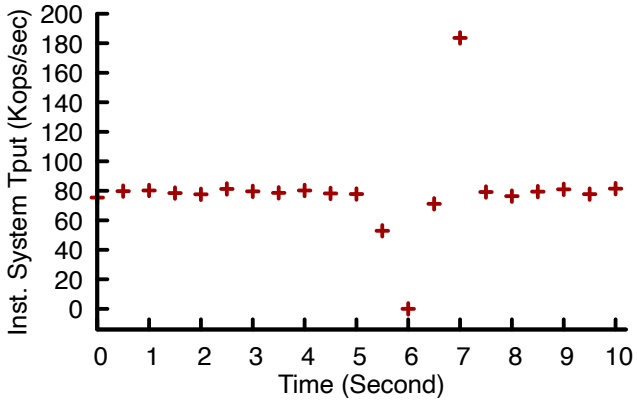
### 8.3.5 Service Availability Under Failures

Copilot can provide 1-slowdown-tolerance and maintain service availability to clients even when a pilot fails since a failed pilot is also a slow pilot. This experiment demonstrates the availability benefit of Copilot and shows that Copilot provides better service availability than Multi-Paxos and EPaxos in the presence of one failure. Specifically, we will show that the throughput and latency of Copilot during one pilot failure is similar to that in the normal case. In contrast, Multi-Paxos and EPaxos incur significant throughput decrease and latency increase during a leader failure (for Multi-Paxos) or any replica failure (for EPaxos). (Note that Copilot and Multi-Paxos can maintain service availability when any non-pilot/non-leader replica fails.)

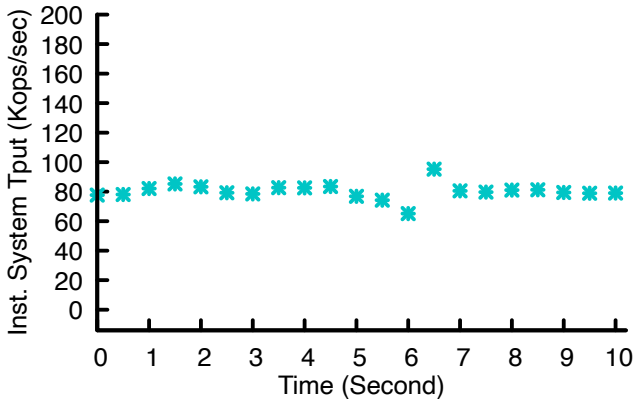
Each client sends commands in an open loop and at approximately the same rate for all systems. The time between successive commands of each client follows an exponential




**(a) Copilot.**

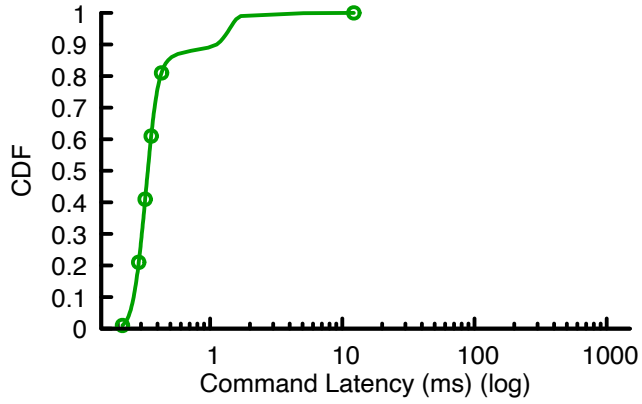



**(b) Multi-Paxos.**

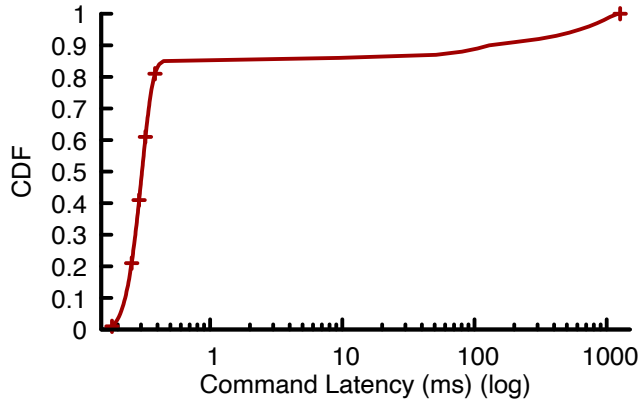



**(c) EPaxos-0%.**

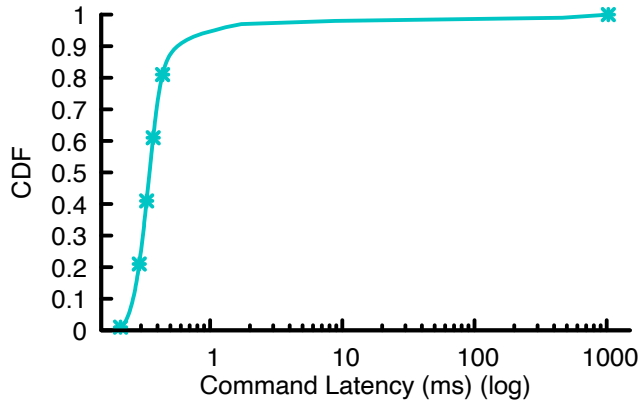
**Figure 8.6: Service availability under failures (throughput).**




**(a) Copilot.**




**(b) Multi-Paxos.**




**(c) EPaxos-0%.**

**Figure 8.7: Service availability under failures (latency).**

distribution with a rate parameter of 2000 commands/second. At this rate, all systems are under medium load. Command retransmission timeout is 100 ms. The failure detection (or leader election timeout) is 1 s for Multi-Paxos and Copilot. When this timeout fires and a failed pilot is detected, Copilot invokes a view-change to replace the failed pilot. If a client suspects that a pilot may have failed, it contacts a random server to ask about the new pilot and then sends its commands to the new pilot. At time 5 seconds, we kill the leader for Multi-Paxos, one of the pilots for Copilot, and a replica for EPaxos.

Figure 8.6 shows each system throughput at every 0.5 second. Figure 8.7 shows the CDFs of latency for each system during this 11-second window. Multi-Paxos system incurs unavailability during the leader failure: starting from the time when the leader failed to the time when the leader failure was detected and a new leader was elected. This results in Multi-Paxos system throughput dropping to 0 commands/second after the time the leader is killed. Shortly after the new leader is elected (at time around 6 seconds), Multi-Paxos system throughput is about twice as high as the steady throughput before the failure. This is as expected since the clients will resend both the timed-out requests during the 1-second leader failure period and the new requests that arrive at the time 6 seconds. After time 6 seconds, Multi-Paxos system throughput becomes steady again. The latency of commands sent during the unavailability is high since Multi-Paxos cannot process and propose these commands until a new leader is elected and the clients redirect their commands to the new leader. Specifically, during the 1-second unavailability period, all commands have a latency of at least 100 ms, the command retransmission timeout; 61.5% of commands have a latency of at least 500 ms; 23.1% of commands have a latency more than 1 s.

In EPaxos, each replica handles the commands for a subset of clients. When a replica fails, all clients that are associated with the failed replica experience service unavailability until they detect the replica failure and redirect their commands to another replica. During the 1-second unavailability period, those clients cannot successfully complete any commands, leading to the system throughput dropping by about 20%. Those clients also

experience high latency of their commands sent during the 1-second unavailability period until they switch to another (alive) replica: all of their commands have a latency of at least 100 ms; 60% of commands have a latency of at least 500 ms; 20% of commands have a latency of at least 1 s. Shortly after those clients switch to different replicas, the system throughput momentarily increases by 20% since those clients send twice as many commands as usual: both the timed-out commands during the last 1-second unavailability period and the new commands that have just arrived. After all retried commands successfully complete, the system throughput becomes steady.

Copilot provides better service availability during one pilot failure due to its proactive redundancy. In Copilot, a client sends its commands to both pilots and both pilots actively propose the commands. Hence, when a pilot fails, the other pilot can continue to propose and complete the commands from the clients. During the pilot failure, Copilot provides a throughput that is close to that in the normal case when there are no pilot failures. During the pilot failure, Copilot still provides low latency for all client commands: 95% of commands have a latency of $< 1.7$ ms, and the median of command latency is $< 1.5$ ms. The maximum command latency during the onset of failure is 12.2 ms, which is controlled by the fast takeover timeout. The fast takeover timeout of 10 ms fires and the alive pilot takes over the ordering work of the failed pilot. Only a small number of commands from the alive pilot which acquire a dependency on uncommitted entries from the failed pilot incur a slight increase in their latency. The new commands proposed in later batches have a latency much lower than 10 ms since they acquire no new (uncommitted) dependency from the failed pilot.

### 8.3.6 Copilot in a Geo-Replicated Setting

Copilot is optimized to achieve good performance in a datacenter setting. Its normal case latency is comparable to Multi-Paxos and EPaxos in a local datacenter setting. In this

experiment, we aim to understand Copilot's normal case latency in a geo-replicated setting where replicas span across geographically dispersed datacenters.

| | VA | CA | SP | LDN | TYO |
|---|---|---|---|---|---|
| CA | 60 | | | | |
| SP | 146 | 194 | | | |
| LDN | 76 | 136 | 214 | | |
| TYO | 162 | 110 | 269 | 233 | |
| SG | 243 | 178 | 333 | 163 | 68 |

**Table 8.1: Round trip latencies in ms between datacenters emulated on Emulab and based on EC2 measurements.**

To emulate a globally-distributed deployment, we choose 5 locations that are spread across the globe: Virginia (VA), California (CA), London (LDN), Tokyo (TYO), and Singapore (SG). The wide-are latencies are based on latencies between EC2 regions [5] and are shown in Table 8.1. We use Linux's tc to emulate wide-are latency between datacenters since machines in the Emulab are physically colocated. Each datacenter location has 1 replica and the same number of clients. We use open-loop clients. The time between successive commands of each client follows an exponential distribution and each client sends commands at approximately the same rate. All systems operate at medium load. We choose the replica at Virginia as the leader for Multi-Paxos since this configuration leads to the lowest latency for Multi-Paxos. Similarly, we choose the replicas at Virginia and California as the two pilots since this configuration leads to the lowest latency for Copilot.

Figure 8.8 shows the CDFs of command latency of Copilot, Multi-Paxos, and EPaxos. EPaxos with 0% conflicts and Multi-Paxos achieve lower latency than other systems since they can commit commands in 1 round-trip. EPaxos with 0% conflicts provides low latency since it always commits in one round-trip and its fast path quorum is optimal (i.e., the same as the majority quorum) with 5 replicas. Multi-Paxos has higher latency than EPaxos with 0% conflicts for the clients at non-leader locations since non-leader replicas need to forward their client commands to the stable leader, and thus incur an additional round-trip.

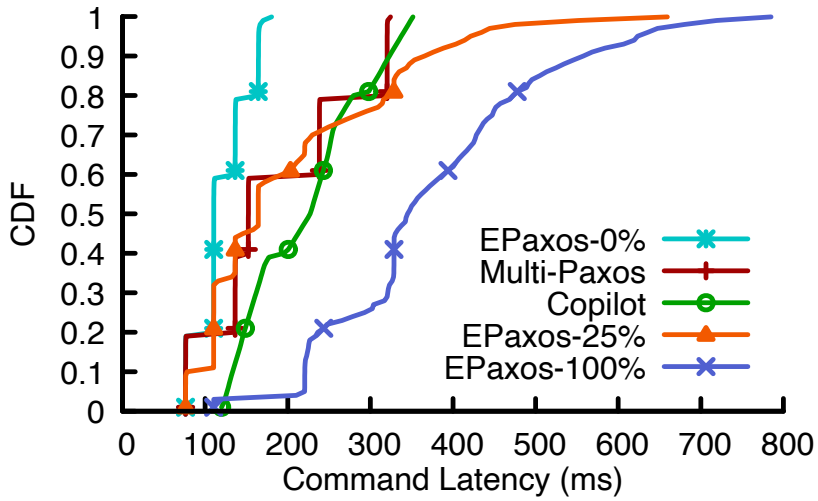



**Figure 8.8: CDF of client command latency for Copilot, Multi-Paxos, and EPaxos in a geo-replicated setting.**

With 25% conflicts, a replica in EPaxos sometimes needs 2 wide-area round-trips to commit its commands and/or has to wait for the commits of dependencies. This results in its latency being higher than EPaxos 0% conflicts. With 100% conflicts, EPaxos has much higher latency than other systems since each replica needs 2 wide-area round-trips to commit its commands most of the time and more likely incurs additional execution delay due to waiting for the commits of dependencies.

Copilot has higher latency than Multi-Paxos. Although the pilots can often commit and execute the commands in 1 round-trip, the additional delay introduced by ping-pong batching waiting time, which is about 1 round-trip between two pilots, increases the latency of Copilot. As a result, the clients at a pilot's location experience higher latency than those at the leader's location in Multi-Paxos. The clients at a non-pilot's location has a latency comparable to those at a non-leader's location in Multi-Paxos since the forwarding latency from a non-pilot replica to a pilot overlaps with the ping-pong batching waiting, masking some or most of it. Optimization of Copilot for a geo-replicated setting is an interesting avenue of future work.

## 8.4 Latent Copilot

Latent Copilot achieves an intermediate tradeoff between Multi-Paxos and Copilot in terms of throughput and slowdown tolerance. It trades off some proactiveness for better performance by operating with one active pilot, which actively proposes commands, and one latent pilot, which proposes commands only when it suspects the other pilot is slow. This evaluation demonstrates the benefit and quantifies the overhead of Latent Copilot. Specifically, it asks:

§8.4.1 Can Latent Copilot tolerate transient slowdowns?

§8.4.2 Can Latent Copilot tolerate slowdowns of varying severity?

§8.4.3 How does the throughput and latency of Latent Copilot compare to Multi-Paxos and Copilot?

### 8.4.1 Transient Slowdowns

Figure 8.9 shows the latency of client commands for Latent Copilot as transient slowdowns of increasing severity are injected. The transient slowdowns are injected every second starting at time 2 seconds. The severity and duration of the slowdowns in order are 0.5 ms, 1 ms, 2 ms, 5 ms, 10 ms, 20 ms, 40 ms, and 80 ms. The slowdowns are injected on the active pilot by stopping all processing for the specified length inside the Go processes.

*Latent Copilot tolerates transient slowdowns.* The latency for Latent Copilot remains low and close to its latency when there are no slowdowns. For pauses that are $< 10$ ms, the timeout value that triggers the latent pilot to propose uncommitted commands, Latent Copilot simply waits out the pause. This does not mask the slowdown and does show up in client command latency, but its magnitude is small enough that latency remains similar. For pauses that are $\geq 10$ ms, the latent pilot detects the commands have not been committed for at least 10 ms. It then proposes these uncommitted commands and completes the ordering work for any uncommitted dependencies from the slow active pilot. This keeps

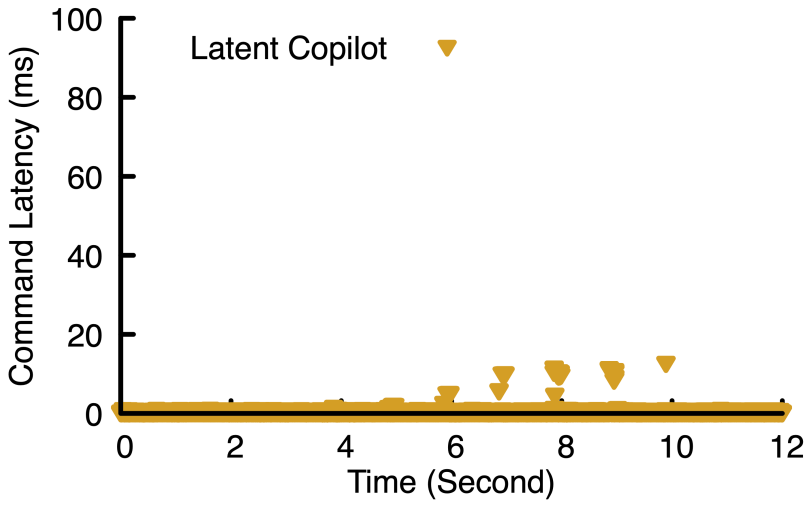



**Figure 8.9: Client command latency for Latent Copilot with transient slowdowns. Transient slowdowns are injected every second starting at time 2 seconds on the active pilot. The severity and duration of the slowdowns in order are 0.5 ms, 1 ms, 2 ms, 5 ms, 10 ms, 20 ms, 40 ms, and 80 ms. Latent Copilot tolerates the transient slowdowns because the latent pilot uses the timeout and fast takeover mechanisms to limit the maximum latency.**

the command latency close to the timeout value of 10 ms. For instance, the maximum command latency is 12.7 ms for a 40 ms pause. The maximum latency during the onset of a slowdown is controlled by the timeout value that the latent pilot uses to detect and propose uncommitted commands.

### 8.4.2 Slowdowns of Varying Severity

Figure 8.10 shows the CDF of latency for Latent Copilot in the normal case (0 slowdown) and with slowdowns of varying severity that last for the duration of the experiment. A slowdown of the given severity is injected on the active pilot at the beginning of the experiment and remains on this pilot for the duration of the experiment. The severity ranges from 0.5 ms to 40 ms.

For the severity $< 5$ ms, Latent Copilot does not mask slowdown and the slowdown does show up in the latency: the latency of client commands in Latent Copilot is proportional to $2\times$ the severity of the slowdown. The magnitude of the slowdown is small enough

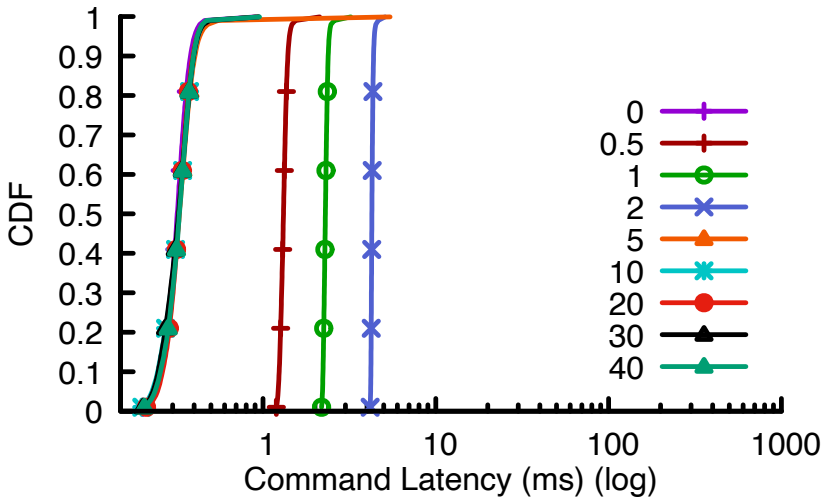



**Figure 8.10: CDF of command latency for Latent Copilot in the normal case (0) and with slowdowns of varying severity in ms. Slowdowns are injected for the duration of the experiment.**

that the client command latency remains similar to the normal case (0 slowdown). The slowdown affects the latency twice since the active pilot appears twice on the path for client commands: the message path is client-to-active-pilot-to-replicas-to-active-pilot-to-client. With low severity, the fast latent pilot still receives the Commit messages from the slow active pilot within the timeout of 10 ms since the time it receives the client commands. Hence, the fast latent pilot does not propose commands and enter active mode.

For the slowdown severity $\geq 5$ ms, Latent Copilot can mask the slowdown and provide a latency that is close to the normal case. When a slowdown of severity $\geq 5$ ms is injected on the active pilot, the latent pilot will quickly propose the uncommitted commands it has received from the clients when the timeout of 10 ms fires because it has not received the Commit messages for these commands from the active pilot. When a sufficient number of new proposals from the latent pilot suggest that the current active pilot is slow, the latent pilot enters active mode and actively proposes new client commands. The slow active pilot switches to latent mode once it receives sufficient number of Commit messages that contain new commands from the other pilot. The fast pilot remains active for the rest of the experiment. Latent Copilot is able to mask slowdown from the slow pilot, which is

now latent, because the fast active pilot actively proposes client commands and can always gather a majority of replies from fast replicas to make fast progress. Hence, the CDFs of command latency with the slowdowns of severity $\geq 5$ ms are similar to the normal case.

### 8.4.3 Performance Without Slow Replicas

Figure 8.11 shows the throughput and latency of Latent Copilot, Copilot, and Multi-Paxos without the thrifty optimization as we increase load. We find that Latent Copilot's throughput is about 3% lower than Multi-Paxos's and about 5% higher than Copilot. Latent Copilot's latency is similar to Multi-Paxos.

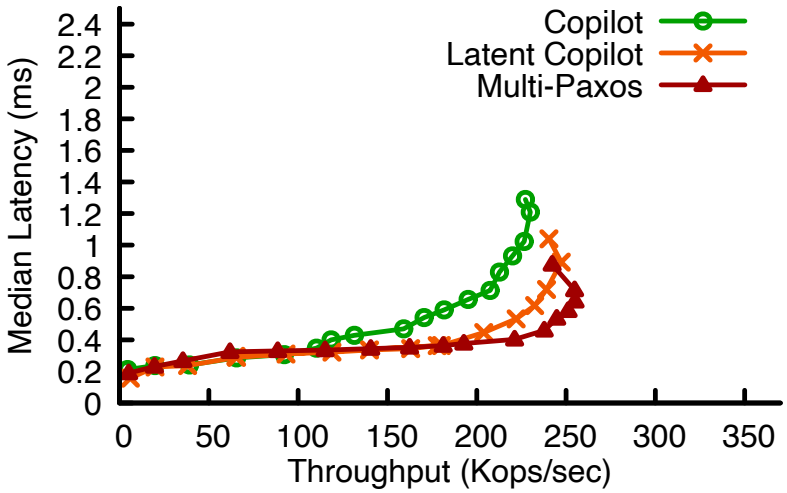



**Figure 8.11: Throughput and latency without the thrifty optimization of Copilot, Latent Copilot, and Multi-Paxos when there are no slow replicas.**

Latent Copilot's low latency and high throughput when there are no slow replicas is due to its operating with one active pilot that actively proposes client commands at any time. Since the active pilot is fast and commits the commands quickly, the latent pilot does not need to propose the same commands it has received from the clients. This allows the active pilot to always commit on the fast path because replicas always agree with the ordering proposed by the active pilot. Committing on the fast path helps Latent Copilot achieve low latency and keep the amount of work the active pilot needs to do for its

batches similar to that of a leader in Multi-Paxos. Having the latent pilot not propose the commands in the normal case helps Latent Copilot avoid the overhead that Copilot incurs: the active pilot in Latent Copilot does not need to do the work of a replica for the other pilot's batches. Thus, Latent Copilot's throughput is higher than Copilot's and comparable to Multi-Paxos's. Latent Copilot's peak throughput is slightly lower than Multi-Paxos's for the following reasons: (1) an active pilot in Latent Copilot incurs some overhead of updating its local state that keeps track the committed commands during the Commit phase, and (2) the ordering messages of Latent Copilot is slightly larger than Multi-Paxos since they carry some additional metadata (e.g., dependency-related metadata, a view ID, etc.), which leads to some overhead during the message serialization. Latent Copilot, Copilot, and Multi-Paxos have competitive performance as we expect since the active pilot, the two pilots, and the leader are the throughput bottlenecks in each system respectively.

# Chapter 9

# Related Work

This section reviews related work. To the best of our knowledge, all previous consensus protocols are not 1-slowdown-tolerant. Copilot's primary distinction is thus being the first 1-slowdown-tolerant consensus protocol. We review related work in consensus protocols, Byzantine consensus protocols, and slowdown cascades.

**Consensus protocols.** There is a growing body of consensus protocols that started with Paxos [29] and Viewstamped Replication [43]. New consensus protocols improve latency and/or throughput on these baselines [32, 31, 37, 23, 4, 55, 47, 34]. Others are designed to be more understandable [45]. SDPaxos [55] includes a throughput-based detection mechanism, similar to that of Aardvark (§2.3), that triggers a view-change for its sequencer that orders commands. Gryff unifies shared registers and consensus [8]. Its unproxied shared register operations are slowdown tolerant while its consensus operations are not. If the network ordering from NOPaxos [34] could be made slowdown tolerant, it could be used to eliminate the need for ping-pong batching to keep the pilots on the fast path in the normal case. To the best of our knowledge, none of these protocols are 1-slowdown-tolerant.

**Paxos, EPaxos, Mencius.** We drew inspiration in our design from Paxos, EPaxos, and Mencius. Our fast takeover protocol uses the classic 2-phase Paxos [29] on a slow pilot's

log to enable a fast pilot to complete its ordering work. Our ordering protocol is influenced by EPaxos's ordering protocol [41]. It draws its use of dependencies and a multi-round ordering protocol with a fast path from EPaxos. Copilot's ordering differs because it orders the same commands twice, totally orders all commands, has only one dependency per entry, and includes fast takeovers. Mencius has all replicas work collaboratively to avoid doing redundant work or conflicting with each other [37]. Our ping-pong batching is inspired by Mencius and lets our pilots avoid conflicting with each other.

**Byzantine consensus protocols.** There is also a vast body of literature on Byzantine consensus protocols [11, 28, 3, 49, 20, 53, 13]. These protocols tolerate Byzantine faults, which Copilot does not. Most use the approach that PBFT introduced for practical systems of having multiple replicas execute a command and reply to the client. Copilot's use of both pilots to execute and reply to clients is inspired by this design.

**Aardvark.** Aardvark focuses on ensuring reliable minimum performance in BFT environments [3]. It employs two mechanisms to detect slowdowns in the leader: a gradually increasing lower bound on the leader's throughput, and an inter-batch heartbeat timer that ensures the leader is proposing new batches quickly enough. Both mechanisms trigger view changes to rotate the leader among replicas. As explained in §2.3, these mechanisms are detection based and hence provide only partial slowdown tolerance for Aardvark, because each limits the effect of a subset of slowdowns and incurs view changes that themselves cause slowdowns (§2.3). Copilot, in contrast, provides 1-slowdown-tolerance, because it *proactively* provides an alternative path for processing at all times, including during a view change to replace a slow pilot.

Note that Aardvark is designed for a Byzantine environment where replicas can be malicious. Copilot assumes nodes follow its protocol and thus would not work in a malicious setting. Focusing on crash faults allows Copilot to use techniques like fast takeovers and ping-pong batching to provide slowdown tolerance with good performance, which would

be vulnerable to manipulation by a Byzantine replica. An interesting question to explore is whether mechanisms from Copilot and Aardvark can be combined to provide 1-slowdown-tolerance in a Byzantine environment.

**Visigoth fault tolerance (VFT).** The Visigoth fault model is a hybrid fault model that fills in the spectrum between crash faults and Byzantine faults, and between synchronous and asynchronous models to enable the design of resource-efficient replication protocols for stateful services in datacenter environments [46]. It provides two knobs that can be set independently: the threshold on the number of slow but correct processes, and the threshold on the number of correlated faulty (malicious) processes. This enables the VFT protocol to be able to handle some Byzantine faults while reducing replication cost (e.g., the number of replicas), which is the focus of VFT. In contrast, our work assumes crash faults and is focused on providing slowdown-tolerance for RSMs in the presence of slow machines. The VFT protocol is not 1-slowdown-tolerant because receiving the client commands and broadcasting them are only done by the leader: if the leader is slow, it slows these stages. Copilot, in contrast, is 1-slowdown-tolerant by providing an alternative path at every stage of the processing.

**Slowdown cascades.** Occult is a scalable, geo-replicated data store that is immune to slowdown cascades [39]. Slowdown cascades occur when one slow shard of a scalable system cascades and affects other shards. They are a mostly orthogonal problem to slowdown tolerance because they are about preventing slowdowns of one part (shard) of a system from affecting other parts (shards) that do different work. Slowdown tolerance, in contrast, is about preventing slowdowns *within* an RSM, which may be one part (shard) of a larger system. Slowdown tolerance within shards decreases the likelihood of slowdown cascades. But they are mostly orthogonal, because cascades can still occur if there are more than $s$ slowdowns within a shard.

**Optimizing consensus protocols in a geo-replicated setting.** Domino [54] is a RSM protocol that aims to reduce commit latency in a geo-replicated setting by using network measurements. Domino runs Dominio Fast Paxos (DFP), a variant of Fast Paxos [32], and Domino Mencius (DM), a variant of Mencius [37], in different consensus instances. In Domino, a client periodically measures the network latency to replicas and collects the network latency among replicas. It then uses the network latency data to choose the consensus protocol that leads to lower commit latency for its requests. Domino pre-partitions its log of requests into two subsets each of which is managed by DFP and DM, respectively. Each log position is associated with a nanosecond-level timestamp. When a DFP client or a DM replica proposes a request, it assigns a future timestamp which is the estimated arrival time of the request to a quorum of replicas. Domino then tries to get the request committed at the log position identified by the timestamp. Assigning a nanosecond-level timestamp based on expected arrival time aims to reduce the conflicts in DFP and enables Domino to commit on a fast path. Since the logs of DFP and DM are interleaved in Domino's request log and neither DFP nor DM are slowdown-tolerant, Domino is not slowdown-tolerant either. Specifically, if the coordinator replica in DFP is slow at sending out a commit of an entry, it will slow down the execution of other entries that follow this entry in the log. Similarly, if any replica in DM is slow, it will increase the request latency of its clients and other DM replicas' clients. In contrast, Copilot provides 1-slowdown-tolerance for RSMs and is optimized for a local datacenter setting. Optimizing Copilot's normal-case latency in a geo-replicated setting by leveraging the network latency and topology is an interesting venue of future work.

Using synchronized clocks and information about network latency among replicas can reduce conflicts in a geo-replicated setting. By imposing intentional delays on message processing at replicas, this technique [51] demonstrates that it can reduce the conflicts in EPaxos [41]. Specifically, when a replica in EPaxos sends a PreAccept message, it also includes a timestamp at which the receiving replicas should process it. The timestamp is

the sum of the time when the message is sent and the one-way delay from this replica to the furthest replica. Since all replicas process the same PreAccept message at the same time, and the PreAccept messages are processed in order of their timestamps, the replicas often agree on the orderings and avoid conflicts. In contrast, Copilot's ping-pong batching tries to coordinate the pilots to avoid conflicts. Thus, the replicas receive the compatible orderings, and Copilot always commits on a fast path in the normal case. Further reducing the latency of Copilot in a geo-replicated setting by using the synchronized clocks and imposing intentional delays at replicas is an interesting venue of future work.

# Chapter 10

# Conclusion

At large-scale, it is common for some machines to be slow. Prior to our work, no consensus protocol is slowdown-tolerant: a single slow machine can significantly increase the latency of these protocols and the latency of the RSMs that they coordinate. It is important for RSMs to be slowdown-tolerant because a service that does not respond in time is not meaningfully available.

Our dissertation makes several contributions toward achieving slowdown-tolerant RSMs that continue to provide normal performance despite the presence of slow replicas. We define $s$-slowdown-tolerance and identify why existing consensus protocols are not slowdown-tolerant. We design, implement, and evaluate two 1-slowdown-tolerant consensus protocols, Copilot and Latent Copilot, which provide slowdown-tolerance despite the presence of 1 slow replica. We describe the design of an $s$-slowdown-tolerant consensus protocol that can tolerate $s$ slow replicas for $s \geq 1$.

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 that allows a fast pilot to safely complete the work of a slow pilot to provide slowdown tolerance. It has two optimizations—ping-pong batching and null dependency elimination—that improve its performance when there are 0

and 1 slow pilots respectively. 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.

Latent Copilot replication is another design and implementation of a 1-slowdown-tolerant consensus protocol. It achieves an intermediate tradeoff between Multi-Paxos and Copilot in terms of throughput and slowdown tolerance. Latent Copilot operates with one active pilot, which actively proposes commands, and one latent pilot, which proposes commands only when it suspects the other pilot is slow. A pilot uses the additional metadata embedded in the ordering protocol messages to learn about the other pilot's progress and determine if it should become active or latent. Our evaluation shows that Latent Copilot is slowdown-tolerant despite the presence of 1 slow replica and its performance without slow replicas is comparable to Multi-Paxos.

Our design of an *s*-slowdown-tolerant protocol is the next crucial step toward achieving *s*-slowdown-tolerant RSMs. We generalize Copilot replication's design—including its ordering protocol, execution protocol, fast takeover, ping-pong batching, and null dependency elimination—to achieve a consensus protocol that is slowdown-tolerant despite the presence of *s* slow replicas.

# Appendix A

# Pseudocode

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

## A.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 (Appendix A.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 A.1 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

```
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
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 replicas
31   wait for at least f other AcceptOk replies
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 replicas
38 }
```

**Figure A.1: Pseudocode for Copilot's ordering protocol at a pilot.**

```
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 A.2: Pseudocode for handling the ordering protocol's messages at a replica.**

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/Replica
// max dependency at each pilot's log; used for checking cycles
maxDep := map[{P,P'} → int]
currPilot := P
while !stop { // execution thread loops until RSM stops
  for e := executedUpTo[currPilot]+1; e ≤ latestEntry[currPilot]; e++ {
    entry = Logs[currPilot][e]
    // Case 0: Current entry is not committed
    if entry.status != COMMITTED {
      // switch to the other pilot's log
      currPilot = (currPilot == P) ? P' : P; break
    }

    // update max dependency among entries of currPilot's log
    maxDep[currPilot] = max(maxDep[currPilot], entry.dep.e)
    depPilot = entry.dep.pid; e' = entry.dep.e
    depEntry = Logs[depPilot][e']

    // Case 1: Entry is no-op, or has no dep, or dep is executed
    if entry.cmds == no-op || e' == ∅ || depEntry.status == EXECUTED {
      executeCmds(currPilot, e); executedUpTo[currPilot] = e; continue
    }

    // Case 2: A cycle exists and currPilot has higher priority
    // (if it has lower priority, the cycle will be handled
    // when execution switches to the other pilot's log)
    hasCycle := maxDep[depPilot] ≥ e
    if hasCycle && hasHighPriority(currPilot) {
      executeCmds(currPilot, e); executedUpTo[currPilot] = e; continue
    }

    // Case 3: Null dependency elimination succeeds
    if entry.nullDepSafe && views[depPilot].view.vid ≥ entry.depViewId
        && checkDepsNullable(depPilot, e') {
      executeCmds(currPilot, e); executedUpTo[currPilot] = e; continue
    }

    // Case 4: Pilot invokes fast takeover if takeover-timeout fires
    if this.isPilot && time.Since(entry.commitTime) ≥ TAKEOVER_TO {
      for i := committedUpTo[depPilot] + 1; i ≤ e'; i++ {
        if Logs[depPilot][i].status != COMMITTED {
          takeover(depPilot, i)
        }
      }
    }
    // switch to the other pilot's log
    currPilot = (currPilot == P) ? P' : P; break
  }
}
```

**Figure A.3: Pseudocode for Copilot's execution protocol.**

```
1 Pilot/Replica
2 // mapping command ID to true/false to keep track executed commands
3 execMap := map[int → bool]
4 func checkDepsNullable(P, e) {
5   for i := executedUpTo[P] + 1; i ≤ e; i++ {
6     if Logs[P][i] == nil || !execMap[Logs[P][i].cmds.id] {
7       return false
8     }
9   }
10  return true
11 }
12
13 func executeCmds(P, e) {
14  foreach cmd in Logs[P][e].cmds {
15    if !execMap[cmd.id] {
16      execute(cmd) // execute the command and update the RSM's state
17      execMap[cmd.id] = true
18      if this.isPilot { send reply to client }
19    }
20  }
21 }
```

**Figure A.4: Pseudocode for helper functions called by Copilot's execution protocol.**

pilot), then the fast takeover and/or view change protocols will be initiated to resolve these entries (§A.3).

Figure A.2 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 §3.2 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 A.3 shows the execution protocol for a replica (and a pilot acting as a replica), and Figure A.4 shows the helper functions that are called by the execution protocol. 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 28). (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. 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 15–17, Figure A.4)—recall that each command appears twice in the logs because they are processed by both 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 47).

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 39–45). This ensures that Copilot remains 1-slowdown-tolerant, since otherwise the execution protocol would be blocked waiting on these dependencies to commit.

## A.2 Fast Takeover

```
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
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 A.5: 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 A.6 and A.8.**

Figure A.5 shows the pseudocode for the fast takeover procedure, which is continued in Figures A.6, A.7, and A.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 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 A.6 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 simulta-

```
1  Pilot
2  func choose_value(P, i,  Q) {
3    Pi_resolved, Pi_init_cmds, Pi_init_dep, P'_uncommitted_entries :=
4      choose_value_common(P, i, Q)
5    if Pi_resolved { return }
6    foreach k in P'_uncommitted_entries {
7      // P.i and P'.k are both unresolved. Prepare both
8      Q_{i,k} = simultaneous_prepare(P, i, P', k)
9      // Check again if P.i can be resolved with the new quorum Q_{i,k}
10     Pi_resolved, _, _, _ = choose_value_common(P, i, Q_{i,k}[P])
11     if Pi_resolved { return }
12
13     P'k_resolved, _, P'k_init_dep, _ :=
14       choose_value_common(P', k, Q_{i,k}[P'])
15     if P'k_resolved {
16       if Logs[P'][k].cmds == no-op || Logs[P'][k].dep ≥ P.i {
17         continue
18       }
19       accept_and_commit(P, i, cmds=no-op, dep=∅)
20       return
21     }
22     if P'k_init_dep ≥ P.i { continue } // P.i and P'.k do not conflict
23
24     // Quorum Q_{i,k} does not contain either of the original pilots
25     // P.i and P'.k conflict; both have < f and ≥ ⌊(f+1)/2⌋ fast accepts
26     if P.i has > ⌊(f+1)/2⌋ FAST_ACCEPTED in Q_{i,k} {
27       accept_and_commit(P', k, cmds=no-op, dep=∅)
28       continue
29     }
30     if P'.k has > ⌊(f+1)/2⌋ FAST_ACCEPTED in Q_{i,k} {
31       accept_and_commit(P, i, cmds=no-op, dep=∅)
32       return
33     }
34     accept_and_commit(P, i, cmds=no-op, dep=∅)
35     accept_and_commit(P', k, cmds=no-op, dep=∅)
36     return
37   } // end of for loop
38   // each concurrent P'.k is either no-op or has dependency ≥ P.i
39   // safe to commit P.i with its initial value
40   accept_and_commit(P, i, cmds=Pi_init_cmds, dep=Pi_init_dep)
41 }
```

**Figure A.6: Pseudocode for `choose_value` procedure.**

```
1  Pilot
2  func simultaneous_prepare(P, i, P', j) {
3    // get higher ballot numbers and prepare both P.i and P'.j
4    Logs[P][i].ballot = make_higher_ballot(Logs[P][i].ballot)
5    Logs[P'][j].ballot = make_higher_ballot(Logs[P'][j].ballot)
6    Logs[P][i].status = PREPARED
7    Logs[P'][j].status = PREPARED
8    send SimultaneousPrepare(P, i, Logs[P][i].ballot,
9                             P', j, Logs[P'][j].ballot) to all replicas
10   wait for at least f other replies
11   Q := map[{P, P'} → []]
12   if ≥ f+1 replies (incl. from itself) are SimultaneousPrepareOk {
13     foreach SimultaneousPrepareOk reply {
14       Q[P].add(reply[P]); Q[P'].add(reply[P'])
15     }
16   } else {
17     # retry prepare phase with higher ballot numbers
18   }
19   return Q
20 }
21
22 Replica
23 on Receiving SimultaneousPrepare(P, i, P_ballot, P', j, P'_ballot)
24   Pi := Logs[P][i]
25   P'j := Logs[P'][j]
26   if P_ballot > Pi.ballot && P'_ballot > P'j.ballot {
27     Pi.ballot = P_ballot
28     P'j.ballot = P'_ballot
29     Pi.status = PREPARED
30     P'j.status = PREPARED
31     send SimultaneousPrepareOk(
32           map[P → (Pi.status, Pi.cmds, Pi.dep, Pi.accept_ballot),
33               P' → (P'j.status, P'j.cmds, P'j.dep, P'j.accept_ballot)]
34          )
35   } else {
36     send SimultaneousPrepareReject(Pi.ballot, P'j.ballot)
37   }
```

**Figure A.7: Pseudocode for `SimultaneousPrepare` phase.**

neously 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 8–36). (Figure A.7 shows the pseudocode for the SimultaneousPrepare phase.) If the value of $P.i$ can be resolved with the new quorum $Q_{i,k}$, `choose_value` returns (lines 10–11). 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 15–17, 22, and 40). 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 19). 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 26–29). 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 30–33).

**`choose_value_common` sub-procedure:** Figure A.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 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.

```
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_init_cmds := nil; Pi_init_dep := nil
6    if ∃ a reply r with r.status == FAST_ACCEPTED {
7      Pi_init_cmds = r.cmds; Pi_init_dep = r.dep
8    }
9    if ∃ a reply r with r.status == COMMITTED/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
16                 && Pi_proposer's reply ∉ Q) { // Case 3
17     accept_and_commit(P, i, cmds=Pi_init_cmds, dep=Pi_init_dep)
18   } else if Pi_proposer's reply ∈ Q ||
19     |reply r with r.status == FAST_ACCEPTED| < ⌊(f+1)/2⌋ { // Case 4
20     accept_and_commit(P, i, cmds=no-op, dep=∅)
21   } else {
22   // Case 5: ⌊(f+1)/2⌋ ≤ |reply r with r.status == FAST_ACCEPTED| < f
23   // and Pi_proposer's reply ∉ Q
24     if |S| < f+1 {
25       send FastAccept(Pi_init_cmds, P, i, Pi_init_dep) to replicas
26       in Q\S; wait for at least f+1-|S| replies; add FAST_ACCEPTED/
27       NOT_ACCEPTED replies to S; recheck cases 1&3; retry if |S|<f+1
28     }
29     max_suggested_dep := max{r.dep.e|r ∈ S ∧ r.status == NOT_ACCEPTED}
30     P'_uncommitted_entries := []
31     // examine each concurrent entry to determine P.i
32     for e' := Pi_init_dep.e + 1; e' ≤ max_suggested_dep; e'++ {
33       if Logs[P'][e'].status == COMMITTED/EXECUTED {
34         if Logs[P'][e'].cmds == no-op || Logs[P'][e'].dep ≥ P.i {
35           continue // found compatible entry
36         }
37         // found incompatible entry; P.i could not commit on fast path
38         accept_and_commit(P, i, cmds=no-op, dep=∅)
39         return true, nil, nil, nil
40       } else { P'_uncommitted_entries.add(e') } // uncommitted entry
41     } // end of for loop
42     if len(P'_uncommitted_entries) > 0 {
43       // need to resolve these entries to resolve P.i
44       return false, Pi_init_cmds, Pi_init_dep, P'_uncommitted_entries
45     }
46     // concurrent entries are compatible with P.i's initial value
47     accept_and_commit(P, i, cmds=Pi_init_cmds, dep=Pi_init_dep)
48   }
49   return true, nil, nil, nil
50 }
```

**Figure A.8: Pseudocode for `choose_value_common` sub-procedure.**

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 implies the proposing pilot of this value may have committed its commands and initial dependency. Thus, `choose_value_common` tries to get the same value committed. It is safe to commit this entry with its initial dependency because it has passed the compatibility check at a majority of replicas. In turn, this ensures that any incompatible entries from the other pilot's log will be ordered after this entry.

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 $|S| \geq (f+1)$, we examine the concurrent entries from the other pilot that may have ordering conflicts with *P.i* (lines 29–47). 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 concurrent entries from the other pilot commit with a no-op or have a dependency $\geq$ *P.i*, we can commit *P.i* with its initial dependency since they 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 24–28). 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 randomized exponential backoff before retrying the takeover.)

## A.3 View Change (Pilot Election)

Figure A.9 and Figure A.10 show 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 [38], 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

```
View Manager (any replica)
vs := views[P] // current view state for pilot P at this replica
func startViewChange(P) {
  vs.mode = MANAGER
  vs.proposed_vid = make_higher_viewid(vs.proposed_vid)
  send ViewChange(vs.view, vs.proposed_vid) to all replicas
  wait for at least f other replies
  if ≥ f+1 replies (including from itself) are ViewChangeOk {
    acceptView()
  } else { # retry startViewChange() with a higher proposed_vid }
}

func acceptView() {
  if ∃ a ViewChangeOk reply r with r.accepted_view != nil {
    // some replicas have already accepted a view
    // select r with the highest accepted view ID
    // use r's accepted view as the new view
    vs.accepted_view = r.accepted_view
    vs.accepted_latest_entry = r.accepted_latest_entry
  } else {
    // no existing accepted views
    // create a new view with this replica as the pilot
    vs.accepted_view = form_view(vs.proposed_vid, my_replica_id)
    vs.accepted_latest_entry = latestEntry[P]
    foreach reply r in ViewChangeOk replies {
      vs.accepted_latest_entry = max(vs.accepted_latest_entry,
                                     r.latest_entry)
    }
  }
  send AcceptView(vs.accepted_view,
                  vs.accepted_latest_entry) to all replicas
  wait for at least f other replies
  if ≥ f+1 replies (including from itself) are AcceptViewOk {
    startView()
  } else { # retry startViewChange() with a higher proposed_vid }
}

func startView() {
  vs.view = vs.accepted_view
  latestEntry[P] = vs.accepted_latest_entry
  vs.mode = ACTIVE
  vs.accepted_view = nil
  send StartView(vs.view, latestEntry[P]) to all replicas
  // fill holes in pilot P's log
  for i := latestCommit[P] + 1; i <= latestEntry[P]; i++ {
    takeover(P, i)
  }
}
```

**Figure A.9: Pseudocode for view change protocol (view manager).**

```
1 Replica
2 vs := views[P] // current view state for pilot P at this replica
3 on Receiving ViewChange(manager_curr_view, manager_proposed_vid) {
4   if manager_proposed_vid ≤ vs.proposed_vid
5       || manager_curr_view.vid < vs.view.vid {
6     send ViewChangeReject(vs.view, vs.proposed_vid); return
7   }
8   if manager_curr_view.vid > vs.view.vid {
9     # update local view of pilot P to match manager_curr_view
10  }
11  vs.mode = FOLLOWER
12  vs.proposed_vid = manager_proposed_vid
13  send ViewChangeOk(latestCommit[P], latestEntry[P],
14                    vs.accepted_view, vs.accepted_latest_entry)
15 }
16
17 on Receiving AcceptView(view, latest_entry) {
18   if vs.proposed_vid == view.vid {
19     vs.accepted_view = view
20     vs.accepted_latest_entry = latest_entry
21     send AcceptViewOk()
22   } else {
23     send AcceptReject(vs.proposed_vid)
24   }
25 }
```

**Figure A.10: Pseudocode for view change protocol (follower).**

that is higher than the highest view ID it has seen for this pilot (lines 4–5, Figure A.9). 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 10, Figure A.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 4–7, Figure A.10). 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 8–14, Figure A.10). Since there may be concurrent view change 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 13, Figure A.10) 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 18, Figure A.9). In particular, it uses the `accepted_latest_entry` value from this accepted view (line 19, Figure A.9). 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 described in Appendix A.2.

# Appendix B

# Proof of Correctness

We prove that Copilot replication is both safe (§B.1), i.e., it provides linearizability, and live (§B.2), i.e., all client commands eventually complete.

## B.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 $b$ [22]. We begin by proving the real-time ordering requirement (Lemma 1) and then prove the total order requirement (Lemma 9).

**Lemma 1** *If command $\beta$ is proposed by a client after command $\alpha$ is committed, $\beta$ will be executed after $\alpha$.*

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

Assume that $\alpha$ 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 $\alpha$ 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 establish that $\beta$ will be proposed in a log entry that is later $P.i$ at pilot $P$ and in a log entry later than $P'.m$ at pilot $P'$. Since a replica can commit $\alpha$ 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 $\beta$ at time $t_2 > t_0$, $\beta$ 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, $\beta$ 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 $\beta$ appears in both entries

$P.j$ and $P'.n$, at least one appearance of $\alpha$ (which is in $P.i$ in this case) is ordered before *both* appearances of $\beta$.

We now enumerate the possible execution orders between $P.i$ and $P'.m$ and show that $\alpha$ is executed before $\beta$ 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 $\alpha \in Logs[P'.m].cmds$, $\alpha$ is executed first and before $\beta$. (The repeated command $\alpha$ in $P.i$ is ignored due to execution deduplication (§3.3).)

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

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

In all cases, we have shown that $\alpha$ is executed before $\beta$. 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 A.2). 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 A.1).

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 26–27, Figure A.1).

**Case 2:** $P'.j$ commits after the FastAccept phase, and *P.i* commits after the Accept phase (lines 17–21, Figure A.1).

This is similar to case 1.

**Case 3:** Both *P.i* and $P'.j$ commit after the Accept phase (lines 19–21, Figure A.1).

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 26–27, Figure A.1). 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 A.2), 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 A.1).

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*. When the pilot $P$ is slow or failed, the fast takeover is invoked by another pilot—either when a takeover-timeout fires or during a pilot election—to complete the ordering work for the entry *P.i*. We will show that the fast takeover procedure of *P.i* will correctly choose the value of $(cmds, dep)$ and commit it in *P.i*.

When a pilot invokes a fast takeover of *P.i*, it runs the Prepare phase by sending Prepare messages that contain a ballot number higher than any ballots it has seen for *P.i* to all replicas. When it gathers at least $f+1$ PrepareOk replies, it calls `choose_value` procedure to pick the correct value of *P.i* based on this set of PrepareOk replies. Let $Q$ be the quorum of these 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 the path—either

fast path or regular path—on which *P.i* was committed and based on when the pilot *P* failed relatively to the ordering phases.

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

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 the 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 the 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 A.1).

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 the 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 $(cmds, dep)$ is also consistent and 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 A.1).

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 subcases, 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*. for the following reasons. (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 and conflicting 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 following 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 $P.i$. It also satisfies the compatibility check since all entries after $P'.j$ have a dependency $\geq P.i$.)

*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. 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. We need to establish some premises for the two subcases first. (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 9–20, Figure A.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 21, Figure A.8). (4) At least $f+1$ replicas in $Q_{ik}$

have received and replied to the FastAccept for $P.i$ and $P'.k$ (lines 24–28, Figure A.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 no-op or have a dependency $\geq P.i$ because $P'.k$'s initially proposed value has been received by a majority of replicas, each of which either has accepted $P'.k$'s initial dependency or has 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 the 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 (potentially) 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 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 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. This proof assumes that the null dependency elimination optimization is not used. We will later show that the null dependency elimination preserves the total ordering of commands in Lemma 8. (Note that the ping-pong batching optimization does not affect the 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 the entry $P.i_0$ of $P$'s log and up to the 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 execution starts with $P'.j$. By Lemma 3, we know $Logs[P'.j].dep \geq P.i$ 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 can 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 with 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 $P.i$. $P.i$ cannot be executed in this case 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 $P'.j$.

The 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 does not affect the correctness. Hence, we focus on showing the null dependency elimination optimization preserves the correctness of Copilot next. We start with a helper lemma that shows 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 the 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) these 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 the entry $P'.j$ or a no-op–i.e., it will not introduce any new commands.

We now show that if a view change for pilot $P'$ occurs, the newly elected pilot in the new view $V_n > V_m$, which is later than $V_m$, 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 the null dependency optimization—i.e., 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. Examining whether there exists an execution where some entry $> P'.j$ is ordered and executed before $P.i$ leads to two cases.

**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 an 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 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 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 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 the null dependency elimination optimization preserves the total ordering of commands (Lemma 8). Hence, Copilot guarantees the 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.* ■

## B.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.

We start with helper lemmas (Lemmas 11 and 12) that shows the fast takeover procedure and the view-change protocol make progress and eventually complete. 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).

Due to FLP [19], no protocol can be both safe and live in a completely asynchronous setting. Since we provide safety in an asynchronous setting, we need to 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.

Note that the last assumption is only necessary for guaranteeing that Copilot makes *useful* progress and a client eventually gets its command completed. Without this assumption, Copilot can still guarantee liveness—e.g., by committing no-ops—but a client is not always guaranteed to have its command committed.

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

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

We focus on the steps in the fast takeover procedure that may prevent Copilot from making progress. Specifically, we focus on the steps that require sending the messages and waiting for the 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 the 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 the canonical Paxos guarantees liveness under partial synchrony assumptions, these phases in Copilot also make progress by relying on the same assumptions and arguments. (Note that as in Paxos, the dueling proposers problem may happen and prevent the fast takeover procedure from successfully preparing and/or getting a value accepted by a majority of replicas—e.g., because they have promised to another proposal with a higher ballot number. In such cases, Copilot retries and applies the common technique of random exponential backoff to mitigate the problem. The retries cannot repeat indefinitely and the fast takeover procedure will eventually complete its Prepare/Accept phases by relying on the partial synchrony assumptions and arguments that Paxos [30] makes to ensure progress.)

The SimultaneousPrepare phase in Copilot is similar to the Prepare phase and also needs the replies from at least a majority of replicas to make progress. By using the same arguments, the SimultaneousPrepare phase will eventually completes under the partial synchrony assumptions.

When a fast takeover tries to send a FastAccept of the initial value to the replicas that have not received the FastAccept from the failed pilot (in order to have at least $(f+1)$ replicas replying to the FastAccept message) (lines 24–28, Figure A.8), it will succeed if no dueling proposers problem occurs and this step will not be repeated. In the presence of the dueling proposers, it may not be able to get some replicas to fast accept the initial value because they may have promised to another proposal with a higher ballot number in the recovered entry. In such case, the recovering replica may have to retry the fast takeover. With the same arguments about the dueling proposers and the common technique to mitigate the problem we mention earlier, a fast takeover will eventually make progress: either it can get more replicas to receive the FastAccept message or it learns that at least a majority of replicas have replied to the 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 (Appendix A.3) is similar to the view-change protocol described for the canonical Paxos [38].

The `startViewChange` and `acceptView` phases of Copilot's view-change protocol require waiting for at least $f+1$ replies before taking the next step. By our assumptions, at least $f+1$ replicas alive and can communicate with each other. Hence, the view-change protocol will not indefinitely wait for the replies and will eventually make progress. Without any concurrent view managers 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 A.9) and accept its new view (lines 32–33, Figure A.9). Once at least majority of replicas accept the new view, the view manager can inform other replicas to start the new view and a new pilot is successfully elected.

In the presence of concurrent view managers (replicas) that initiate a view-change at the same time, a view manager may not be able to successfully form a new view, which is similar to the dueling proposers problem. In such case, a view manager may have to retry until a new view is formed or a new pilot is elected. However, the retries will not repeat indefinitely by relying on the same partial synchrony assumptions and arguments that Multi-Paxos [38] makes to ensure progress. (The common techniques of random exponential backoff can help increase the likelihood that a view manager eventually succeeds in forming a new view.)

Copilot's view-change protocol uses the fast takeover procedure (Appendix A.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 a client.*

*Proof.* Assume that a client sends its command $\alpha$ to both pilots, and at least one pilot can receive the command $\alpha$. Without loss of generality, assume that the pilot $P$ puts the command $\alpha$ in its log entry $P.i$ and proposes its entry $P.i$. (Note that 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. A view change eventually completes as we have shown in Lemma 12.)

First, we show that the command $\alpha$ will be eventually be committed. Then, we show that the command $\alpha$ will be executed, and a client will receive a reply. To show the command $\alpha$ will be eventually committed, we consider different cases: with and without failures, and network partition.

Consider the failure-free case. A pilot can make progress in its FastAccept phase and Accept phase as long as it receives the 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 $f+1$.) With the same partial synchrony assumptions and arguments that Paxos [30], a pilot is guaranteed to be able to gather the replies from at least a majority of replicas. Thus, a pilot can complete its FastAccept phase and Accept phase (which happens if it does not succeed on the fast path) 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 happens 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 Lemma 11. If the failed pilot has committed a value in $P.i$ before it fails, the fast takeover guarantees the same value, which contains the command

$\alpha$, can ever be committed in *P.i*. If the failed pilot has not committed any value in *P.i* before it fails, the fast takeover may commit a no-op in *P.i*. In such case, the client may not receive a reply for its command $\alpha$ and will resend the command to the pilots (including the newly elected pilot) if it has not received a reply for its command from either pilot within a timeout. At least one pilot should be able to commit its entry, which contains the command $\alpha$, without failing—every newly elected pilot cannot keep failing because we assume that only 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 will eventually detect and initiate a view-change to elect a new pilot, which will be a replica in the majority side of the partition. The client will eventually resend its command to both pilots (including the newly elected pilot) if it has not received a reply for its command $\alpha$ within a timeout. As we have shown above, at least one pilot should be able to commit its entry that contains the command $\alpha$.

Now we have the command $\alpha$ is committed in the entry *P.i*. We show that the command $\alpha$ will eventually be executed by showing either *P.i* is executed or the same command $\alpha$, which appears in some other log entry, has been executed.

Consider the case where execution switches to the *P*'s log and tries to execute *P.i*. (This switching happens when the other pilot's log has no new entries to execute or executing an entry in its log requires first executing some entries that include *P.i* in *P*'s log.) If executing *P.i* requires executing some entries that have not been committed—e.g., because the other pilot has failed—in the other pilot's log, those entries will be resolved when a timeout triggers a fast takeover and/or a view-change. A fast takeover or a 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 $\alpha$ has not been executed, it will be executed for the first time when *P.i* is executed. If the command $\alpha$ has been executed,

execution will avoid executing the command $\alpha$ in *P.i* due to the deduplication (§3.3). (The same command $\alpha$ can appear in an committed entry in the other pilot's log because the client sends its command $\alpha$ to both pilots and the other pilot has successfully committed the command $\alpha$.) In both cases, the command $\alpha$ has been executed.

Consider the case where execution never switches to the *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*, which contains the command $\alpha$, have been executed. In this case, the command $\alpha$ has been executed.

We have shown that the command $\alpha$ will eventually be executed. When it is executed for the first time, Copilot will send the client a reply for its command $\alpha$.

We have shown that Copilot guarantees 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. ■

# Bibliography


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

[2] Phillipe Ajoux, Nathan Bronson, Sanjeev Kumar, Wyatt Lloyd, and Kaushik Veeraraghavan. Challenges to adopting stronger consistency at scale. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[3] Lorenzo Alvisi, Allen Clement, Mike Dahlin, Mirco Marchetti, and Edmund Wong. Making byzantine fault tolerant systems tolerate byzantine faults. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2009.

[4] Balaji Arun, Sebastiano Peluso, Roberto Palmieri, Giuliano Losa, and Binoy Ravindran. Speeding up consensus by chasing fast decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[5] `https://www.cloudping.co/`, 2021.

[6] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

[7] Marc Brooker, Tao Chen, and Fan Ping. Millions of tiny databases. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

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

[9] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[10] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

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

[12] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[13] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *ACM Symposium on Operating System Principles (SOSP)*, 2009.

[14] `https://www.cockroachlabs.com/product/`, 2021.

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

[16] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2), 2013.

[17] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2), 1988.

[18] `https://etcd.io/docs/v3.4.0/tuning/`, 2021.

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

[20] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2010.

[21] Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. Meaningful availability. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

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

[23] Heidi Howard, Dahlia Malkhi, and Alexander Spiegelman. Flexible paxos: Quorum intersection revisited. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[24] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. Capturing and enhancing in situ system observability for failure detection. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[25] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[26] Michael Isard. Autopilot: automatic data center management. *Operating Systems Review*, 41(2), 2007.

[27] Jonathan Kirsch and Yair Amir. Paxos for system builders: An overview. In *ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.

[28] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: Speculative byzantine fault tolerance. In *ACM Symposium on Operating System Principles (SOSP)*, 2007.

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

[30] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32, 2001.

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

[32] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2), October 2006.

[33] Joshua B. Leners, Hao Wu, Wei-Lun Hung, Marcos K. Aguilera, and Michael Walfish. Detecting failures in distributed systems with the falcon spy network. In *ACM Symposium on Operating System Principles (SOSP)*, 2011.

[34] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[35] Barbara Liskov and James Cowling. Viewstamped replication revisited. `http://www.pmg.lcs.mit.edu/papers/vr-revisited.pdf`, 2012.

[36] Chang Lou, Peng Huang, and Scott Smith. Comprehensive and efficient runtime checking in system software through watchdogs. In *ACM SIGOPS Workshop on Hot Topics in Operating Systems (HotOS)*, 2019.

[37] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machines for WANs. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

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

[39] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. I can't believe it's not causal! scalable causal consistency with no slowdown cascades. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[40] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. Turbine: Facebook's Service Management Platform for Stream Processing. In *International Conference on Data Engineering (ICDE)*, 2020.

[41] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *ACM Symposium on Operating System Principles (SOSP)*, 2013.

[42] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. Tolerating slowdowns in replicated state machines using copilots. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2020.

[43] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A general primary copy. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 1988.

[44] Diego Ongaro. *Consensus: Bridging Theory And Practice*. PhD thesis, Stanford University, 2014.

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

[46] Daniel Porto, João Leitão, Cheng Li, Allen Clement, Aniket Kate, Flavio Junqueira, and Rodrigo Rodrigues. Visigoth fault tolerance. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2015.

[47] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr. Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

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

[49] Siddhartha Sen, Wyatt Lloyd, and Michael J Freedman. Prophecy: Using history for high-throughput fault tolerance. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010.

[50] https://azure.microsoft.com/en-gb/support/legal/sla/summary/, 2021.

[51] Sarah Tollman, Seo Jin Park, and John Ousterhout. Epaxos revisited. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2021.

[52] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[53] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. Zz and the art of practical BFT execution. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2011.

[54] Xinan Yan, Linguan Yang, and Bernard Wong. Domino: using network measurements to reduce state machine replication latency in wans. In *International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, 2020.

[55] Hanyu Zhao, Quanlu Zhang, Zhi Yang, Ming Wu, and Yafei Dai. SDPaxos: Building efficient semi-decentralized geo-replicated state machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.