Tolerating Slowdowns in Replicated State Machines using Copilots

Khiem Ngo, *Princeton University;* Siddhartha Sen, *Microsoft Research;*
Wyatt Lloyd, *Princeton University*

https://www.usenix.org/conference/osdi20/presentation/ngo

This paper is included in the Proceedings of the
14th USENIX Symposium on Operating Systems
Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the
14th USENIX Symposium on Operating
Systems Design and Implementation
is sponsored by USENIX

Tolerating Slowdowns in Replicated State Machines using Copilots

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

Abstract

Replicated state machines are linearizable, fault-tolerant groups of replicas that are coordinated using a consensus algorithm. 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.

1 Introduction

Replicated state machines (RSMs) are linearizable, fault-tolerant groups of replicas coordinated by a consensus algorithm [46]. Linearizability gives the RSM the illusion of being a single machine that responds to client commands one by one [21]. 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 [13, 14], cloud storage [6, 9], and service managers [25, 39]. 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 [2, 15]. 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 [5, 6, 20, 48].

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 [1, 3, 23, 24, 32, 35]. 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 [3, 10, 28] or rely on the collaboration of multiple replicas [36, 40]. 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 per-

sisting 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 [28, 37] that are applied independently to the log of each pilot. Copilot combines the logs of the two pilots using mechanisms inspired by EPaxos [40]. 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.

In summary, this work makes the following contributions:

- 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).

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 [46]. 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* [21].

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 1-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 work, we focus on the practical case of 1-slowdown-tolerance. Designing RSMs that are s-slowdown-tolerant for $s > 1$ 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 [26, 28, 29, 37] 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 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 [10], because it provides more redundancy.)

Figure 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.

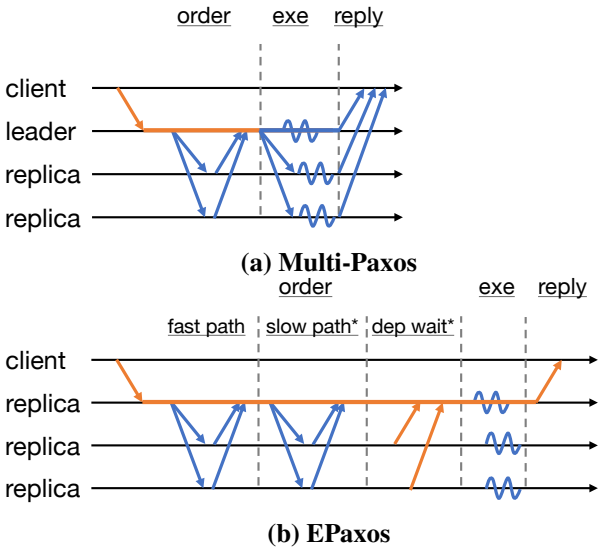


Figure 1: Message diagrams with execution for Multi-Paxos (a) and EPaxos (b). Orange components indicate parts of each protocol that are not slowdown tolerant because they lack redundancy. Blue components indicate parts with redundancy. EPaxos ordering phases that are only sometimes necessary are marked with asterisks (*).

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 [40] 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 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 des-

ignated 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 [51], 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 slowdowns 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 [43, 44] or thousands [8, 14, 17] 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.

3 Design

The core idea behind Copilot is to use two distinguished replicas, the pilot (P) and the copilot (P'), to redundantly process every client command. Figure 2 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.

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

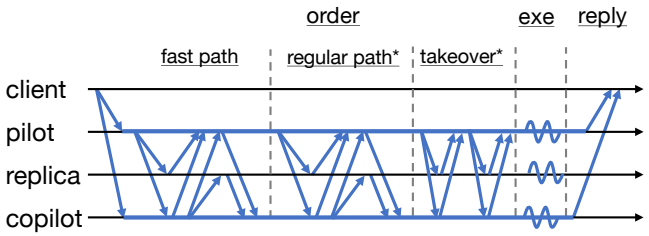


Figure 2: Message diagram with execution for Copilot. All components are in blue because all have the necessary redundancy to avoid any slow replica. Phases that are only sometimes necessary are marked with asterisks (*). The takeover phase only executes when it is necessary to prevent one pilot from 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.

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 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.

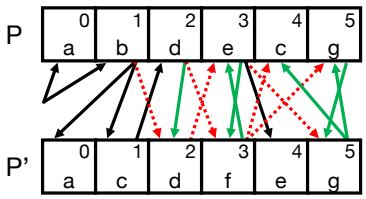
A pair of dependencies are *compatible* if at least one orders its entry after the other. Figure 3a shows examples of compatible and incompatible dependencies. $P'.1$ with dependency $P.1$, and $P.2$ with dependency $P'.1$ are compatible because $P.2$ is ordered after $P'.1$. $P'.3$ with dependency $P.2$ and $P.3$ with dependency $P'.2$ are *incompatible* because neither is ordered after the other. Incompatible dependencies must be avoided because they could lead to replicas with different subsets of the pilot and copilot logs executing entries in different orders, e.g., one replica executing $P.3$ then $P'.3$ and another executing $P'.3$ then $P.3$.

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 $< P.i$.

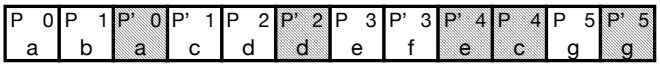
If it has not accepted a later entry, then this same check will prevent the replica from fast accepting any incompatible dependencies from the other pilot in the future. If it has accepted a later entry, but that entry's dependency is on $P.i$ or a later entry, then that entry, call it $P'.k$, is ordered 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.)

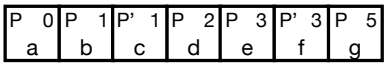
[1] This size is 2/3, 3/5, 5/7, and 6/9 for common RSM sizes.



(a) Dependencies join the two logs.



(b) Combined log with duplicates.



(c) Execution order from the combined log.

Figure 3: 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.

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 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 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 3a. 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 3a. (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 3a.

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) $P.i$ and $P'.j$ are in a cycle and P is the pilot log. 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 [29]. 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 appear in our accompanying technical report [41]. 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 first possible incompatible entry in the other pilot's log. If that entry is not yet committed, the recovering replica recovers that entry by repeating the above procedure, which enables it to safely distinguish between the subcases.

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 [37], 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.

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.

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). Our technical report [41] contains the full proofs; 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 first possible incompatible entry 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 [18], we assume the system is eventually partially-synchronous [16] 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 precede P_i (otherwise there would have been a cycle), which by our inductive assumption has been executed.

If there is a cycle and P has higher priority, Copilot breaks the cycle in favor of P and executes $P.(i+1)$. If 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 $P'.j$ may not have committed. In this case, P eventually does a fast takeover of $P'.j$'s entry. 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.

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.

6 Evaluation

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

§6.3 Can Copilot tolerate transient slowdowns?
§6.4 Can Copilot tolerate slowdowns of varying severity?
§6.5 Can it tolerate slowdowns of varying manifestations?
§6.6 How does the throughput and latency of Copilot compare to existing consensus protocols?

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. Althought this incurs more messages and processing, we find that Copilot's throughput and latency are competitive with Multi-Paxos and EPaxos.

6.1 Implementation and Baseline

We implemented Copilot in Go using the framework of EPaxos [40] 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 [11, 26, 37]. 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.

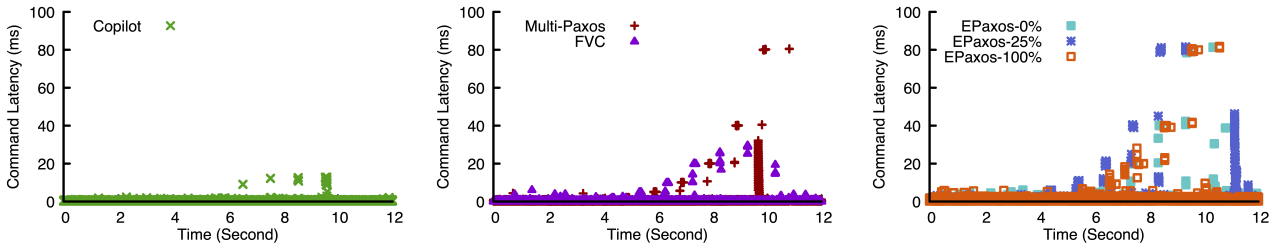
EPaxos and Multi-Paxos can use the *thrifty* optimization to send and receive messages only to the required number of other replicas. The thrifty optimization improves performance by decreasing load on all replicas in EPaxos and the leader in Multi-Paxos. It also harms slowdown-tolerance by eliminating 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 [34, 42]. 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.

6.2 Experimental Setup

Experiments were run on the Emulab testbed [49], 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. Evaluation and 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.



(a) Copilot. **(b) Multi-Paxos.** **(c) EPaxos.**

Figure 4: 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.

We use 5-replica RSMs since they are a common setup for fault-tolerant services inside a datacenter [8]. 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.

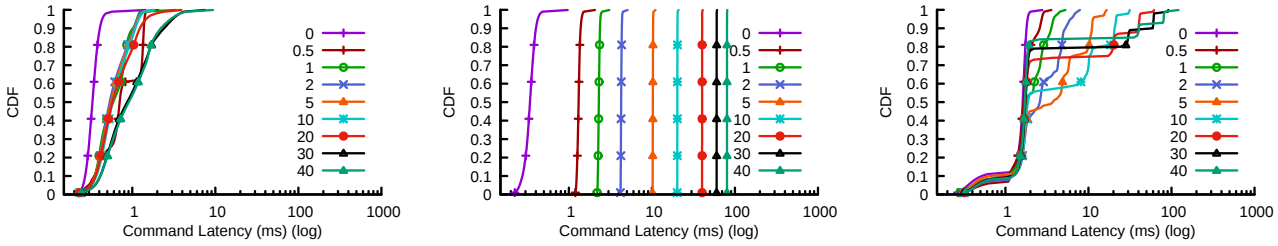
6.3 Transient Slowdowns

Figure 4 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) Copilot. **(b) Multi-Paxos.** **(c) EPaxos-25%.**

Figure 5: 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.

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.

6.4 Slowdowns of Varying Severity

Figure 5 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 5b shows the CDF of latency for Multi-Paxos. The latency of client commands in Multi-Paxos is proportional to 2× 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.

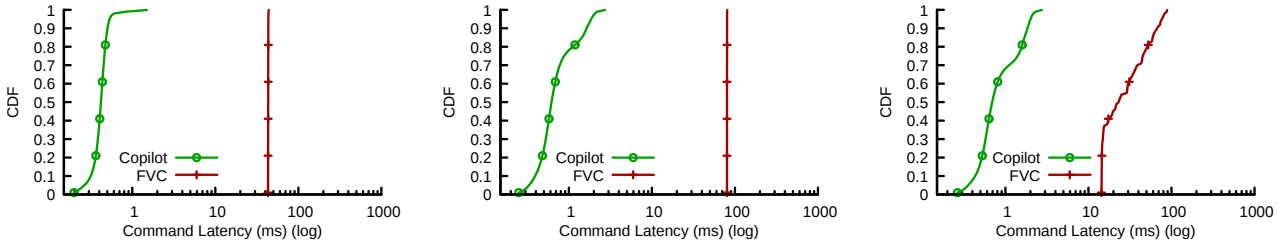
Figure 5c shows the CDF of latency for EPaxos with 25% conflicts. Normal case latency is higher than Multi-Paxos because EPaxos processes batches together, and if one command in a batch acquires a dependency then the entire batch goes to the slow path and does a dependency wait. With 25% conflicts, almost 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, 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 5a 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.

6.5 Slowdowns of Varying Manifestations

Figure 6 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 6a considers a slowdown manifested by a slowed processing path for client commands with a fast processing path for messages from replicas. This experiment uses 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.



(a) Slow for clients. (b) Slow with fast heartbeats. (c) Gradually slow.

Figure 6: 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.

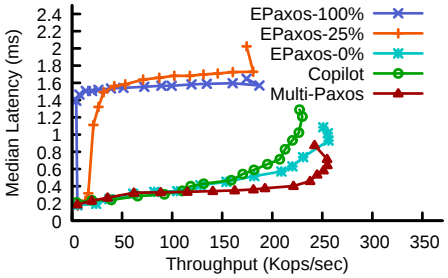


Figure 7: Throughput and latency without the thrifty optimization of the systems when there are no slow replicas.

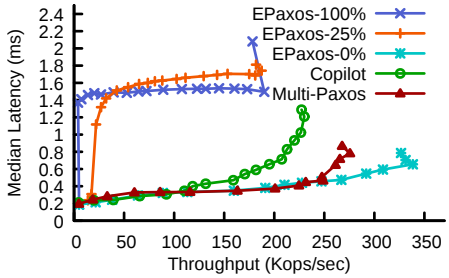


Figure 8: Throughput and latency with the thrifty optimization of the systems when there are no slow replicas.

Figure 6b 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 6c 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.

6.6 Performance Without Slow Replicas

Figure 7 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 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 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 in this case 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.

7 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 [28] and Viewstamped Replication [42]. New consensus protocols improve latency and/or throughput on these baselines [4, 22, 30, 31, 33, 36, 45, 51]. Others are designed to be more understandable [44]. SDPaxos [51] 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 [7]. Its unproxied shared register operations are slowdown tolerant while its consensus operations are not. If the network ordering from NOPaxos [33] 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 [28] 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 [40]. 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 [36]. 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 [3, 10, 12, 19, 27, 47, 50]. 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.

Slowdown cascades. Occult is a scalable, geo-replicated data store that is immune to slowdown cascades [38]. 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.

8 Conclusion

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

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

References

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

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

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

[4] B. Arun, S. Peluso, R. Palmieri, G. Losa, and B. Ravindran. Speeding up consensus by chasing fast decisions. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017.

[5] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly Media, Inc., 2016.

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

[7] M. Burke, A. Cheng, and W. Lloyd. Gryff: Unifying consensus and shared registers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

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

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

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

[11] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live: an engineering perspective. In *ACM Symposium on Principles of Distributed Computing (PODC)*, 2007.

[12] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *ACM Symposium on Operating System Principles (SOSP)*, 2009.

[13] Cockroach DB. `https://www.cockroachlabs.com/product/`, 2020.

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

[15] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[17] etcd docs — Tuning. `https://etcd.io/docs/v3.4.0/tuning/`, 2020.

[18] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[19] R. Guerraoui, N. Knežević, V. Quéma, and M. Vukolić. The next 700 BFT protocols. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2010.

[20] T. Hauer, P. Hoffmann, J. Lunney, D. Ardelean, and A. Diwan. Meaningful availability. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2020.

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

[22] H. Howard, D. Malkhi, and A. Spiegelman. Flexible paxos: Quorum intersection revisited. In *International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

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

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

[25] M. Isard. Autopilot: automatic data center management. *Operating Systems Review*, 41(2):60–67, 2007.

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

[27] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: Speculative byzantine fault tolerance. In *ACM Symposium on Operating System Principles (SOSP)*, Oct. 2007.

[28] L. Lamport. The part-time parliament. *ACM Transac-*

tions on Computer Systems (TOCS), 16(2), 1998.

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

[30] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, March 2005.

[31] L. Lamport. Fast paxos. *Distributed Computing*, 19(2): 79–103, Oct. 2006.

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

[33] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

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

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

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

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

[38] S. A. Mehdi, C. Littley, N. Crooks, L. Alvisi, N. Bronson, and W. 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.

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

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

[41] K. Ngo, S. Sen, and W. Lloyd. Tolerating slowdowns in replicated state machines using copilots. Technical Report TR-004-20, Princeton University, Computer Science Department, 2020.

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

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

[44] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference (ATC)*, 2014.

[45] D. R. Ports, J. Li, V. Liu, N. K. Sharma, and A. Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2015.

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

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

[48] SLA summary for Azure services. `https://azure.microsoft.com/en-gb/support/legal/sla/summary/`, 2020.

[49] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[50] T. Wood, R. Singh, A. Venkataramani, P. Shenoy, and E. Cecchet. Zz and the art of practical BFT execution. In *ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2011.

[51] H. Zhao, Q. Zhang, Z. Yang, M. Wu, and Y. Dai. SDPaxos: Building efficient semi-decentralized geo-replicated state machines. In *ACM Symposium on Cloud Computing (SoCC)*, 2018.