Avicenna: Masking Slowdowns in Replicated State Machines with Counterfactual Evaluation

Christopher Hodsdon [‡*], Zijian Qin[★], Khiem Ngo[◇*]
Siddhartha Sen[°], Ethan Katz-Bassett[†], Wyatt Lloyd[★]
[‡] Databricks [★] Princeton University [◇] Datadog [°] Microsoft Research [†] Columbia University

Abstract

Geo-distributed replicated state machines (RSMs) are at the heart of many production distributed systems, offering linearizability and fault tolerance via consensus protocols. Most existing protocols target crash fault tolerance, however, and are vulnerable to fail-slow faults, where a single slow replica can significantly degrade system latency. Existing protocols that tolerate fail-slow faults do so with much higher normal-case latency in geo-distributed settings.

This paper presents Avicenna, the first consensus protocol for geo-distributed RSMs that maintains low normal-case latency while tolerating a single fail-slow replica. Avicenna uses a single leader to order commands, naturally tolerating a fail-slow follower. To tolerate a fail-slow leader, Avicenna compares the current latency with the counterfactual latency clients would experience if a different replica, the shadow leader, were the leader. When that comparison indicates the current leader might be slow, Avicenna quickly promotes the shadow leader with a fast leader rotation protocol. Our evaluation shows Avicenna has the same normal-case latency as Multi-Paxos while tolerating fail-slow faults.

CCS Concepts: • **Computer systems organization → Dependable and fault-tolerant systems and networks**.

Keywords: distributed system, replicated state machines, fault tolerance

ACM Reference Format:
Christopher Hodsdon, Zijian Qin, Khiem Ngo, Siddhartha Sen, Ethan Katz-Bassett, Wyatt Lloyd. 2026. Avicenna: Masking Slowdowns in Replicated State Machines with Counterfactual Evaluation. In *21st European Conference on Computer Systems (EUROSYS '26), April 27–30, 2026, Edinburgh, Scotland Uk.* ACM, New York, NY, USA, 23 pages. https://doi.org/10.1145/3767295.3803615

[*] Christopher Hodsdon and Khiem Ngo did this work while graduate students at Princeton University.

This work is licensed under a Creative Commons Attribution 4.0 International License.
EUROSYS '26, Edinburgh, Scotland Uk
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2212-7/2026/04
https://doi.org/10.1145/3767295.3803615

1 Introduction

Geo-distributed replicated state machines (RSMs) are an important building block for many production distributed systems, such as coordination services like Chubby [8] and ZooKeeper [22], and distributed databases like Spanner [12] and CockroachDB [49]. An RSM is a set of individual machines that start from the same initial state, use a consensus protocol to agree on an order of client commands, and apply those commands in that order to maintain identical state transitions. As a result, RSMs ensure linearizability and fault tolerance, providing applications with the abstraction of a single machine that never fails [48]. Geo-distributing replicas enables deployments to exploit locality to provide lower client-perceived latency and to increase fault independence across replicas to improve system availability.

Today, most deployed RSMs are designed to tolerate crash failures, where a replica fails by stopping completely. Recent studies show, however, that *fail-slow faults* are prevalent, where hardware or software components degrade in performance without fully crashing [15, 16, 31, 32, 43]. NIC overheating or misconfiguration can delay or drop messages and firmware bugs can cause disk accesses to run slower. A slow replica is correct in the sense that it processes messages and applies state transitions. Unfortunately, most existing consensus protocols do not mask fail-slow faults, and even a single slow replica can slow down the entire RSM [30, 39]. For example, consider leader-based protocols like Multi-Paxos [24, 25, 52] and Raft [42], where the leader coordinates all commands. If the leader becomes slow, the entire system will become as slow as the leader.

The challenge for designing fail-slow fault-tolerant protocols lies in the non-binary and dynamic nature of fail-slow behaviors, which renders ineffective simple solutions that work for crash detection, such as static thresholds [30, 31]. Fail-slow fault tolerance has been shown to be achievable by adding proactive redundancy [39]. However, this redundancy requires additional messages for coordination that degrade normal-case latency, which is an especially undesirable tradeoff in geo-distributed settings. Our evaluation of this approach, as exemplified by Copilot [39], shows its median latency is 1.17–1.58x higher than Multi-Paxos.

In this paper, we show that geo-distributed RSMs can achieve fail-slow fault tolerance without sacrificing normal-case latency. To do so, we provide a new formal definition of a fail-slow fault-tolerant RSM. The key difference with prior

definitions is that our new definition is achievable in a geo-distributed setting. Then, we introduce **Avicenna**, the first consensus protocol for geo-distributed RSMs that tolerates a single fail-slow replica without sacrificing normal-case latency. During normal operation, Avicenna acts like Multi-Paxos, where a single leader determines the execution order of commands, naturally tolerating fail-slow followers. When the leader is slow, Avicenna quickly rotates the leadership to another non-slow replica. The system then resumes normal operation with the new leader coordinating commands.

Though conceptually straightforward, Avicenna's design faces two challenges. First, detecting leader fail-slow faults must be fast, accurate, and adaptive to diverse environments and workloads. As shown in previous studies [30, 31] and our measurements, a static threshold is ineffective at detecting fail-slow faults. A conservative threshold can only detect the most severe faults, allowing mild but persistent slowdowns to continue degrading performance, while an aggressive one triggers unnecessary leader rotations, hurting system availability. This challenge is further aggravated in dynamic environments and workloads, where changes in performance metrics can be natural rather than evidence of fail-slow faults. Furthermore, some fail-slow faults behave as gray failures [21], where external clients observe degraded performance while the system's internal monitoring mechanism observes the system as healthy, making fail-slow fault detection more challenging.

Avicenna solves this challenge by using **counterfactual evaluation**, which continuously compares the client-perceived latency under the current leader and the latency that clients would experience if another replica, the *shadow leader*, were the leader. To determine client latency with the shadow leader, Avicenna uses *shadow processing* where the shadow leader proposes, orders, and commits client commands in parallel to the real leader. Shadow processing is entirely independent of real processing and thus does not require additional messages for coordination between the two. This independence, however, makes shadow execution of commands incomparable to the real execution of those commands. Avicenna thus avoids shadow execution and instead uses the shadow leader's execution of the corresponding commands in the real log for its comparison. Then, based on real and shadow latency feedback from clients, Avicenna evaluates whether the real leader may be fail-slow. This approach enables precise and environment-adaptive detection of fail-slow leaders.

The second major challenge in designing Avicenna is ensuring that leader rotation is safe and fast. In leader-based protocols like Multi-Paxos and Raft, the leader fail-over protocol ensures safety by requiring the new leader to communicate with a majority of replicas. This majority communication involves two network message delays to geographically distant replicas, during which time the system is halted and no new commands are processed. Such a high latency overhead undermines the benefits of leader rotation by adding considerable latency to client commands.

Avicenna solves this challenge with its **fast rotation protocol** that allows the shadow leader to take over upon receiving only a single special message from a nearby replica. Avicenna ensures this design is safe by restricting the replicas that are involved in quorums for the real leader so that the shadow leader can identify all commands committed by the real leader upon receiving that single special message.

We evaluate Avicenna in a geo-distributed setting using AWS. Overall, we observe that Avicenna consistently tolerates fail-slow faults of different kinds and severity, including scenarios when they behave as gray failures. Furthermore, Avicenna delivers normal-case latency that is significantly lower than Copilot and that matches Multi-Paxos. Avicenna's tradeoff is that it has lower throughput than Multi-Paxos due to the overhead of shadow processing.

This paper makes the following contributions:

- A definition of fail-slow fault tolerance for consensus protocols that is applicable to local and geo-replication.
- The first RSM protocol that tolerates one fail-slow replica while maintaining the same normal-case latency as the current state-of-practice in the wide area.
- An evaluation of Avicenna and its open-source codebase at https://github.com/princeton-sns/Avicenna-eurosys2026.

2 Background & Motivation

System model. In this paper, we assume the network is asynchronous where messages can be arbitrarily delayed, reordered, and dropped. Processes can operate at arbitrary speeds, and we do not assume clock synchronization. Replicas run independently and only communicate with each other using network messages. We do not consider Byzantine faults [26]. In order to tolerate f crash failures, at least $2f + 1$ replicas are required.

Linearizability. RSMs ensure linearizability for client commands. Linearizability is a consistency model that guarantees two properties: (1) commands must be executed in some total order, and (2) this order must respect the real-time order, i.e., if command a_m completes before command a_n begins in real-time, then a_m must be ordered before a_n [18].

Fail-slow. The fail-slow (or fail-stutter) fault model captures components that function correctly but with degraded performance [5, 43]. It is between the Byzantine model [26], where failed components can behave arbitrarily, and the crash model [47], where components fail by stopping.

Fail-slow faults are common. Fail-slow faults have long been overlooked, although recent studies show they can be as frequent as crash failures [15, 32, 43]. Fail-slow failures can be caused by internal reasons, such as firmware bugs and NIC driver issues, and external factors, including overheating or

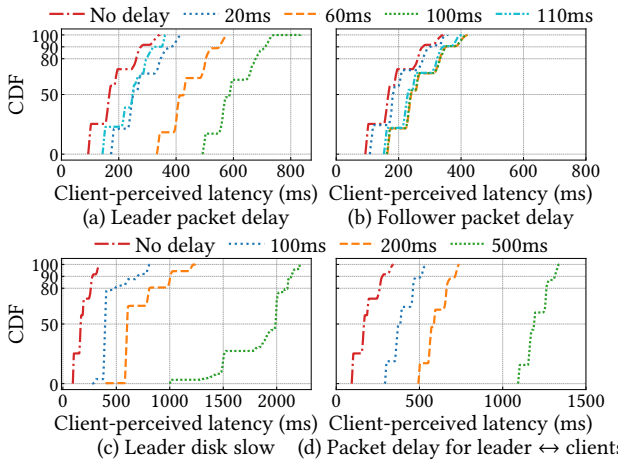

Figure 1. CDF of client-perceived latency of Multi-Paxos with different injected fail-slow faults.

partially failed power supplies [15, 16, 31, 32]. Furthermore, a fail-slow component not only affects its host machine, but can also cause cascading effects across the cluster [15]. As modern computer systems grow in complexity and scale, fail-slow faults are becoming more and more common [5, 29].

Existing RSMs do not tolerate fail-slow faults. Despite their prevalence and impact, existing production RSMs do not handle fail-slow faults. Multi-Paxos [24, 25, 52] and Raft [42] are the state-of-practice for distributed systems [1–3, 7, 12, 22, 49], where a single leader orders commands, broadcasts the order to replicas, and commits commands when receiving replies from a majority of replicas (including itself).

However, because the leader is in charge of all commands, a fail-slow leader can degrade the latency of the entire system. In Fig. 1 (a), Multi-Paxos suffers significant latency degradation under packet delays injected to the leader. For instance, the 90th percentile latency degradation is 2.8× under a 100 ms packet delay. This experiment was done with 7 geo-distributed replicas, each co-located with 20 clients. More details about experiment setup can be found in Section 5.1. As shown by recent studies, packet delay can vary from microseconds to seconds [16]. Thus, this experiment reveals the vulnerability to fail-slow faults of production systems that use Multi-Paxos as their building block.

Our other measurement results suggest, however, that tolerating such faults purely by protocol design might be possible. When the packet delay is injected to a follower, as shown in Fig. 1 (b), latency degradation remains small and does not scale with the magnitude of the delay. This is expected since the leader can commit a command as long as it receives replies from a majority of replicas and thus can safely exclude the fail-slow follower. Furthermore, in Fig. 1 (a), when packet delay at the leader increases to 110 ms, latency degradation becomes less severe than the case with a 20 ms delay. In this case, the packet delay is severe enough that the round-trip time for a heartbeat message exceeds the leader change timeout, allowing a non-slow replica to take over as leader.

This observation motivates us to design a fail-slow fault-tolerant RSM that rotates the leadership to another replica when the current leader is slow. However, a static threshold, like the one used here, is unreliable for detecting fail-slow leaders [30, 31]. A conservative threshold can only detect severe enough faults and leaves mild faults continuously degrading latency; an aggressive threshold triggers unnecessary leader rotations, sacrificing system availability. Manually fine-tuning thresholds is not only cumbersome but also unreliable, as any chosen value is tailored to a specific environment or workload and may not generalize well to others. In addition, with dynamic environments and workloads, observed variations in performance metrics are to be expected and should not be mistaken as evidence of fail-slow faults.

To make this problem worse, existing systems usually utilize internal monitoring mechanisms, such as heartbeat messages, to detect failed components, which may not faithfully reflect *client-perceived* system performance. This can leave the fail-slow faults undetected when they behave as *gray failures* [21],[1] where external clients observe the system as unhealthy but internal monitoring mechanisms observe the system as healthy [20, 21, 30]. This difference in observability between impacted clients and monitoring components poses additional challenges in handling fail-slow faults. As shown in Fig. 1 (c), when disk writes are slow at the leader of Multi-Paxos, the system latency severely degrades. Even when the slowdown reaches 500 ms, which is significantly higher than the re-election interval, the leader re-election is still not triggered. This is because the leader still responds to heartbeat messages normally although its disk writes are severely slow. The same problem is also manifested when the packets between the leader and clients are delayed, shown in Fig. 1 (d). This observation motivates us to bridge the gap of observability between external clients and internal monitoring components, triggering leader rotation when the leader's fail-slow faults impact client-observed latency.

Tolerating fail-slow faults in geo-distributed settings is challenging. Geo-distributed RSM deployments, where replicas are spread across multiple data centers, are common in production and have gained increasing academic attention [4, 10, 11, 14, 37, 46, 49, 53]. For example, CockroachDB [49] uses Raft [42] for consistent data replication, with replicas distributed across geographic regions by default. Geo-distribution can reduce client-perceived latency by exploiting locality and it improves system availability by leveraging failure independence across geographic zones.

Designing fail-slow fault-tolerant RSM protocols in this common setting is challenging. Particularly, non-negligible,

[1]Gray and fail-slow failures are distinct but overlapping fault models. Some faults are only fail-slow, some are only gray, and some are both.

dynamic, and heterogeneous network latencies in wide-area networks (WANs) demand caution from system designers. While fail-slow fault tolerance can be achieved by adding proactive redundancy, the associated complexity is amplified in geo-distributed settings and can degrade normal-case latency, ultimately undermining the intended benefits. For example, Copilot [39] is the state-of-the-art RSM protocol that masks a single fail-slow replica. It achieves this by proactively adding redundancy, where two leaders coordinate commands concurrently during normal operation. In this way, each command can traverse two disjoint paths, preventing any fail-slow leader from degrading system latency.

However, this robustness comes at a performance cost. If the two leaders do not issue concurrent proposals, commands are committed on the fast path with two network message delays, the same as Multi-Paxos. However, if proposals are concurrent, they must be properly resolved between the two leaders to maintain linearizability, resulting in a slow path with four network message delays. To avoid the slow path, Copilot uses a ping-pong batching optimization that has leaders alternate proposals by waiting on each other. In geo-distributed settings this coordination via waiting for messages significantly increases latency. Our evaluation in Section 5 shows that Copilot with ping-pong batching has 1.17–1.58x higher median latency than Multi-Paxos.

3 Defining Fail-slow Fault Tolerance

This section formally defines a fail-slow fault-tolerant RSM. Based on our definition, we show why Multi-Paxos does not achieve it. We then contrast our definition with Copilot's, which is unachievable in geo-distributed settings.

Definition of fail-slow fault tolerance. A replica is fail-slow if it functions correctly but with degraded latency. An RSM tolerates s fail-slow replicas if clients perceive normal latency despite any s replicas being fail-slow. To formalize this, we consider a hypothetical system where those s fail-slow replicas are removed. This represents an ideal scenario where fail-slow replicas can be detected and isolated *accurately* and *instantly*. Denote the latency gap between the hypothetical system and the real system as ϵ. Then, we have the following definition of fail-slow fault tolerance:

Definition 3.1 (ϵ-s-Fail-Slow Fault Tolerance)**.** *An RSM is ϵ-s-fail-slow fault-tolerant if its latency in the presence of s fail-slow replicas increases over that of the corresponding hypothetical system by at most ϵ.*

Note that $s \leq f$, where f is the maximum number of replicas the system can tolerate under the crash model.

Analyzing Multi-Paxos and Avicenna. According to this definition, protocols like Multi-Paxos are not fail-slow fault tolerant because they have an ϵ of ∞. To see this, consider that in the hypothetical system, a fail-slow leader is accurately and instantly identified and isolated with another replica taking over leadership without delay. In contrast, in the real system, when there are gray fail-slow faults of the leader, such as the slow disk writes as shown in Fig. 1 (c), the system's internal monitoring mechanism never triggers leader election and the fail-slow leader remains in place indefinitely. Because these gray failures never trigger leader election, the latency gap for these protocols is unbounded.

In this paper, we take a first step toward fail-slow fault-tolerant RSMs by presenting Avicenna, an ϵ-1-fail-slow fault-tolerant RSM. Avicenna is designed to tolerate a single fail-slow replica. As demonstrated by evaluation results in Section 5, the ϵ of Avicenna is as small as that of Copilot while maintaining the same normal-case latency as Multi-Paxos. We leave tolerating multiple fail-slow replicas without sacrificing normal-case latency to future work.

Critique of Copilot's definition. Copilot [39] defines a system as *s-slowdown-tolerant* if when s replicas are slow, it matches the latency of a hypothetical system where the s slowest replicas are replaced by clones of the slowest remaining (non-slow) replica.

While intuitive, Copilot's definition is unachievable in geo-distributed settings due to unavoidable heterogeneity in network latency between replicas. For example, consider an RSM with 2 replicas in London and 1 replica in Oregon. When 1 replica in London is fail-slow, a quorum of 2 replicas in London can no longer be formed. Copilot's definition, however, requires a slowdown-tolerant RSM to match the latency of that unachievable configuration. Avicenna avoids such a replacement model. Instead, it considers a hypothetical system where the s slow replicas are simply removed. This reflects an ideal but still physically realizable baseline.

4 Design

This section details Avicenna's design. Section 4.1 covers normal operation. Sections 4.2 and 4.3 describe the leader fail-slow detection and reaction protocols, respectively. Section 4.4 completes the design and adds optimizations. Pseudo code for Avicenna and formal proofs of its safety and liveness are provided in Appendix A.

4.1 Normal Operation

This subsection describes Avicenna's normal-case operation protocol, which is shown in Fig. 2. Avicenna uses a single *real leader* to order commands, which achieves the same normal-case latency as Multi-Paxos and naturally tolerates any single fail-slow follower. In addition, Avicenna has a *shadow leader* run shadow processing for commands, which lays the foundation to tolerate fail-slow faults at the real leader.

Replica roles and configurations. Each replica maintains a monotonically increasing phase number starting from 0. The role of each replica remains unchanged during a phase.

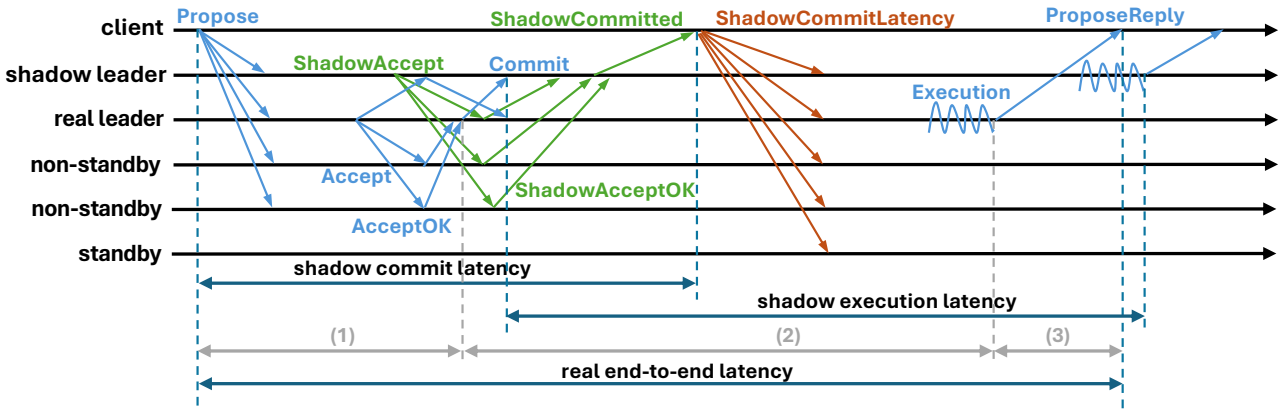

Figure 2. Normal-case operation of Avicenna. Messages are spaced out to improve readability; the latency for clients between sending a request and receiving a response matches that of Multi-Paxos. Real processing, coordinated by the real leader, determines the execution order of commands. Shadow processing, coordinated by the shadow leader, happens in parallel to the real processing to determine system behavior when the shadow leader is in control. The client directly measures the real end-to-end latency, which is decomposed into three components. To construct the counterfactual shadow end-to-end latency, the client measures and reports the shadow commit latency, and the shadow leader measures the shadow execution latency.

In each phase, one replica acts as the real leader and coordinates real processing. The real leader for the next phase acts as the shadow leader in the current phase, coordinating shadow processing. The $f-1$ replicas that are farthest from the real leader act as *standbys*, who do not participate in commit quorums. The other $f+2$ replicas, including the real and shadow leaders, are *non-standby replicas* that participate in commit quorums. Restricting participation in commit quorums enables Avicenna's fast leader rotation protocol.

Real processing. Clients send each command to all non-standbys. Each command is tagged with a `ClientID`, a client-chosen `CommandID`, and a flag `IsShadow`. `IsShadow` is used to determine if a command should be shadow processed. Upon receiving the command, the real leader puts the command in the next available entry of its *real log* and sets the status to `Proposed`. Entries in the real log are uniquely identified by monotonically increasing `Inst` numbers starting from 0.

The real leader then broadcasts an Accept message to *all non-standby* replicas, including `Phase`, `Inst`, `ClientID`, `CommandID`, and `IsShadow`. When receiving an Accept, each non-standby replica puts the command in the corresponding real-log entry, marks it `Accepted`, and replies with an AcceptOK message. Upon receiving $f+1$ AcceptOKs (including its own), the real leader marks the entry `Committed` and broadcasts a Commit message to *all* replicas. Replicas then set the status to `Committed`. Once the command is executed, each replica replies to the client with a ProposeReply message. A command can be executed only after it is committed and all commands in preceding log entries have been executed.

Standbys do not participate in commit quorums, but do learn Commits and execute commands.

Shadow processing. To enable fail-slow leader detection, Avicenna introduces shadow processing. The real leader of the next phase is the shadow leader in the current phase and coordinates *shadow processing* to counter-factually evaluate the system's behavior with the shadow leader in charge.

Separate from the real log, each replica maintains a *shadow log*, whose entries are uniquely identified by monotonically increasing `ShadowInst` numbers starting from 0. If `IsShadow` is set to `True` by the client, the shadow leader puts the command in the next available entry of its shadow log and broadcasts a ShadowAccept message to all replicas, including `Phase`, `ShadowInst`, `ClientID`, `CommandID`, and `IsShadow`. When receiving a ShadowAccept, *all non-standby* replicas put the command in the corresponding shadow log entry, mark it `ShadowAccepted`, and reply with a ShadowAcceptOK message. Upon receiving $f+1$ ShadowAcceptOKs (including its own), the shadow leader marks the entry `ShadowCommitted`, sends a ShadowCommitted message to the client to enable reconstruction of shadow end-to-end latency (Section 4.2), and broadcasts a ShadowCommit message to all replicas. Replicas then set the status to `ShadowCommitted`.

Due to the independence of shadow processing, commands in the shadow log may appear in a different order from the real log, making shadow execution of commands incomparable to real execution of those commands. Thus, commands in the shadow log are *never* executed. Instead, the shadow leader executes the same commands in its real log to reconstruct an end-to-end latency for comparison (Section 4.2).

It is worth noting that the shadow leader responds to Accept messages from the real leader with AcceptOK, and the real leader similarly responds to ShadowAccept messages from the shadow leader with ShadowAcceptOK.

Finally, if a client times out waiting for a command with `IsShadow=False`, it retransmits the command with

`IsShadow=True` to ensure bounded latency.

Avicenna achieves the same normal-case latency as Multi-Paxos. First, a single *real leader* orders and commits commands in two network message delays, identical to Multi-Paxos. Second, although standbys in Avicenna do not participate in commit quorums, this design does not increase quorum latency. Standbys are intentionally chosen to be the replicas farthest from the real leader, so the $f + 1$ replicas that form the quorum are the closest ones. As a result, the quorum latency is the same as that of Multi-Paxos, which does not designate standbys but often experiences similar effects due to geographical proximity influencing quorum composition. Third, shadow processing is independent from real processing and so requires no additional messages to coordinate proposals between real and shadow leaders.

Safety. In the normal operation, a single real leader determines the execution order of commands, i.e., the order in the real log. Commands in the shadow log are never executed. Thus, linearizability is guaranteed during normal operation.

Liveness. During normal operation, $f - 1$ replicas are standbys that do not participate in commit quorums. Hence there are $f + 2$ non-standby replicas, and if two or more *non-standby* replicas crash, a quorum of $f + 1$ can no longer be formed, making the system stall. Simply rotating to the next phase may not resolve this situation if the configuration still includes insufficient live non-standby replicas. To handle this problem, Avicenna includes special phases with *Armageddon* configurations where all $2f + 1$ replicas are non-standbys.

Note that safety is never violated regardless of whether a phase with *Armageddon* configuration exists. To ensure liveness, a phase with an *Armageddon* configuration must be rotated into eventually. More details about Armageddon configurations can be found in Section 4.4. However, we argue that it is unusual for a second crash to occur before the first is repaired, especially in geo-distributed settings where replicas have high failure independence. Thus, we include Armageddon configurations infrequently. These cases do not affect Avicenna's 1-fail-slow tolerance because they only occur when there are two or more failures.

Tolerating fail-slow followers. In normal-case operation, the real leader commits a command upon receiving $f + 1$ AcceptOKs from $f + 2$ non-standby replicas (including itself). All replicas execute commands and reply to clients. As a result, Avicenna naturally tolerates any one fail-slow follower: the real leader will simply use the replies from the other replicas, which is identical to what would happen if that fail-slow follower were removed from the system.

4.2 Fail-slow Detection

In this subsection, we describe Avicenna's fail-slow leader detection protocol. The goal is to detect fail-slow faults of the real leader *quickly*, *accurately*, and in a manner that is *adaptive* to environments and workloads.

Counterfactual evaluation. Avicenna relies on *counterfactual evaluation*, which continuously compares the current latency and the latency that clients would experience had the shadow leader been in charge, to detect a fail-slow leader. Comparing to an alternative allows Avicenna to avoid relying on an ineffective static threshold. Using latency feedback from clients allows Avicenna to detect all types of fail-slow faults, including gray failures.

For each command c the client measures the *real end-to-end (e2e) latency*, $\ell^{\text{real}}_{\text{e2e}}(c)$, as the time from sending c to receiving the first ProposeReply for c, and piggybacks this latency value on its next command proposal. We measure e2e latency because it directly captures application-perceived responsiveness.

Estimating the *shadow e2e latency* is less straightforward. Although shadow processing constantly simulates the system behavior if the shadow leader were actually in charge, it is entirely independent from real processing. Due to this independence, commands in the shadow log may appear in a different order from the real log, which makes shadow execution of a command incomparable to its real execution. For instance, consider a scenario with no fail-slow faults where command c_1 and c_{100} takes 1 ms and 100 ms to execute, respectively. Let both commands commit at the same instant in the real and shadow logs. If the real-log order is c_{100}, c_1 and the shadow-log order is c_1, c_{100}, then the time between when c_1 commits and finishes executing will be much larger with the real leader, despite there being no fail-slow fault.

As a result, Avicenna avoids executing commands in the shadow log, and the *shadow e2e latency* is not captured directly. To fill this gap and enable a fair comparison between the real and shadow leaders, we first decompose the *real e2e latency* into three components (shown in Fig. 2): (1) client sends `Propose` → real leader commits; (2) real leader commits → real leader finishes execution; (3) real leader finishes execution → client receives `ProposeReply`. To simulate the scenario where the shadow leader is in charge, the counterfactual *shadow e2e latency* for command c, denoted as $\hat{\ell}^{\text{shadow}}_{\text{e2e}}(c)$, must account for analogous components, substituting the real leader with the shadow leader. To this end, we propose a *reconstruction* mechanism:

$$\hat{\ell}^{\text{shadow}}_{\text{e2e}}(c) = \ell^{\text{shadow}}_{\text{commit}}(c) + \ell^{\text{shadow}}_{\text{exec}}(c),$$

where $\ell^{\text{shadow}}_{\text{commit}}$, the *shadow commit latency*, corresponds to components 1 and 3, and $\ell^{\text{shadow}}_{\text{exec}}$, the *shadow execution latency*, captures component 2. In the following, we describe how each term is measured and why this reconstruction is faithful.

The shadow commit latency. Component (1) in real processing maps directly to shadow processing: from the client's send of c (with `IsShadow=True`) to the shadow leader's shadow-commit of c. Component (3) in real processing is a one-way message delay of the ProposeReply sent from the

real leader to the client. To mirror both of these in shadow processing, the shadow leader sends a ShadowCommitted message upon shadow committing the command, notifying the client that this command has been shadow committed. The client measures

$$\ell^{\text{shadow}}_{\text{commit}}(c) = \text{send}(c) \rightarrow \text{receive ShadowCommitted}(c)$$

and reports this to replicas.

The shadow execution latency. Component (2) in real processing includes both the *queuing delay* from commit to execution start and the execution time itself at the real leader. At first glance, it seems infeasible to measure the corresponding latency in shadow processing, since queuing delay can vary due to the independence of the real and shadow logs. However, every command in the shadow log also appears in the real log, and the shadow leader executes commands in its real log, just like any other replica. Thus, the shadow leader measures

$$\ell^{\text{shadow}}_{\text{exec}}(c) = \text{receive Committed}(c) \rightarrow \text{finish executing } c$$

and piggybacks this duration on its next ShadowCommit. This duration captures the queuing delay and execution time at the shadow leader, corresponding directly to component (2) in real processing. Both terms are local *durations*, not absolute timestamps, so no clock synchronization is required. Each replica reconstructs $\hat{\ell}^{\text{shadow}}_{\text{e2e}}(c)$ by adding $\ell^{\text{shadow}}_{\text{commit}}(c)$ received from the client to $\ell^{\text{shadow}}_{\text{exec}}(c)$ received from the shadow leader.

Each replica maintains a *latency store* to track paired real and shadow e2e latencies. To enable a fair comparison between real and shadow leaders, pairs are inserted only when both latency values for the same command are available; similarly, both are removed together. To enforce this, replicas keep a buffer for unpaired latencies. When the counterpart arrives, the pair is flushed into the latency store.

Objective functions. Based on latencies in the store, a naïve approach to trigger leader rotation would be to handcraft fixed rules. However, such fixed policies are inflexible and poorly suited to dynamic environments. To address this, Avicenna allows applications to define custom *objective functions* that determine when to trigger leader rotation based on criteria tailored to specific performance goals. Here, we formalize the interface and give some examples.

Denote $\mathcal{R}_T$ as the set containing *real* latencies whose insertion timestamps fall within the last T seconds, and $\mathcal{G}_T$ as the set containing *shadow* latencies inserted within the last T seconds. Given any scalar aggregator $A(\cdot)$ (e.g., max, a percentile, a trimmed mean), Avicenna calculates

$$F_T = A(\mathcal{R}_T) - (1+\tau)A(\mathcal{G}_T) - \beta$$

and triggers rotation when $F_T > 0$, where $\tau \in (0, 1)$ sets the *multiplicative* margin and $\beta \geq 0$ sets the *additive* margin. These margins allow applications to make a tradeoff between higher reactivity and having a more consistent leader. Avicenna exposes $A(\cdot)$ as an interface to applications for defining custom objective functions.

Example: Max objective. $A = \max$, the maximum of the set. This minimizes worst-case latency and reacts quickly, but can be jitter-sensitive.

Example: Percentile objective. $A = \text{Perc}_p$, the p-th percentile of the set. This smooths transient spikes by focusing on the p-th percentile rather than the maximum, while guaranteeing reaction within T seconds under a sustained slowdown.

Example: Tail-mean objective. $A = \text{Tail}_q$, the mean over the top q% tail of the set. This focuses on tail latency while being less brittle than the pure max.

Adaptive to environments and workloads. Detection is *relative*: both $\mathcal{R}_T$ and $\mathcal{G}_T$ co-move when environments or workloads change, so no static threshold is required. In addition, using a common time window T for both sets ensures fairness under nonstationary workloads.

4.3 Fail-slow Reaction

This subsection describes Avicenna's leader fail-slow reaction protocol, which is shown in Fig. 3. Avicenna mitigates fail-slow faults at the real leader by rotating leadership to the shadow leader. This leadership rotation must happen quickly to minimize the impact of fail-slow faults.

Rotation. Any replica that suspects the real leader of being slow using the objective function on latencies in its latency store broadcasts a Rotate message to all replicas. This message contains the replica's current `Phase` number and its real-log entries along with their statuses. Upon receiving a Rotate message with a matching `Phase` number, the recipient replica constructs its own Rotate message and broadcasts it to all replicas, if it has not done so. After broadcasting a Rotate message, the replica ignores any subsequent Accept or ShadowAccept messages in that phase. If the replica is a real or shadow leader, it also stops putting new commands received from clients into real or shadow log entries. Replicas still commit and execute commands for that phase.

Log merging. Before the shadow leader takes over leadership, to guarantee safety, we must ensure all committed commands are preserved across leadership.

Once a non-standby replica has received at least one Rotate message from a non-standby replica, it begins log merging. Standbys wait until they have received two Rotate messages from non-standby replicas. *Log merging* reconciles received log entries with a replica's own local log to ensure consistency before moving to the next phase. Merging is performed entry by entry: for each `Inst`, the replica compares the command and status across logs. After completing one `Inst`, the replica advances to the next `Inst`. The comparison and merging logic for each `Inst` follows these rules:

(1) If a log entry has been committed locally, skip;

(2) If a log entry has status `Committed` in *any* received log and is not yet committed locally, immediately learn it as

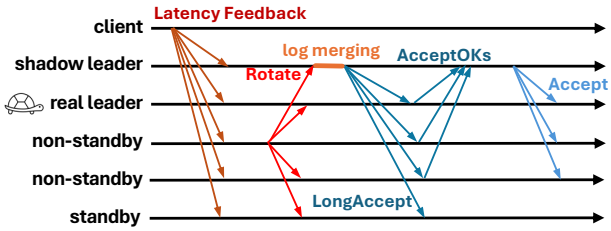

Figure 3. The fail-slow reaction protocol. Some messages are omitted for readability. When a replica suspects the real leader is slow, it broadcasts a ROTATE message. The shadow leader safely takes over leadership after merging its log with a non-standby's and broadcasts a LONGACCEPT message, after which it assumes normal operation as a real leader.

`Committed` and broadcast a COMMIT message;

(3) If a log entry has status `Accepted` in only one log, copy that command to the local log entry as `Accepted`.

(4) If a log entry has status `Accepted` in at least two logs (including its own) and their commands are different, copy the one with the highest phase to the local log entry as `Accepted`.

The shadow log is merged in the same way as the real log. Here we assume replicas keep their entire logs indefinitely for exposition, and we present an optimization to reduce log exchanging and merging overheads in Section 4.4.

Leadership takeover. After log merging, replicas increment `Phase` by one and start performing roles in the new phase. By design, the shadow leader of the previous phase now becomes the new real leader. Before proposing new commands received from clients, the new real leader must first propose all uncommitted commands in its real log. To achieve this, the new real leader broadcasts a LONGACCEPT message, containing all real-log entries from the smallest uncommitted `Inst` to the highest `Inst`, along with their statuses. To reduce latency spikes during rotation, the new leader also batches any received yet uncommitted commands into a new entry and includes it in the LONGACCEPT. Each non-standby replica in the new phase responds with ACCEPTOKs for entries marked `Accepted` and immediately commits an entry with status `Committed` if it has not done so. The new real leader then begins proposing new commands as normal, and other replicas resume normal operation.

Similarly, if a replica becomes the shadow leader in the new phase, it broadcasts a LONGSHADOWACCEPT message and begins proposing new commands.

Low latency overhead. Avicenna is designed to enable fast leader takeover. Excluding standbys from commit quorums allows the shadow leader to learn all committed commands after receiving just *one* ROTATE from a non-standby replica. Moreover, that one ROTATE often comes from a nearby replica; because standbys are chosen to be those geographically farthest from the real leader and the shadow leader is typically near the real leader, the non-standby tend to also be close to the shadow leader. Thus, the use of standbys always reduces the quantity and often reduces the magnitude of message delays during leader takeover.

Safety. The core of ensuring safety is to preserve all committed log entries during leader rotation. During normal operation, $f-1$ standbys do not participate in commit quorums, so any committed entry must have been accepted by at least $f+1$ of the $f+2$ non-standby replicas. Since the real-log order is determined by a single real leader at a time, there can only be at most one accepted command for each log entry. Thus, merging any two real logs from non-standby replicas is sufficient to preserve all committed entries. This ensures correctness during leader transitions.

If the leader rotation is triggered in a phase with *Armageddon* configurations, when any replica can accept log entries, log merging is similar to Multi-Paxos: each replica must collect real logs from at least $f+1$ replicas (including itself) before beginning the log merging process. The rules for log merging are the same as those described previously.

4.4 Design Extensions and Optimizations

This subsection completes the design of Avicenna with mechanisms that reduce detection latency, log merging overhead, and throughput overhead.

Armageddon configurations. To guarantee liveness, Avicenna includes Armageddon configurations where all $2f+1$ replicas are non-standbys. Specifically, phase p is an Armageddon phase if $(p+1) \mod (N+1) = 0$ (N is the number of replicas); other phases are normal phases with $f-1$ standbys. Armageddon phases are entered by rotations.

To ensure rotation can be successfully triggered when multiple non-standbys fail in a normal phase, Avicenna has progress-triggered rotations in addition to latency-triggered rotations. Each replica keeps a progress timer and resets it when receiving a COMMIT message from the real leader. When this timer expires, the replica broadcasts a ROTATE message and starts the rotation protocol described in Section 4.3. This mechanism ensures that Avicenna can eventually enter an Armageddon phase and guarantees liveness with up to f crashes.

Short path for latency feedback. The default counterfactual evaluation mechanism compares real and shadow e2e latencies, which requires a command to *finish execution* before the client can report latency. However, execution may be slow for some commands (e.g., scanning the entire database), delaying the fail-slow detection. To accelerate detection, we introduce a *short path* that compares *commit* latencies, as shown in Fig. 4.

Recall that the shadow leader sends SHADOWCOMMITTED to the client once it gathers $f+1$ SHADOWACCEPTOKs for a command, which allows the client to measure the *shadow commit latency*. Similarly, in the short path, for commands

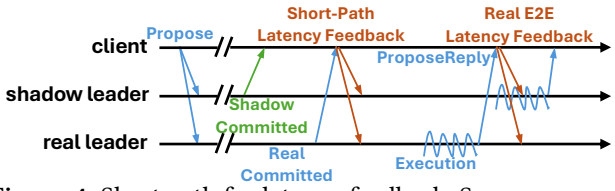

Figure 4. Short path for latency feedback. Some messages are omitted for readability.

with `IsShadow=True`, the real leader also sends RealCommitted to the client upon receiving $f+1$ AcceptOKs, notifying the client that the command has been committed and will be executed eventually. The client measures the *real commit latency*, denoted as $\ell^{\text{real}}_{\text{commit}}$, as the time from sending the command to receiving RealCommitted. Once the client receives both RealCommitted and ShadowCommitted for a command, it reports the pair $\ell^{\text{real}}_{\text{commit}}$ and $\ell^{\text{shadow}}_{\text{commit}}$ to all replicas.

Each replica maintains a separate latency store for real and shadow commit latency pairs received from clients. Let $\mathcal{R}_T^{\text{commit}}$ and $\mathcal{G}_T^{\text{commit}}$ denote, respectively, the real and shadow commit latency samples whose insertion timestamps fall within the last T seconds. The same objective-function interface applies: for any scalar aggregator $A(\cdot)$, we calculate

$$F_T^{\text{commit}} = A(\mathcal{R}_T^{\text{commit}}) - (1+\tau)A(\mathcal{G}_T^{\text{commit}}) - \beta$$

and rotate when $\max(F_T, F_T^{\text{commit}}) > 0$.

Note that the short path *does not* observe execution slowdowns after commit; therefore, it complements rather than replaces e2e-based detection. Avicenna uses both, triggering rotation when either signal exceeds the threshold.

Client early report. Based on the short path for latency feedback, we further accelerate the fail-slow detection via an *early report* mechanism. By default, a client reports real and shadow commit latency after receiving both. However, under severe leader slowdowns or crashes, RealCommitted may be delayed, degrading the timeliness of detection.

To enable timely latency reporting, if ShadowCommitted of a command arrives first, the client starts a timer. If RealCommitted has not arrived before a timeout, the client measures a *real commit at-least latency*, defined as the time elapsed from sending the command until the timer expires, and pairs it with the shadow commit latency in its short-path feedback. The timeout interval is configurable and should be high enough to avoid frequent timeouts during normal conditions yet low enough to enable timely feedback. Replicas treat early reports identically to regular commit latency reports to compute F_T^{commit}. If RealCommitted later arrives, the client re-sends the pair with both concrete latencies, superseding the prior *at-least* value.

For symmetry and fairness of comparison between real and shadow latencies, if RealCommitted arrives first, the client likewise starts the timer and, upon timeout, reports *shadow commit at-least latency* paired with real commit latency.

Reducing log exchanging and merging overhead. To keep log exchanging and merging overhead low, Avicenna only includes recent entries that are not guaranteed to be in all non-standby logs. To do so, the real leader tracks a prefix of real-log entries that can be safely omitted when sending Rotate. The real leader marks an entry as `OverCommitted` after receiving $f+2$ AcceptOKs (including itself), meaning all non-standbys have accepted it. The real leader keeps a progressive `mininst` such that all entries prior to `mininst` are `OverCommitted`, and piggybacks `mininst` in Commit messages. When constructing a Rotate message, each replica includes only its real-log entries with `inst` > `mininst`. A similar optimization applies to the shadow log.

This truncation reduces both transfer and merge latency without hurting safety, since real-log entries prior to `mininst` are already accepted by all non-standby replicas and thus known to the shadow leader.

Sampled shadow processing. Avicenna's primary tradeoff is that the additional work it does to shadow process commands decreases throughput relative to Multi-Paxos. To partially mitigate this tradeoff, Avicenna can shadow process only some commands, e.g., 5% of commands. Such sampling requires no changes to the design because the shadow log is not executed. It only requires the flag `IsShadow`.

5 Evaluation

We evaluate Avicenna to answer the following questions:

- What is Avicenna's normal-case latency in a geo-distributed setting?
- How well does Avicenna tolerate fail-slow faults of various types and severities?
- What are the benefits brought by the optimizations?
- What is the effect of different objective functions?
- What is Avicenna's throughput-latency trade-off?

5.1 Methodology

Implementation. Avicenna is implemented in Go in the same codebase as Copilot [39] and other baselines to enable apples-to-apples comparisons. Avicenna's implementation is open source and available at https://github.com/princeton-sns/Avicenna-eurosys2026.

Testbed and setup. We answer the first four questions in a geo-distributed setting using AWS EC2 c5a.8xlarge virtual machines (32 vCPUs with 64 GiB memory). We run 7 replicas ($f = 3$) in the following locations (Setting 1): Virginia, Oregon, Mumbai, Tokyo, Singapore, Sydney, and London. Each location has 1 replica and 20 clients. Moreover, to make the answer for the first question more general, we run experiments in an additional setting (Setting 2), where 7 replicas run in Virginia, Oregon, California, Mumbai, Frankfurt, Tokyo, and Seoul.

We answer the throughput-latency question in a single

datacenter setting using Emulab d430 machines (2 Intel E5-2630v3 8-Core CPUs with 64 GiB memory). We run 7 replicas ($f = 3$) and a variable number of clients.

Baselines. We evaluate the following baseline protocols:

Multi-Paxos-FVC. In line with Copilot [39], we use a *fast view change* version of Multi-Paxos (Multi-Paxos-FVC) to represent the state-of-practice leader-based protocols. We adopt the exact implementation in the Copilot code base. Multi-Paxos-FVC deviates from standard Multi-Paxos [24, 25, 52] in two ways: (1) View changes are triggered by a master process that never fails or becomes slow. The master process pings the leader every 70 ms (the RTT between master and the leader) and triggers a view-change as soon as it does not receive a reply from the leader after a view-change interval. The view-change interval is configured to be 280 ms, 4 times of the RTT between the master and the leader. (2) Instead of running a leader election, a view-change immediately designates a replica as the next leader. These two modifications make Multi-Paxos-FVC the performance upper bound of Multi-Paxos during fail-slow faults.

EPaxos [34]. We use EPaxos as a representative leaderless protocol. We fixed an implementation bug in the original code base that prevented any fast-path commits when running with 7 replicas. In line with EPaxos Revisited [51], instead of adjusting the fraction of accessing a single hot key to control conflict, we use a Zipfian distribution to approximate real-world access patterns where conflicts are controlled by a skewness factor. With batching, two batches conflict if any operations in the two batches conflict. Each client sends its commands to its co-located replica.

Copilot [39]. We use Copilot as a representative state-of-the-art slowdown-tolerant consensus protocol. We adopt the exact implementation in the Copilot code base. Unless otherwise specified, the ping-pong batching optimization is enabled. To ensure a fair comparison with Multi-Paxos-FVC, the fast take-over interval is also configured to be 280 ms. The ping-pong timeout interval is configured to be 140 ms, 2 RTTs between the two leaders.

Configurations. Here, we specify other configuration parameters. We use a key-value store built upon consensus protocols with 100,000 keys in total. Unless otherwise specified, all commands are 2 B writes. All clients are closed-loop. The batching interval is 10 ms in the geo-replicated setting and 250 μs in the single datacenter setting. At the start of each experiment, replicas ping each other to construct a point-to-point RTT map. For each replica, we estimate the maximum client end-to-end latency under hypothetical leadership, then sort replicas in ascending order of this estimate. Ranks are assigned as $r \in \{0, \ldots, N-1\}$, where N is the number of replicas. In Copilot, replicas with $r = 0$ and 1 are the two leaders. In phase p of Multi-Paxos-FVC, the replica with rank $r \equiv p \mod N$ is the leader. In Avicenna, any phase p with $(p+1) \mod (N+1) = 0$ is an Armageddon phase with replica $r \equiv \lfloor p/(N+1) \rfloor \mod N$ as the real leader; other phases are normal phases with replica $r \equiv p \mod (N+1)$ as the leader. In Avicenna, the real leader of the next phase acts as the shadow leader in the current phase. Standbys in each phase are chosen as the $f-1$ (2 in this case) replicas with the largest RTTs to the real leader.

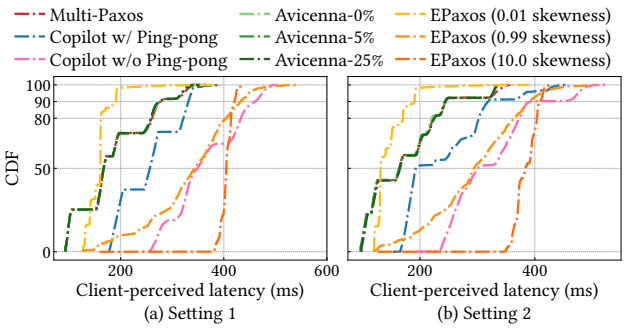

Figure 5. CDF of normal-case latency to show Avicenna provides low latency. Two different geo-distributed settings are presented to show the result is general.

Avicenna-x% indicates each client randomly chooses x% of its commands to be shadow processed. For the objective function we use $A = \text{Tail}_{95}$, $T = 5$ s, $\tau = 0.2$, and $\beta = 10$ ms (the batching interval). We also show evaluation results under some other example objective functions.

5.2 Normal-case Latency

Fig. 5 shows the CDF of normal-case latency in 2 geo-distributed settings without fail-slow faults. The latency increases in steps due to client locality: clients in the same site have similar latencies. In setting 1, Copilot with and without Ping-pong has 1.58x and 1.88x higher median latency than Multi-Paxos, respectively. In setting 2, Copilot's latency degradation is lower but still has 1.17× (with Ping-pong) and 1.83× (without Ping-pong) higher median latency. Avicenna achieves the same latency as Multi-Paxos because both use a single leader to order commands, yielding a leader-commit latency of 2 message delays. Avicenna's shadow processing is independent of real processing and requires no extra messages for coordination. In EPaxos, each client proposes its commands to its co-located replica. For EPaxos with 0.01 skewness (a nearly uniform distribution), almost all commands can be committed in the fast path, leading to lower median latency than Multi-Paxos. With 0.99 skewness (medium-high skew), the conflict rate increases and some batches are committed in the slow path, EPaxos shows 1.89× and 1.67× higher median latency than Multi-Paxos in setting 1 and 2, respectively. With 10.0 skewness (extremely skewed and unrealistic, but trying to match the 100% conflict setting from the original EPaxos paper), more than 90% of batches are committed in the slow path, leading to 2.22× and 2.17× higher median latency than Multi-Paxos in setting 1 and 2, respectively.

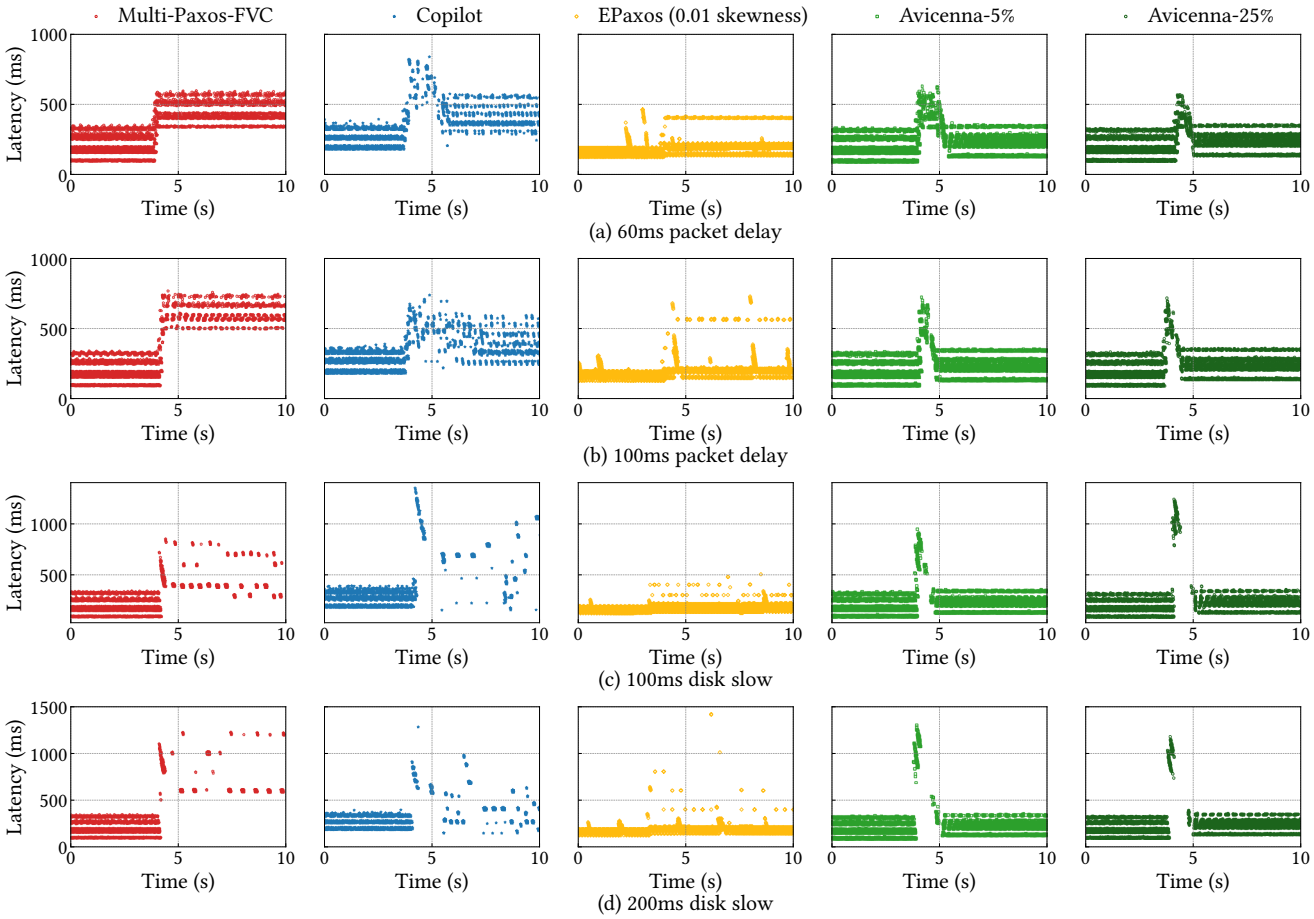

Figure 6. Latency for packet delays and disk slowdowns.

5.3 Fail-Slow Fault Tolerance

In this subsection, we inject fail-slow faults of different kinds and severity to evaluate fail-slow fault tolerance. In line with Lu et al. [30], we inject faults at the software-level rather than the hardware level, which avoids uncontrollable factors and allows for reproducible and targeted fault injection. For Multi-Paxos-FVC and Avicenna, which use a single leader, we inject fail-slow faults to the leader. For Copilot, which uses two leaders, faults are injected to one of the two leaders. For EPaxos, where all replicas are egalitarian, we injected faults into one replica.

Each experiment begins with a warmup period without faults of roughly 20 s, followed by the injection of the fault. Results are shown in Fig. 6. Each point in the figure is the client-perceived e2e latency for a command over a 10 s time window, with faults injected at ≈ 4 s.

Packet delay fail-slow faults. In this subsection, we evaluate fail-slow fault tolerance to packet delay scenarios. This can happen due to NIC firmware bugs, NICs overheating, etc. We use the `tc` command to manually inject packet delays. As shown by a study from Microsoft [16], packet delays can range from several milliseconds to seconds. We show packet delays of 60 and 100 ms.

As shown in Fig. 6 (a) and (b), Avicenna tolerates these injected packet delays. After fault injection, all packets sent to or from the real leader are delayed, delaying commands' commits. The shadow leader can still shadow commit commands to a quorum without the real leader, offering normal shadow latency. Counterfactual evaluation then detects that the shadow leader would yield lower client-perceived latency than the real leader and triggers a leader rotation. The temporary latency spikes during leader rotation are comparable to Copilot. We do not observe a substantial difference between Avicenna-5% and Avicenna-25%, which suggests a small portion of commands being shadow processed is sufficient to detect the fail-slow leader.

Copilot also tolerates packet delays because every command traverses two disjoint paths coordinated by two leaders. With ping-pong batching the fast leader still waits out the ping-pong batching timeout, resulting in higher latencies.

Multi-Paxos-FVC does not tolerate these injected packet delays. With 60 ms or 100 ms of injected delays, the leader-master RTT reaches 188 ms and 268 ms, respectively—both below the 280 ms view-change interval—so the fail-slow

leader is undetected and remains in place.

EPaxos does not tolerate packet delays. In EPaxos, each command is designated to one replica, which is responsible for ordering and committing that command. In this experiment, each client submits commands to its co-located replica. A slow replica delays all commands designated to it. Commands designated to non-slow replicas are affected if their fast quorums include the slow replica; otherwise, they are unaffected. Thus, unlike Multi-Paxos, where a slow leader delays all commands, a slow replica in EPaxos only delays a subset of commands. With a skewness of 0.01 shown in Fig. 6 (a) and (b), EPaxos shows its performance upper bound for slowdown tolerance.

Disk fail-slow faults. In this subsection, we evaluate fail-slow fault-tolerance in disk write scenarios. RSMs rely on writing critical state (e.g., log entries) to disk to tolerate crash failures. As a result, if disk writes become slow, the replica will be fail-slow. To simulate disk write behaviors, we inject software-level delays using `sleep()` calls at points where writing to disk is required to maintain crash fault tolerance. As a recent study [32] shows disk access delay can range from several milliseconds to several seconds, we set the fail-slow severity to be 100 and 200 ms.

As shown in Fig. 6 (c) and (d), Avicenna tolerates these injected disk slows. After fault injection, the real leader delays each command when syncing to disk before broadcasting Accepts, delaying commits in real processing. The shadow leader can still shadow commit in a quorum without the real leader. Counterfactual evaluation then detects that shadow latency is lower than real latency and triggers a leader rotation. We observe larger latency spikes here than under packet delay. With a disk slowdown, each command is delayed at the start of real processing and can be observed by clients only after commit. By contrast, packet delays affect a command every time it traverses the real leader, so the impact is surfaced earlier by delayed commands ordered before the fault but committed after it.

Copilot also tolerates disk slowdowns but exhibits much higher latency spikes than Avicenna. Multi-Paxos-FVC does not tolerate injected disk slowdowns. Disk slowdowns behave as gray failures, where faults significantly impact command processing, but the leader still replies to pings normally. As a result, the view-change timer never expires and view change is never triggered. EPaxos does not tolerate disk slowdowns for the same reasons as the packet delay scenario. EPaxos also shows its performance upper bound of slowdown tolerance in Fig. 6 (c) and (d) with a skewness of 0.01.

5.4 Early-report and Short-path Optimizations

In this subsection, we evaluate the benefit of the early-report and short-path optimizations (Section 4.4). Each point in Fig. 7 is the client-perceived latency for a command. As

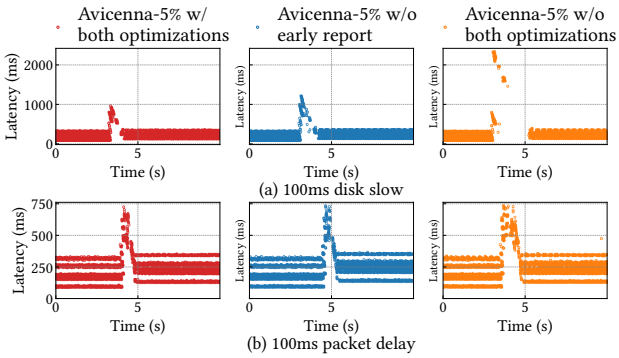

Figure 7. Benefit of the early-report and short-path optimizations. We use Avicenna-5% and inject 100 ms disk slow and 100 ms packet delay to the leader.

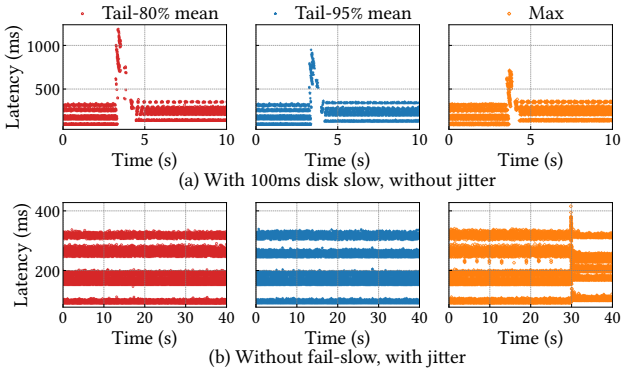

Figure 8. Avicenna-5% with different objective functions.

shown in Fig. 7 (a), with a 100 ms disk slow injected to the leader, Avicenna-5% with both optimizations reacts fastest and exhibits the smallest latency spikes. When disabling early-report optimization, the reaction time and latency spikes during rotation both increase. Disabling the short-path mechanism further amplifies the latency spikes and prolongs the transition. This is because these optimizations expedite fail-slow detection and disabling them extends the leadership of the fail-slow leader. Note that the early-report mechanism is built on short-path optimization, so when disabling short-path, both optimizations are disabled.

In Fig. 7 (b), with a 100 ms packet slow injected on the leader, disabling early-report yields no visible change, and turning off short-path slightly increases the transition. Unlike a disk slowdown, packet delays impact a command every time it goes through the leader so fail-slow faults can surface quickly even without these optimizations.

5.5 Objective Functions

In this subsection, we show how Avicenna behaves with different objective functions. We use $A = \text{Tail}_{80}$, $A = \text{Tail}_{95}$, and $A = \max$ as examples. Each point in Fig. 8 is the client-perceived latency of a command.

As shown in Fig. 8 (a), when injecting a 100 ms disk slowdown at the leader, $A = \max$ detects the fail-slow fault and

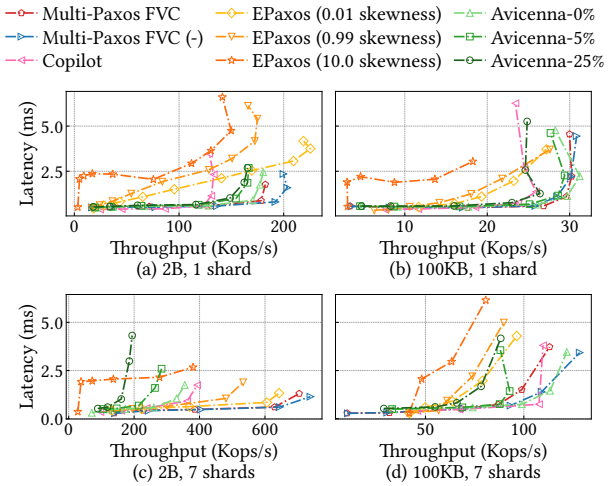

Figure 9. Throughput and latency in a single datacenter.

triggers a leader rotation the fastest since it only considers the highest real and shadow latencies.

In Fig. 8 (b), there is no fail-slow fault but there is system-wide jitter. We manually add jitter to all replicas after warmup: every 3 s we add 150 ms delays for packets to and from clients for a duration of 200 ms. A = max triggers leader rotation although there is no fail-slow fault. In contrast, $A = \text{Tail}_{80}$ and $A = \text{Tail}_{95}$ do not trigger leader rotation.

Overall, these results highlight the trade-off in objective-function choice between reactivity to the fail-slow leader and sensitivity to noise and jitters.

5.6 Throughput-Latency Trade-Off

In this subsection, we show the throughput-latency trade-off between Avicenna and Multi-Paxos-FVC for a single datacenter deployment. We show results on both 1-shard and 7-shard deployments. A *shard* is an independent data partition coordinated by its own RSM instance. The 7-shard deployment is in line with some real-world deployments [2, 3, 12] where a single cluster of machines hosts multiple independent data partitions simultaneously. In this setup, leadership is distributed to better utilize host machine resources. For example, in Multi-Paxos, machine-1 runs the leader process for shard-1 and follower processes for other shards; machine-2 runs the leader process for shard-2 and follower processes for other shards; and so on. Processes for different shards on the same machine operate independently. Shards are independent from each other and each client only interacts with one shard.

In Multi-Paxos-FVC, the leader sends Accepts to all replicas, and all replicas reply with an AcceptOK, resulting in higher message complexity than Avicenna-0% where standbys are not sent Accepts. To enable a fair comparison, we also implement Multi-Paxos-FVC(-), where the leader sends Accepts to only $f + 2$ replicas (including itself).

Fig. 9 shows results for lightweight 2B writes in (a) and (c) as well as heavier-weight commands that allocate and then write 100 KB in (b) and (d). For lightweight operations that use many shards per machine to saturate all cores (c) Avicenna has one third of the throughput of Multi-Paxos. This is the tradeoff of Avicenna: it requires a given deployment to use three times as many machines (that run more shards) to provide a target throughput in exchange for its fail-slow tolerance. This tradeoff is not necessary, however, in three situations. First, if the target throughput is less than what Avicenna can achieve using one set of replicas, e.g., $\leq$ 220 Kops/sec. Second, when heavy-weight commands are used (b,d), Avicenna has comparable throughput to Multi-Paxos. Third, for a single shard (a) its throughput is comparable to Multi-Paxos even with lightweight commands. We also run the 2B-writes execution with 3 and 5 replicas, shown in Appendix B.

6 Discussion

Tolerating multiple slow replicas. Avicenna is designed to tolerate a single slow replica. It detects when the real leader is slow by the shadow leader shadow committing a command with a quorum that excludes the real leader. Generalizing this approach to tolerate two (or more) slowdowns would require shadow processing with at least two (or more) shadow leaders. Ensuring the available quorums for these shadow leaders enables detecting all combinations of two (or more) slowdowns is non-trivial. Thus, we leave designing a protocol that tolerates multiple fail-slow replicas without sacrificing normal-case latency as an interesting avenue for future work.

Counterfactual evaluation for other purposes. In Avicenna, counterfactual evaluation is used to detect a slow leader by continuously comparing real and shadow latencies. We believe counterfactual evaluation can be extended for use in other scenarios as well. For instance, it could be used to identify when changes in network conditions mean that the system should reconfigure which replicas are serving as non-standbys. Or it could be used in scenarios with other constraints, e.g., minimizing latency while respecting a bandwidth constraint on a given location. Or it could be used to detect when it would be useful to modify other configuration parameters, e.g., what replicas are part of a quorum lease for providing low latency reads [35]. It also seems likely to be useful in situations beyond RSMs, such as selecting the participant in a distributed transaction that will serve as the coordinator. Exploring these uses and developing designs for them are exciting avenues for future work.

7 Related Work

Multi-Paxos [24, 25, 52], Raft [42], and Viewstamped Replication [28, 41] are the state-of-practice leader-based protocols, which rely on a single leader to order commands. Due to their simplicity and good performance, many production systems

use them as a building block [1, 7, 8, 12, 22, 49]. These protocols target crash tolerance. Avicenna builds on the principles of these protocols providing the same crash tolerance and latency while also tolerating any single fail-slow replica.

Several other leader-based protocols have been proposed [4, 17, 23, 27, 33, 39, 44–46]. Some of these works adopt a multi-leader approach [4, 33, 39]. Specifically, Avicenna's rotation protocol is inspired by Mencius [33], which uses rotating leaders and noops to optimize for wide-area network deployments. MR99 [36] is a single-instance multi-value consensus protocol that uses an unreliable failure detector [9]. Avicenna's rotation mechanism is inspired by its all-to-all pattern using a deterministic leader order that allows the next leader to begin ordering in 1 message delay from any quorum to the next leader.

EPaxos [34] is a representative leaderless protocol that uses dependencies and fast quorums to optimize for WAN deployments and provide latency lower than Multi-Paxos when operations can take its fast path. Timestamp ordered queuing [51] was later proposed to increase the rate of going through its fast path. Some other leaderless protocols have been proposed in recent years [14, 50, 53]. To the best of our knowledge, in all leaderless protocols, each command is designated to a replica. If a replica is slow, commands designated to it will be affected. Avicenna will have higher latency than EPaxos/other such protocols in the same situations as Multi-Paxos in exchange for its is fail-slow tolerance. An interesting avenue of future work is determining if the latency improvements of leaderless protocols are possible with fail-slow tolerance. The idea of running multiple consensuses in parallel was also used in [6], which uses it to detect a malicious leader.

The fail-slow fault model was introduced by Arpaci-Dusseau et al. [5]. The notion of limpware was introduced and studied by Do et al. [13]. The latest study of the impact of fail-slow faults on distributed system is Lu et al. [30]. Additional work focuses on causes, impact, and mitigation techniques of low-level hardware fail-slow faults [15, 31, 32]. In contrast, Avicenna takes a service-level approach and proposes a fail-slow fault-tolerant RSM.

Copilot [39] is the state-of-the-art slowdown-tolerant RSM protocol, which uses two leaders to concurrently order commands, preventing any slow leader from degrading system latency. However, its slow path leads to high latency in geo-replicated settings. Avicenna is inspired by its two-leader structure, while the shadow leader in Avicenna is purely reactive and independent to avoid interference with the real leader and thus achieve the same normal-case latency as Multi-Paxos. We developed an initial version of a fail-slow fault-tolerant RSM protocol, called Latent Copilot, with one replica acting like the leader in Multi-Paxos, and another replica taking over leadership when it suspects that the leader is slow [38]. Avicenna builds on this initial version to make fail-slow leader detection faster, more accurate, and adaptive to changing environments as well as making leader rotation faster.

An earlier version of Avicenna was described in Christopher Hodsdon's dissertation [19].

8 Conclusion

This paper presents Avicenna, the first RSM protocol that tolerates one fail-slow replica without sacrificing normal-case latency in geo-distributed settings. In normal-case operation, Avicenna relies on a single leader to determine the execution order of commands, naturally tolerating any fail-slow follower. Counterfactual evaluation is employed to enable accurate and adaptive fail-slow detection of the leader. When the leader is suspected to be slow, the fast rotation protocol has the shadow leader take over. Evaluation results show that Avicenna robustly tolerates fail-slow leader faults and maintains the same normal-case latency as Multi-Paxos.

Acknowledgments We thank our shepherd, Andreas Haeberlen, and the anonymous reviewers for their insights and help in refining the ideas of this work.

This material is based upon work supported by the National Science Foundation (NSF) under Grant No. CNS 1827977, and CNS 2321723. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. Some results presented in this paper were obtained using CloudBank [40], which is supported by the NSF under award #1925001. Additionally, this work used AWS computing from CloudBank through allocation CIS260172 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program, which is supported by U.S. NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296.

References

[1] [n. d.]. etcd: A distributed, reliable key-value store for the most critical data of a distributed system. https://etcd.io/.

[2] [n. d.]. How we built a general purpose key value store for Facebook with ZippyDB. https://engineering.fb.com/2021/08/06/core-infra/zippydb/.

[3] [n. d.]. TiKV is a highly scalable, low latency, and easy to use key-value database. https://tikv.org/.

[4] Ailidani Ailijiang, Aleksey Charapko, Murat Demirbas, and Tevfik Kosar. 2019. Wpaxos: Wide area network flexible consensus. *IEEE Transactions on Parallel and Distributed Systems* 31, 1 (2019), 211–223.

[5] Remzi H Arpaci-Dusseau and Andrea C Arpaci-Dusseau. 2001. Fail-stutter fault tolerance. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOs)*.

[6] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. 2013. Rbft: Redundant byzantine fault tolerance. In *2013 IEEE 33rd international conference on distributed computing systems*. IEEE, 297–306.

[7] William J Bolosky, Dexter Bradshaw, Randolph B Haagens, Norbert P Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a High-Performance data store. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[8] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th ACM Symposium on Operating systems design and implementation (SOSP)*.

[9] Tushar Deepak Chandra and Sam Toueg. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)* 43, 2 (1996), 225–267.

[10] Paulo Coelho and Fernando Pedone. 2018. Geographic state machine replication. In *Proceedings of the IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*.

[11] Paulo Coelho and Fernando Pedone. 2021. GeoPaxos+: practical geographical state machine replication. In *Proceedings of the IEEE 40th International Symposium on Reliable Distributed Systems (SRDS)*.

[12] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. 2012. Spanner: Google's Globally-Distributed Database. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[13] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. 2013. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th annual Symposium on Cloud Computing (SoCC)*.

[14] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. 2020. State-machine replication for planet-scale systems. In *Proceedings of the 15th European Conference on Computer Systems (EuroSys)*.

[15] Haryadi S Gunawi, Riza O Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. 2018. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*.

[16] Chuanxiong Guo, Lihua Yuan, Dong Xiang, Yingnong Dang, Ray Huang, Dave Maltz, Zhaoyi Liu, Vin Wang, Bin Pang, Hua Chen, et al. 2015. Pingmesh: A large-scale system for data center network latency measurement and analysis. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (SIGCOMM)*.

[17] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. 2014. Rex: Replication at the speed of multi-core. In *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*.

[18] Maurice P Herlihy and Jeannette M Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[19] Christopher Hodsdon. 2023. *Stronger Abstractions and Performance Guarantees for Building Strongly Consistent Distributed Services*. Ph. D. Dissertation. Princeton University.

[20] Peng Huang, Chuanxiong Guo, Jacob R Lorch, Lidong Zhou, and Yingnong Dang. 2018. Capturing and enhancing in situ system observability for failure detection. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[21] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. 2017. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOs)*.

[22] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC)*.

[23] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about eve: Execute-Verify replication for Multi-Core servers. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[24] Jonathan Kirsch and Yair Amir. 2008. Paxos for System Builders: an overview. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*.

[25] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Transactions on Computer Systems* 16, 2 (1998), 133–169.

[26] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (1982), 382–401.

[27] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. 2016. Just say NO to paxos overhead: Replacing consensus with network ordering. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

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

[29] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, detecting and localizing partial failures in large system software. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[30] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. 2025. One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems. In *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[31] Ruiming Lu, Erci Xu, Yiming Zhang, Fengyi Zhu, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Jiwu Shu, Minglu Li, et al. 2023. Perseus: A Fail-Slow detection framework for cloud storage systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies (FAST)*.

[32] Ruiming Lu, Erci Xu, Yiming Zhang, Zhaosheng Zhu, Mengtian Wang, Zongpeng Zhu, Guangtao Xue, Minglu Li, and Jiesheng Wu. 2022. NVMe SSD failures in the field: the Fail-Stop and the Fail-Slow. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*.

[33] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. 2008. Mencius: building efficient replicated state machines for WANs. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[34] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*.

[35] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2014. Paxos quorum leases: Fast reads without sacrificing writes. In *Proceedings of the ACM Symposium on Cloud Computing*. 1–13.

[36] Achour Mostéfaoui and Michel Raynal. 1999. Solving consensus using Chandra-Toueg's unreliable failure detectors: A general quorum-based approach. In *International Symposium on Distributed Computing*. Springer, 49–63.

[37] Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2018. Dpaxos: Managing data closer to users for low-latency and mobile applications. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*.

[38] Khiem Ngo. 2021. *Tolerating Slowdowns in Replication State Machines*. Ph. D. Dissertation. Princeton University.

[39] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating slowdowns in replicated state machines using copilots. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[40] Michael Norman, Vince Kellen, Shava Smallen, Brian DeMeulle, Shawn Strande, Ed Lazowska, Naomi Alterman, Rob Fatland, Sarah Stone, Amanda Tan, et al. 2021. Cloudbank: Managed services to simplify cloud access for computer science research and education. In *Practice and Experience in Advanced Research Computing 2021: Evolution Across All Dimensions*. 1–4.

[41] Brian M Oki and Barbara H Liskov. 1988. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX annual technical conference (ATC)*.

[43] Biswaranjan Panda, Deepthi Srinivasan, Huan Ke, Karan Gupta, Vinayak Khot, and Haryadi S Gunawi. 2019. IASO: A Fail-Slow Detection and Mitigation Framework for Distributed Storage Services. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*.

[44] Seo Jin Park and John Ousterhout. 2019. Exploiting commutativity for practical fast replication. In *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[45] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. 2015. Designing distributed systems using approximate synchrony in data center networks. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[46] Fedor Ryabinin, Alexey Gotsman, and Pierre Sutra. 2024. SwiftPaxos: Fast Geo-Replicated State Machines. In *Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[47] Richard D Schlichting and Fred B Schneider. 1983. Fail-stop processors: An approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems (TOCS)* 1, 3 (1983), 222–238.

[48] Fred B Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[49] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM International Conference on Management of Data (SIGMOD)*.

[50] Pasindu Tennage, Cristina Basescu, Lefteris Kokoris-Kogias, Ewa Syta, Philipp Jovanovic, Vero Estrada-Galinanes, and Bryan Ford. 2023. QuePaxa: Escaping the tyranny of timeouts in consensus. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*.

[51] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. EPaxos revisited. In *Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*.

[52] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos made moderately complex. *Comput. Surveys* 47, 3 (2015), 1–36.

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

A Proof of Correctness

In this section, we prove the correctness of Avicenna. Our proof reasons about the real process because the real and shadow processes are completely independent and only the real process impacts correctness.

We first describe correctness-related protocols in Avicenna in pseudo code Section A.1, then we prove the safety Section A.2 and liveness Section A.3 of Avicenna by referring to the pseudo code.

A.1 Pseudo Code

In this subsection, we first provides some common notations used hereafter, then describes protocols in Avicenna that are correctness-related in pseudo code.

Notation:
phase: $p (p \geq 0)$
replica: $r_i (0 \leq i \leq N-1)$
real leader at phase p: $r_{rl(p)}$
command: $a_i (i \geq 0)$
client who proposes a_i: c_{a_i}
the set of non-standbys in phase p: $\mathcal{N}_p$
execution order at replica r_i that r_i executes a_m before a_n: $a_m \xrightarrow{\text{exec}_i} a_n$
real-time order that a_m completes before a_n starts: $a_m \xrightarrow{\text{rto}} a_n$

Algorithm 1 Replica

```
1:  class Replica
2:    variables:
3:      id : int                                  ▷ replica Id
4:      phase : int                               ▷ current phase
5:      realLog : array
6:      sentRotate : map[int][bool]  ▷ whether the replica has broadcast ROTATE in a phase
7:      progressTimer : timer                     ▷ track progress
8:    methods:
9:      Init()
10:     DoRotation()                              ▷ Handle rotation logic
11:     HandleMsg(msg)                            ▷ Handle messages
12: end class
```

Algorithm 2 Replica Logic (r Replica)

```
r.Init()
while true do
    switch Event do
        case latency-triggered rotation:
            if r.sentRotate[phase] = false then
                r.DoRotation(null)
            end if

        case r.progressTimer.expire():
            if r.sentRotate[phase] = false then
                r.DoRotation(null)
            end if

        case msg:
            r.HandleMsg(msg)

    end switch
end while
```

Algorithm 3 DoRotation (msg)

```
if r.sentRotate[r.phase] == false then
    ROTATE.sender = r.Id
    ROTATE.phase = r.phase
    ROTATE.rotateLog = r.realLog
    r.sentRotate[r.phase] = true
    Broadcast ROTATE
    if r ∈ N_{r.phase} then
        r.rotateLog.append(r.realLog)
    end if
end if
if msg && msg.sender ∈ N_{r.phase} then
    r.rotateLog.append(msg.rotateLog)
end if
logRequired := r.IsArmageddon() ? f+1 : 2
if r.rotateLog.size() == logRequired then
    LogMerge(r.rotateLog)
    GoToPhase(r.phase + 1)
end if
```

Algorithm 4 HandleMsg (msg)

```
switch msg.type do
    case ROTATE:
        if msg.phase == r.phase then
            r.DoRotation(msg)
        end if

    case PROPOSE:
        if r.sentRotate[r.phase] == false then
            r.HandlePropose(msg)                ▷ leader only
        end if

    case ACCEPT:
        if msg.phase == r.phase
           && r.sentRotate[r.phase] == false
           && (r.realLog[msg.entry] == {}
               || r.realLog[msg.entry].phase < msg.phase)
        then
            r.realLog[msg.entry].cmd = msg.cmd
            r.realLog[msg.entry].status = Accepted
            r.realLog[msg.entry].phase = msg.phase
            send ACCEPTOK to msg.sender
        end if

    case COMMIT:
        if msg.phase == r.phase
           && (r.realLog[msg.entry] == {}
           || r.realLog[msg.entry].status ≠ Committed)
        then
            r.progressTimer.reset()
            r.realLog[msg.entry].cmd = msg.cmd
            r.realLog[msg.entry].status = Committed
            r.realLog[msg.entry].phase = msg.phase
        end if

    case LONGACCEPT:
        if msg.phase == r.phase
        && r.sentRotate[r.phase] == false
        then
            r.handleLongAccept(msg)
        end if

    case ACCEPTOK:                              ▷ leader only
        if msg.phase == r.phase then
            r.handleAcceptOK(msg)
        end if

end switch
```

Algorithm 5 handlePropose (msg)

```
1:  k := len(r.realLog)
2:  r.realLog[k].cmd = msg.cmd
3:  r.realLog[k].phase = r.phase
4:  r.realLog[k].status = Received
5:  Accept.cmd = msg.cmd
6:  Accept.sender = r.Id
7:  Accept.entry = k
8:  Accept.phase = r.phase
9:  Broadcast Accept to all non-standbys
10: r.realLog[k].status = Accepted
11: r.ackNum[k] = 1
```

Algorithm 6 handleLongAccept (msg)

```
1:  while e := msg.entry do
2:      if r.realLog[e.index].status == Committed then
3:          continue
4:      end if
5:      if r.realLog[e.index]=={}
            || r.realLog[e.index].phase < e.phase then
6:          r.realLog[e.index].cmd = e.cmd
7:          r.realLog[e.index].status = e.status
8:          r.realLog[e.index].phase = e.phase
9:          Send AcceptOK to msg.sender
10:     end if
11: end while
```

Algorithm 7 handleAcceptOK (msg)

```
1:  if r.realLog[msg.entry].status == Committed then
2:      return
3:  end if
4:  r.ackNum[msg.entry]++
5:  if r.ackNum[msg.entry] >= f+1 then
6:      Commit.entry = msg.entry
7:      Commit.cmd = r.realLog[msg.entry].cmd
8:      Commit.phase = r.phase
9:      r.realLog[msg.entry].status = Committed
10:     Broadcast Commit
11: end if
```

Algorithm 8 LogMerge(RotateLog)

```
1:  i := r.minimumUncommitted
2:  while i <= r.maximumLogIndex do
3:      if r.realLog[i].status == Committed then
4:          i++
5:          continue
6:      end if
7:      A := {e ∈ r.rotateLog | e.index == i}
8:      for all e ∈ A do
9:          if ∃e that e.status == Committed then
10:             r.realLog[i].cmd = e.cmd
11:             r.realLog[i].status = Committed
12:             r.realLog[i].phase = e.phase
13:             break
14:         end if
15:         B := {e ∈ A | e.status == Accepted}
16:         if B ≠ {} then
17:             e* := arg max_{e∈B} e.phase
18:             r.realLog[i].cmd = e*.cmd
19:             r.realLog[i].status == Accepted
20:             r.realLog[i].phase = e*.phase
21:         end if
22:     end for
23:     i++
24: end while
```

Algorithm 9 GoToPhase (phase)

```
1:  r.phase = phase
2:  r.sentRotate[phase] = false
3:  if r is real leader then
4:      i := r.minimumUncommitted
5:      LongAccept := {}
6:      while i <= r.maximumLogIndex do
7:          e := {}
8:          e.cmd = r.realLog[i].cmd
9:          e.index = i
10:         e.phase = r.phase
11:         LongAccept.append(e)
12:     end while
13:     Broadcast LongAccept
14: end if
```

A.2 Proof of Safety

In this subsection, we prove that Avicenna provides linearizability: client commands are (1) executed in some total order, and (2) this order is consistent with the real-time order, i.e., if command a_m completes before command a_n begins in real-time, then a_m must be ordered before a_n.

We first introduce some necessary lemmas, then prove the total order, and finally prove the real-time ordering.

Lemma 1. *If there exist replicas r_i, r_j and an index k such that*
r_i.realLog[k].status ∈ {Accepted, Committed}
∧ $r_j$realLog[k].status ∈ {Accepted, Committed}
∧ r_i.realLog[k].cmd ≠ r_j.realLog[k].cmd,
then we have r_i.realLog[k].phase ≠ r_j.realLog[k].phase.

Proof.

In Avicenna, only the real leader sends Accept and Commit messages. Thus, for any *msg*, if *msg*.phase = *p* && *msg*.type ∈ {Accept, Commit, LongAccept}, then *msg*.sender = $r_{rl(p)}$. For a LongAccept, any entry with status Accepted is equivalent to an Accept message; any entry with status Committed is equivalent to a Commit message. Therefore, without loss of generality, we can only care about *msg* that *msg*.type ∈ {Accept, Commit}.

According to line 13 and line 21 in Alg. 4, a replica r processes Accept and Commit messages from $r_{rl(p)}$ if and only if r.phase == p; otherwise, r ignores such messages.

For a replica r, denote the procedure of line 12–18 in Alg. 4 and line 5–10 in Alg. 6 (if e.status == Accepted) as Proc{r, Accept, Accept.entry}; denote the procedure of line 20–26 in Alg. 4 and line 5–10 in Alg. 6 (if e.status == Committed) as Proc{r, Commit, Commit.entry}. Specifically, Proc{r, *msg*, *msg*.entry}(*msg* ∈ {Accept, Commit}) ⇒ r.realLog[*msg*.entry].cmd = *msg*.cmd, and r.realLog[*msg*.entry].phase = *msg*.phase.

Consider a replica r and an index k where r.realLog[k].status ∈ {Accepted, Committed}, the following two cases cover all possibilities of how r set this log entry.

Case 1: If r.realLog[k].phase = r.phase = p.
In this case, r sets this log entry by Proc{r, *msg*, *msg*.entry}, where *msg* ∈ {Accept, Commit}, *msg*.entry = k, r.realLog[k].cmd = *msg*.cmd, *msg*.phase = r.phase = r.realLog[k].phase = p, and *msg*.sender = $r_{rl(p)}$. $r_{rl(p)}$ can be r itself.

Case 2: If p_m = r.realLog[k].phase < r.phase = p_n.
In this case, r sets this log entry from log merge during rotation from phase $p_n - 1$ to p_n (Alg. 8). Let r get this entry from r' (r' can be r itself). According to Alg. 8, log merge does not change the position of the log entry. Hence, r'.realLog[k] = r.realLog[k].
If $p_m = p_n - 1$, by *case 1*, r' set this log entry by Proc{r', *msg*, *msg*.entry}, where *msg*.entry = k, r.realLog[k].cmd = r'.realLog[k].cmd = *msg*.cmd, *msg*.phase = r'.realLog[k].phase = r.realLog[k].phase = p_m, *msg*.sender = $r_{rl(p_m)}$. $r_{rl(p_m)}$ can be r' itself.
If $p_m < p_n - 1$, then r' got this entry from log merge during rotation from phase $p_n - 2$ to $p_n - 1$. Let r' get this entry from r'' (r'' can be r' itself). By induction, there exists r^* that set this entry by Proc{r^*, *msg*, *msg*.entry}, where *msg*.entry = k, r^*.realLog[k].cmd = r.realLog[k].cmd = *msg*.cmd, *msg*.phase = r^*.realLog[k].phase = r.realLog[k].phase = p_m, *msg*.sender = $r_{rl(p_m)}$. $r_{rl(p_m)}$ can be r^* itself.

Based on *case 1* and *case 2*, we have

$$\begin{aligned} &r.\text{realLog}[k].\text{status} \in \{\text{Accepted, Committed}\} \\ &\Rightarrow \exists\, r^\star, msg: \\ &\Big(\text{Proc}\{r^\star, msg, k\} \wedge r.\text{realLog}[k].\text{cmd} = msg.\text{cmd} \wedge \\ &msg.\text{phase} = r.\text{realLog}[k].\text{phase} \wedge msg.\text{entry} = k\Big). \end{aligned} \tag{1}$$

Now, if $\exists\, r_i, r_j, k$ where
r_i.realLog[k].status ∈ {Accepted, Committed}
∧ r_j.realLog[k].status ∈ {Accepted, Committed}
∧ r_i.realLog[k].cmd ≠ r_j.realLog[k].cmd, we prove r_i.realLog[k].phase ≠ r_j.realLog[k].phase by contradiction.

By (1), we know

$$\begin{aligned} &\exists r_{i1}, msg_i, r_{j1}, msg_j: \\ &\Big(\text{Proc}\{r_{i1}, msg_i, k\} \wedge r_i.\text{realLog}[k].\text{cmd} = msg_i.\text{cmd} \wedge \\ &msg_i.\text{phase} = r_i.\text{realLog}[k].\text{phase} \wedge msg_i.\text{entry} = k\Big); \\ &\Big(\text{Proc}\{r_{j1}, msg_j, k\} \wedge r_j.\text{realLog}[k].\text{cmd} = msg_j.\text{cmd} \wedge \\ &msg_j.\text{phase} = r_j.\text{realLog}[k].\text{phase} \wedge msg_j.\text{entry} = k\Big). \end{aligned} \tag{2}$$

By r_i.realLog[k].cmd ≠ r_j.realLog[k].cmd and (2), we have msg_i.cmd ≠ msg_j.cmd.

Suppose r_i.realLog[k].phase = r_j.realLog[k].phase = p, by only the real leader for a phase sending Accept and Commit messages, we have msg_i.sender = msg_j.sender = $r_{rl(p)}$.

Therefore, the real leader $r_{rl(p)}$ puts different commands to the same log entry, which is in contradiction to Avicenna's protocol that a real leader puts only one command per log entry.

∴ r_i.realLog[k].phase ≠ r_j.realLog[k].phase. □

Lemma 2. *If ∃ r, k, a_m, p_1 such that r.realLog[k].cmd = a_m, r.realLog[k].status = Committed, r.realLog[k].phase = p_1, then ∀r_j with r_j.phase = p_2($p_2 > p_1$), we have r_j.realLog[k].cmd = a_m.*

Proof.

Denote A as the set of replicas that accept a_m in phase p_0,

then we have $A \subset \mathcal{N}_{p_0}$.

$\because$ Phases grow monotonically

$\therefore p_0 \leq p_1$.

A real leader only commits a command after at least $f+1$ non-stanbys accepts that command. Thus, $|A| \geq f+1$.

$\therefore \forall r \in A$, we have
r.realLog$[k]$.cmd $= a_m$, r.realLog$[k]$.phase $= p_0$, r.realLog$[k]$.status $\in$ {Accepted, Committed}.

Consider any r that moves from phase p_0 to p_0+1.

Let B be the set of replicas from which r merges logs.

r only merges log from non-standbys $\Rightarrow B \subset \mathcal{N}_{p_0}$.

If p_0 is a normal phase, $|\mathcal{N}_{p_0}| = f+2, |B| \geq 2$. Thus, $|A| + |B| \geq f+3 > |\mathcal{N}_{p_0}|$

If p_0 is an Armageddon phase, $|\mathcal{N}_{p_0}| = 2f+1, |B| \geq f+1$. Thus, $|A| + |B| \geq 2f+2 > |\mathcal{N}_{p_0}|$

$C := \{r \mid r \in A \wedge r \in B\}$, then we have $|C| \geq 1$

$$\begin{aligned} \therefore\ & \exists\, r_i \in B \text{ that } r_i.\text{realLog}[k].\text{cmd} = a_m \\ & \wedge\ r_i.\text{realLog}[k].\text{phase} = p_0 \\ & \wedge\ r_i.\text{realLog}[k].\text{status} \in \{\text{Accepted}, \text{Committed}\} \end{aligned} \tag{3}$$

Equipped with this intersection in C of at least one replica with knowledge of a_m, we now examine the three possible log merging cases and show for each that $\forall r_i$ with r_i.phase $= p_0+1$, we have r_i.realLog$[k]$.cmd $= a_m$.

Case 1: $\exists\, r_{i_1} \in B$ that r_{i_1}.realLog$[k]$.status = Committed.

In this case, r_{i_1}.realLog$[k]$.cmd $= a_m$. We prove this by contradiction.

Assume to contradict r_{i_1}.realLog$[k]$.cmd $= a_n (a_m \neq a_n)$. There are two subcases.

If r_{i_1}.realLog$[k]$.phase $= p_0$, then the real leader $r_{rl(p_0)}$ commits a_n to $r_{rl(p_0)}$.realLog$[k]$ (because only the real leader for a phase can commit a command in that phase). By premise, a_m is accepted at phase p_0 by replicas in A. So, the real leader $r_{rl(p_0)}$ also puts a_m in $r_{rl(p_0)}$.realLog$[k]$. This results in contradiction against the protocol that a real leader only puts one command per log entry.

In the other case, $p_m = r_{i_1}$.realLog$[k]$.phase $< p_0$. The real leader $r_{rl(p_m)}$ commits a_n at $r_{rl(p_m)}$.realLog$[k]$ in phase p_m. If $p_m = p_0 - 1$, according to line 9–14 in Alg. 8, all replicas in phase p_0, particularly the real leader $r_{rl(p_0)}$, have a_n at its real log entry k. If $p_m = p_0 - 2$, according to line 9–14 in Alg. 8, all replicas in phase $p_0 - 1$ will have a_n at its log entry k, and all replicas in phase p_0, particularly the real leader $r_{rl(p_0)}$, will have a_n at its log entry k. By induction, if $p_m < p_0$, $r_{rl(p_0)}$ will have a_n at its log entry k. By premise, a_m is accepted at p_0 by replicas in A. So, the real leader $r_{rl(p_0)}$ also puts a_m in $r_{rl(p_0)}$.realLog$[k]$. This results in contradiction against the protocol that a real leader only puts one command per log entry.

Therefore, r_{i_1}.realLog$[k]$.cmd must be a_m. Thus, in this case, according to line 9–14 in Alg. 8, r sets r.realLog$[k]$.cmd $= a_m$.

Case 2: $\exists\, r_{i_2} \in B$ that r_{i_2}.realLog$[k]$.status = Accepted but r_{i_2}.realLog$[k]$.cmd $\neq a_m$.

By (3) and Lemma 1, we have r_{i_2}.realLog$[k]$.phase $\neq p_0$.

$\because p_0$ is the current phase and phase grows monotonically.

$\therefore r_{i_2}$.realLog$[k]$.phase $< p_0$.

In this case, according to line 15–21 in Alg. 8 And a symmetric induction to Case 1, r sets r.realLog$[k]$.cmd $= a_m$

Case 3: For each $r \in B$, r.realLog$[k]$.status = Accepted $\iff$ r.realLog$[k]$.cmd $= a_m$.

In this case, according to lines 15–21 in Alg. 8, r sets r.realLog$[k]$.cmd $= a_m$.

$\therefore$ Based on case 1, 2, and 3, every r that successfully moves into p_0+1, we have r.realLog$[k]$.cmd $= a_m$.

$\therefore$ $\forall r$ that moves from p_0+1 to p_0+2, let it merge logs from $\{r_{i_1}, r_{i_2}, ..., r_{i_n}\}$. Then $\forall r_i \in \{r_{i_1}, r_{i_2}, ..., r_{i_n}\}$, we have r_i.realLog$[k]$.cmd $= a_m$. Therefore, r can only set r.realLog$[k]$ to a_m.

By induction, $\forall r$ in p_2 where $p_2 > p_0$, r.realLog$[k]$.cmd $= a_m$.

$\because p_0 \leq p_1$

$\therefore \forall r$ in p_2 where $p_2 > p_1$, r.realLog$[k]$.cmd $= a_m$. □

Lemma 3. *Suppose at some time there exist a replica r and an index k such that $\forall i \leq k$, r.realLog$[i]$.status = Committed. Let $p_1 := r$.phase at that time. Then from now on, $\forall p \geq p_1$, we have $len(r_{rl(p)}.\text{realLog}) \geq k$ and $\forall\ i \leq k$, $r_{rl(p)}$.realLog$[i]$.cmd $= r$.realLog$[i]$.cmd.*

Proof.

$\because p_1 = r$.phase at this moment.

$\therefore \forall i$, we have r.realLog$[i]$.phase $\leq p_1$.

In particular, for each $i \leq k$, we have r.realLog$[i]$.phase $\leq p_1$ and r.realLog$[i]$.status = Committed.

For any $p \geq p_1$ and consider the real leader $r_{rl(p)}$. We show that for every $i \leq k$,

$$r_{rl(p)}.\text{realLog}[i].\text{cmd} = r.\text{realLog}[i].\text{cmd}.$$

There are two cases.

Case 1: r.realLog$[i]$.phase $< p_1$. According to Lemma 2, for all replicas in phase p_1, particularly $r_{rl(p_1)}$, we have $r_{rl(p_1)}$.realLog$[i]$.cmd $= r$.realLog$[i]$.cmd.

Case 2: r.realLog$[i]$.phase $= p_1$. r.realLog$[i]$ is committed at phase p_1 by $r_{rl(p_1)}$, so we have $r_{rl(p_1)}$.realLog$[i]$.cmd $= r$.realLog$[i]$.cmd.

Combining *case 1* and *case 2*, we know when $p = p_1$, for every $i \leq k$, $r_{rl(p)}$.realLog$[i]$.cmd $= r$.realLog$[i]$.cmd.

Applying Lemma 2, we know for $p > p_1$, for every $i \leq k$, $r_{rl(p)}$.realLog$[i]$.cmd $= r$.realLog$[i]$.cmd.

$\therefore \forall\ i \leq k$ and $p \geq p_1$, we have $len(r_{rl(p)}.\text{realLog}) \geq k$, and $r_{rl(p)}$.realLog$[i]$.cmd $= r$.realLog$[i]$.cmd □

Theorem 4. *Total order: If $\exists r_i, a_m \xrightarrow{\text{exec}_i} a_n$, then $\forall r_j, \neg a_n \xrightarrow{\text{exec}_j} a_m$.*

Proof.

Without loss of generality, assume a_m is committed in

phase p_1 and a_n is committed in phase p_2. Then by commits only coming from the real leader in a phase we have $r_{rl(p_1)}$ commits a_m to $r_{rl_{p_1}}$.realLog$[k]$ and $r_{rl(p_2)}$ commits a_n to $r_{rl(p_2)}$.realLog$[t], k \neq t$.

According to Lemma 2, $\forall r$ at phase $p > p_1$, we have r.realLog$[k]$.cmd $= a_m$; $\forall r$ at phase $p > p_2$, we have r.realLog$[t]$.cmd $= a_n$.

$\therefore \forall r$ that have a_m and a_n committed, we have r.phase $\geq \max\{p_1, p_2\}$, so r.realLog$[k]$.cmd $= a_m, r$.realLog$[t]$.cmd $= a_n$.

If $\exists\ r_i$ that $a_m \xrightarrow{\text{exec}_i} a_n$, then we have $k < t$. And thus by the in order execution of the real log we have:

$\therefore \forall r_j$ that executes a_m and a_n, we have $a_m \xrightarrow{\text{exec}_j} a_n$

$\therefore \forall r_j$, we have $\neg a_n \xrightarrow{\text{exec}_j} a_m$ □

Theorem 5. *Real-time order: If $a_m \xrightarrow{\text{rto}} a_n$, then $\forall r_i$, we have $\neg a_n \xrightarrow{\text{exec}_i} a_m$*

Proof.

Case 1: If a_m, a_n are both committed in the same phase. Without loss of generality, they are committed in phase p_1, then the real leader $r_{rl(p_1)}$ commits a_m to $r_{rl(p_1)}$.realLog$[k]$ and a_n to $r_{rl(p_1)}$.realLog$[t], k \neq t$.

$\because a_m \xrightarrow{\text{rto}} a_n$

$\therefore \exists\ r_i$ where r_i.realLog$[k]$.cmd $= a_m$, plus: r_i executes $a_m \xrightarrow{\text{rto}} c_{a_m}$ receives response for $a_m \xrightarrow{\text{rto}} c_{a_n}$ proposes $a_n \xrightarrow{\text{rto}} r_{rl(p_1)}$ receives a_n

$\therefore$ When $r_{rl(p_1)}$ receives $a_n, \exists\ r_i$ that r_i has executed a_m and r_i.realLog$[k]$.cmd $= a_m$

By protocol, each replica executes the command in a log entry only if this entry and all entires before it are committed.

$\therefore \forall h \leq k$, we have r_i.realLog$[h]$.status = Committed and r_i.realLog$[h]$.cmd $\neq a_n$

$\therefore$ According to Lemma 3, we have $len(r_{rl(p_1)}.\text{realLog}) \geq k$, and $\forall h \leq k$, we have $r_{rl(p_1)}$.realLog$[h]$.cmd $= r_i$.realLog$[h]$.cmd $\neq a_n$.

$\therefore$ According to Alg. 5, $r_{rl(p_1)}$ puts a_n at $r_{rl(p_1)}$.realLog$[t], t > k$

$\therefore \forall r_i$ that executes a_m and a_n, we have $a_m \xrightarrow{\text{exec}_i} a_n$

$\therefore \forall r_i$, we have $\neg a_n \xrightarrow{\text{exec}_i} a_m$

Case 2: If a_m and a_n are committed in different phases. Without loss of generality, assume a_m is committed in phase p_1 and a_n is committed in phase $p_2 (p_1 \neq p_2)$, then $r_{rl(p_1)}$ commits a_m to $r_{rl(p_1)}$.realLog$[k]$ and $r_{rl(p_2)}$ commits a_n to $r_{rl(p_2)}$.realLog$[t], k \neq t$.

$\because a_m \xrightarrow{\text{rto}} a_n$

$\therefore \exists\ r_i$ where $r_i$realLog$[k]$.cmd $= a_m$, plus: r_i executes $a_m \xrightarrow{\text{rto}} c_{a_m}$ receives response for $a_m \xrightarrow{\text{rto}} c_{a_n}$ proposes $a_n \xrightarrow{\text{rto}} r_{rl(p_2)}$ orders a_n

$\therefore$ when $r_{rl(p_2)}$ orders $a_n, \exists\ r_i$ that has executed a_m and r_i.realLog$[k]$.cmd $= a_m$

By protocol, each replica executes the command in a log entry only if this entry and all entires before it are committed.

$\therefore$ when $r_{rl(p_2)}$ orders $a_n, \forall j \leq k$, we have r_i.realLog$[j]$.status = Committed and r_i.realLog$[j]$.cmd $\neq a_n$

$\because p_1 \neq p_2$ and phase grows monotonically

$\therefore p_2 > p_1$

$\therefore$ according to Lemma 3, when $r_{rl(p_2)}$ orders $a_n, \forall j \leq k$, we have $r_{rl(p_2)}$.realLog$[j]$.cmd $= r_i$.realLog$[i]$.cmd $\neq a_n$

$\therefore t > k$

$\therefore \forall r_i$ that executes a_m and a_n, we have $a_m \xrightarrow{\text{exec}_i} a_n$

$\therefore \forall r_i$, we have $\neg a_n \xrightarrow{\text{exec}_i} a_m$ □

A.3 Proof of Liveness

In this subsection, we prove that Avicenna guarantees liveness that all client commands eventually complete.

We make the following assumptions in proving liveness:
(1) No more than f replicas are faulty.
(2) A majority of replicas can communicate with each other within a timeout, and messages eventually arrive at their destination before their receiver times out.
(3) A client keeps resending its command after a timeout until its command is successfully completed.

Lemma 6. *If in a phase, there are fewer than $f+1$ non-faulty non-standbys, or the real leader is faulty, then at least $f+1$ non-faulty replicas will eventually do rotation and go to the next phase.*

Proof.

We first show that in this situation rotation will eventually be triggered by the progress timer. Then we show at least $f+1$ non-faulty replicas can finish the rotation and go to the next phase. To show rotation will be triggered, we divide this situation into two cases which cover all possibilities:

Case 1: The real leader is faulty.

In Avicenna, the real leader is the only replica that sends Commit messages. In this case, no Commit message will be sent.

Case 2: The real leader is not faulty but there are fewer than $f+1$ non-standbys.

In Avicenna, the real leader only sends a Commit message after collecting AcceptOKs from at least $f+1$ non-standbys (including itself). In this case, the real leader can no longer collect enough AcceptOKs, thus, no Commit message will be sent.

In both *case 1* and *case 2*, there will be no Commit message sent from the current real leader. In Avicenna, receiving a Commit message is the only time a replica resets its progress timer. Thus, for a non-faulty replica, its progress timer will eventually expires and it will start doing rotation.

We have shown that rotation will eventually be triggered. Now we show at least $f+1$ non-faulty replicas will eventually finish the rotation and go to the next phase. By assumption

(1) and the fact that there are at most $f-1$ standbys, there are at least 2 non-faulty non-standbys. Thus by assumption (2), after starting rotation, at least $f+1$ non-faulty replicas will eventually collect logs from at least 2 non-standbys. After log merging, they will go to the next phase.

$\therefore$ In this situation, at least $f+1$ non-faulty non-standbys will eventually do rotation and go to the next phase. □

Lemma 7. *The rotation procedure makes progress and eventually terminates. The definition of termination is that there are at least $f+1$ non-faulty non-standbys in a phase, one of which is the real leader.*

Proof.

We first show that if a replica starts doing rotation, at least $f+1$ non-faulty replicas will eventually start doing rotation and go to the next phase. Then we show that this procedure will eventually terminate by having at least $f+1$ non-faulty non-standbys in a phase, one of which is the real leader.

If a replica broadcasts a ROTATE message, by assumption (1) and (2), at least $f+1$ non-faulty replicas will start the rotation procedure. At least 2 of them are non-standbys since there are at most $f-1$ standbys. Thus, at least $f+1$ non-faulty replicas will receive logs from at least 2 non-standbys. Then they will finish log merging and go to the next phase. The possibilities of the new phase can be covered by the following two cases:

Case 1: In the new phase, there are at least $f+1$ non-faulty non-standbys, one of which is the real leader.

In this case, the rotation procedure terminates by definition.

Case 2: In the new phase, there are fewer than $f+1$ non-faulty non-standbys, or the real leader is faulty.

Without loss of generality, denote the old phase as p_0. By Lemma 6, at least $f+1$ non-faulty replicas will eventually do rotation and go to phase p_0+1. If phase p_0+1 satisfies *case 1*, the rotation procedure terminates by definition; otherwise, by Lemma 6, at least $f+1$ non-faulty replicas will eventually do rotation and go to phase p_0+2. Next, we prove that *case 1* will eventually be satisfied at a phase $p_1 > p_0$.

In Avicenna, the role of each replica in a phase is deterministic. Phase p starts from 0 and grows monotonically. Phase p is an Armageddon phase (all $2f+1$ replicas are non-standbys) if $(p+1) \mod (N+1) = 0$; otherwise, phase p is non-Armageddon ($f+2$ replicas are non-standbys, others are standbys). Thus, every $N+1$ phases in Avicenna is an Armageddon phase. In an Armageddon phase, the replica r with r.Id $= \lfloor p/(N+1) \rfloor \mod N$ is the real leader in that phase. Therefore, given any $f+1$ non-faulty replicas, there are infinite number of phases p where $\lfloor p/(N+1) \rfloor \mod N$ is one of the replica's r.Id.

Therefore, starting from any phase p_0, *case 1* will be eventually satisfied at a phase $p_1 > p_0$.

$\therefore$ The rotation procedure will eventually make progress and terminate by having at least $f+1$ non-faulty non-standbys in a phase, one of which is the real leader. □

Theorem 8. *Avicenna makes progress by eventually committing and executing a client command, and replying to the client.*

Proof.

In Avicenna, a client sends its command c to both the real and shadow leaders. The shadow leader keeps track of received but uncommitted commands.

First, we show that command c will eventually be committed. Then, we show that command c will eventually be executed and the client will receive a reply.

Case 1: There are at least $f+1$ non-faulty non-standbys, one of which is the real leader. And there is no rotation triggered in a reasonable amount of time after the client sending c.

In this case, by assumption (1) and (2), the real leader will eventually collect ACCEPTOKs from $f+1$ non-standbys (including itself) and commit the command c.

Case 2: After the client sends c, rotations happen.

In this case, according to Lemma 7, the rotation procedure will eventually terminate and there are at least $f+1$ non-faulty non-standbys, one of which is the real leader. If command c is preserved across rotations, the new real leader will eventually commit c. If command c is not preserved across rotations, the client will eventually time out and retransmit c to the new real leader. Then the new real leader will commit command c.

In both *case 1* and *case 2*, we have shown that command c will eventually be committed. Now we show that command c will eventually be executed, which happens after all commands in the real log before c have been executed. The key to the proof is to show that all entries before c will eventually be committed.

For any non-empty log entry before c that has not been committed, by analysis above, they will be eventually committed. If there's any empty log entry before c, without loss of generality, denote the log index of the lowest empty entry as k, which is before c by definition. The real leader will commit a no-op in log entry k by the protocol. Therefore, all log entries before c, either empty or not, will eventually be committed. Then, the replica with c and all entries before c committed will execute commands in the log order, so it will eventually execute c and send a reply to the client.

$\therefore$ If a client sends a command, this command will eventually be executed and the client will receive a reply. □

B Additional Evaluation Results

This section shows the throughput-latency evaluation results with 3 and 5 replicas. As shown in Fig. 10 (a) and (b), when running only 1 shard, Avicenna shows competitive throughput with Multi-Paxos. When running multiple shards with

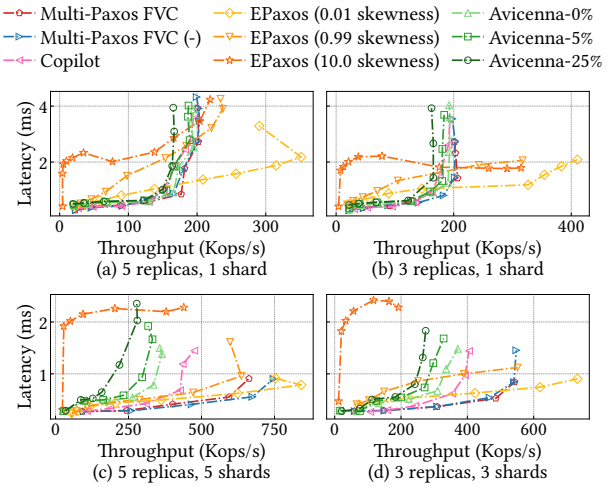

Figure 10. Throughput and latency in a single datacenter with 3 and 5 replicas.

small commands, as shown in Fig. 10 (c) and (d), Avicenna shows one third to half throughput compared to Multi-Paxos. We acknowledge that this is the major design trade-off of Avicenna.