HoliPaxos: Towards More Predictable Performance in State Machine Replication

Zhiying Liang
Pennsylvania State University
zvl5490@psu.edu

Vahab Jabrayilov
Columbia University
vj2267@columbia.edu

Abutalib Aghayev
Pennsylvania State University
agayev@gmail.com

Aleksey Charapko
University of New Hampshire
aleksey.charapko@unh.edu

ABSTRACT

State machine replication (SMR) algorithms ensure redundancy in critical systems and, as a result, underpin fault-tolerant distributed databases. Good SMR protocol performance is essential for capacity planning and meeting desired performance objectives. However, many implementations of popular SMR algorithms, such as MultiPaxos and Raft, have issues that make their performance unpredictable. This unpredictability often arises from certain "bolt-on" additions to core protocols, such as external failure detectors and replication log compaction. In this paper, we argue that tighter integration of such traditionally ad-hoc mechanisms with the core replication protocols can stabilize performance, making the solutions more reliable and more accessible to accurate capacity planning. Moreover, we show that these integrations can be non-disruptive for the underlying consensus algorithm, resulting in systems that preserve the simplicity and safety of traditional single-leader consensus-based SMR. To that order, we integrate the failure and slowdown detectors inside the SMR and achieve better performance and faster fail-over under various network partitions and node slowdown events. We also illustrate that tight integration of replication log management, pruning, and snapshotting can reduce memory and CPU usage while avoiding performance fluctuations associated with traditional log compaction and cleanup approaches.

PVLDB Reference Format:
Zhiying Liang, Vahab Jabrayilov, Abutalib Aghayev, and Aleksey Charapko. HoliPaxos: Towards More Predictable Performance in State Machine Replication. PVLDB, 18(8): 2505 - 2518, 2025.
doi:10.14778/3742728.3742744

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Zhiying12/holipaxos-artifect.

1 INTRODUCTION

State machine replication (SMR) is a cornerstone of modern fault-tolerant distributed data stores and databases [10, 11, 45, 46, 57, 58]. SMR algorithms provide strongly consistent replication by ensuring that every replica applies the same sequence of deterministic commands to its copy of a state machine. While many SMR protocols exist, MultiPaxos [49] and Raft [42], the two very similar protocols [20, 50], remain popular among major large-scale production systems such as Google Spanner [10], Azure Storage [6], Amazon DynamoDB [11], CockroachDB [45], and MongoDB [58].

The popularity of MultiPaxos and Raft in production is overwhelming despite extensive research into alternative approaches. Many SMR solutions emerge as optimizations to existing protocols and primarily focus on enhancing performance [8, 14, 27, 35, 36, 40, 44, 48]. Nevertheless, the relative simplicity of MultiPaxos and Raft compared to these other solutions, along with a better understanding of how these protocols behave in real systems, preserves the status quo and continues to separate production algorithms from their academic counterparts. Yet, it is unfair to say that Raft and MultiPaxos have no issues. Consider a 2020 CloudFlare outage [23, 31], which uncovered a liveness problem with the Raft protocol that prevents recovery under certain specific network partitions [19]. The academic community rushed to solve the problem, proposing solutions for Raft [23, 41] and MultiPaxos [37]. Both solutions are rather complicated and modify and extend the core protocols.

Some sources of friction for these battle-tested protocols come from the reliance on "ad-hoc" or "bolt-on" components for supplementary yet essential tasks, ranging from failure detection to slowdown detection to replication log management and cleanup. These ad-hoc components may introduce performance unpredictability and general instability into the larger system. In some cases, like the Raft failure, such unpredictability can lead to an outright failure. In others, it can complicate capacity planning and increase the system's operating cost by forcing overprovisioning to absorb the performance fluctuations. Performance unpredictability, especially with inadequate capacity allocation, can also lead to other severe performance problems, such as metastable failures [4, 21].

In this paper, we propose integrating these "bolt-on" components more fully with the core consensus protocol to improve the performance predictability of SMR systems and make them less susceptible to catastrophic failures. To that order, we develop HoliPaxos, a **holi**stic approach towards Multi**Paxos** protocol. HoliPaxos incorporates the failure detectors, slowdown detectors, and replication log management into the SMR without altering the underlying consensus protocol. Broadly, our contribution is a design paradigm of co-integration that allows for simpler individual solutions to combine and solve complicated problems. Specifically, our contributions are as follows: ① We integrate the Raft-style [42] failure

Proceedings of the VLDB Endowment, Vol. 18, No. 8 ISSN 2150-8097.
doi:10.14778/3742728.3742744

detector with additional optimizations to allow SMR to tolerate partial network partitions effectively. HoliPaxos handles network partitions as well as the state-of-the-art [37] while being faster in failure-free scenarios. ② We integrate a slowdown detection mechanism to enable leader change when the current leader experiences capacity degradation. HoliPaxos recovers the cluster from slowdown events in seconds while having up to 10× the throughput than Copilots Paxos, the current state-of-the-art slowdown tolerant SMR [38]. Our integrated slowdown detector relies on changepoint detection to self-diagnose a leader as potentially slow and remedy the situation by enacting a graceful leadership change. ③ Finally, we integrate log management and cleanup to avoid running and continuous costs of snapshotting. Our log cleanup forgoes periodic snapshotting in favor of continuous cleanup with on-demand snapshots for recovery. This reduced performance variance and improved memory usage by up to 50%.

These three contributions together make HoliPaxos a much more predictable protocol with fewer performance fluctuations in the failure-free operation and better handling of obscure failure situations, like partial network partitions and slowdowns and gray-failures [22]. Furthermore, our holistic approach is significantly simpler, and as a result faster, than the state-of-the-art new SMR protocols designed from the ground up to tolerate the same problems we address – Omni-Paxos [37] for partition tolerance and Copilots Paxos [38] for slowdown tolerance. We open-sourced the complete protocol specifications and code [29].

2 BACKGROUND & MOTIVATION

2.1 Single-Decree Paxos and MultiPaxos

Single-Decree Paxos [24, 25], a fundamental distributed consensus algorithm, relies on a group of nodes collectively agreeing on some non-trivial single value. It operates in two distinct phases: *Prepare* and *Accept*. In the *Prepare Phase*, a node prepares to become a leader with a unique ballot number higher than any it has encountered. If, upon receiving promises from a majority of nodes, the prospective leader concludes that the majority has not seen a higher ballot than its own, then it becomes a leader. Subsequently, the leader selects a value with the highest ballot among the received promises or a new value if no values are found in the promises. Moving forward, the leader advances to the *Accept Phase* where nodes accept the proposed value. The consensus is reached once the leader receives a majority of Accept Phase acknowledgments.

MultiPaxos [7, 49], a well-known practical optimization of Single-Decree Paxos, allows a leader to propose a sequence of values in *Accept Phase*, as shown in Figure 1. These values are proposed in a log structure, sometimes called *replication log*, where each log item corresponds to a command or a batch of commands. The log executes commands in an orderly manner against a state machine maintained by each replica, implementing a Replicated State Machine (RSM). The commands in the log exist in one of three states: *In-Progress, Committed, and Executed*. *In-Progress* commands are ones proposed by the leader but currently lacking the majority votes at the leader. Once a leader collects the majority votes for a command, it changes its state to *committed*. Only *committed* commands from the log can apply to the state machine. Once applied, they become *executed* and may eventually be cleaned up from the log. If the

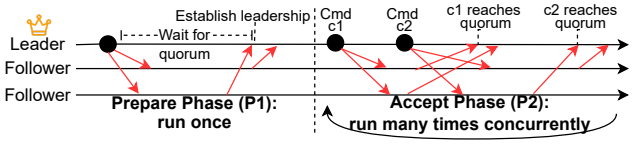


Figure 1: Overview of the MultiPaxos Algorithm.

next operation to be executed is not committed, the state machine stalls and waits for the commit decision before resuming execution. An implicit or explicit *Commit Phase* is needed to allow the leader to distribute the commit decision about each item to the followers. The *Prepare Phase* only comes into play in the event of a leader change due to a failure or leader placement optimizations [47, 53].

2.2 Performance Predictability

Performance predictability is an essential characteristic of distributed data-intensive systems. Systems with more predictable performance in various adverse conditions tend to be more reliable and less susceptible to certain failures, such as metastable failures [4, 21]. Often, performance predictability is more important to operators than cost and efficiency. Consider AWS DynamoDB, which experienced a large metastable failure early in its production history [11]. The cache malfunction sent too much traffic to the underlying metadata database, overloading it and preventing caches from refilling. The ultimate fix overprovisioned the metadata database so the system could survive a complete cache failure.

In the paper, we take a more narrow scope and discuss the performance predictability of leader-based state machine replication algorithms such as MultiPaxos, Viewstamped Replication [40] and Raft. These protocols often lie at the core of large distributed databases and storage solutions critical to all applications built on top, making their performance predictability under a wide range of operating conditions paramount to the reliability of an entire application stack. Nevertheless, as we show next, these traditional SMR solutions can still suffer performance predictability issues, often due to the "bolt-on" components required for practical operations.

2.2.1 Failure Detectors & Non-crash Failures. The node-local failure detector is often one of these "bolt-on" components, allowing each node to independently detect a problem with the leader and initiate a leader change, potentially triggering a leader-churn — a direct outcome of FLP impossibility [13]. The simplest churn happens when all correct nodes detect the failure and simultaneously try to become new leaders. This condition is not too dangerous and often leads to several rounds of leader election until one node is "lucky enough" to fully complete the election cycle. The typical mitigation strategy for this problem is random backoff intervals to minimize the chance of multiple nodes running a leader election at once [42]. Other solutions may rely on probabilistic consensus to bypass FLP and remove the destructive interference [47].

Unfortunately, ad-hoc-style failure detectors may not have access to protocol and cluster information useful for recovering from more intricate failures, such as partial network partitions [37] or node slowdowns [38]. Some protocols, such as Raft [42], boast tighter integration of failure detectors. Despite this, Raft limits its pool of potential leaders based on the candidates' up-to-dateness. In some

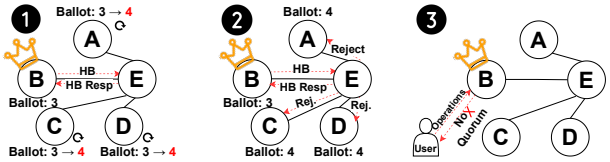


Figure 2: The Leader-Quorum-Loss partition. ① Nodes A, B, C, and D disconnect from all nodes except E. E still answers the heartbeat from Leader B. ② A, C, and D each run a leader election, but lack votes to succeed. ③ The leader B does not have a quorum for replication, while no node can replace it.

network partition cases, such extra selectivity prevents the failure detector from guiding the protocol to choosing the only node that could have worked as the leader if it was sufficiently up-to-date.

Partial Network Partitions. A partial network partition is a scenario where at least one node loses connection to some other node, yet both remain connected by a third node [2]. This situation poses a risk of livelocks in SMR systems, including MultiPaxos and Raft, with a notable example being the Cloudflare outage [31]. Below, we summarize two types of network partitions described in [37] that can cause performance problems due to the inadequacies in ad-hoc failure detectors or additional leadership requirements.

Leader-Quorum-Loss Partition. Figure 2 shows a partial network partition that can cause problems due to inadequate failure detectors. Starting with a fully connected five-node cluster with Node B as the leader, a partition isolates all nodes except Node E. Only Node E retains quorum, while Node B becomes a nonfunctional leader. Nodes A, C, and D start leader elections but fail to secure quorum. Prior studies note that MultiPaxos fails here as Node E continues to respond to Node B's heartbeats and would not start the leader election [37]. However, the root cause is the failure detector's lack of protocol knowledge to consider leader B nonfunctional.

Leader-Churning Partition. A specific partition can often cause local failure detectors to trigger leader churn, degrading performance. The churn occurs when ① churning nodes cannot communicate to each other, and ② they can initiate leader elections. Figure 3 illustrates such a case in a three-node fully connected cluster, initially led by Node A. When Nodes A and C become unreachable from each other, C starts an election and becomes the new leader. Later, Node A learns about C's new leadership via Node B and triggers another election due to missing heartbeats from C, resulting in cyclic leadership changes—a livelock situation. More complicated churn scenarios arise in larger clusters. However, the conditions for churn remain the same; as such, fixing churn during partitions requires disabling some nodes from initiating leader elections.

Leader Slowdowns. Traditional failure detectors used in SMR often can detect only severe problems, like nodes crashing or becoming disconnected. This is because these detectors rely on a simple binary timeout mechanism – a node either fails and cannot renew the lease or reply to a message in time or operates "sufficiently" well to respond to leases/timeouts. An intermittent but prolonged failure, such as excessive packet loss or resource starvation due to a noisy neighbor syndrome, may go unnoticed by such a binary failure detector. Such *gray failures* [22] remain a big problem for cloud-scale data-intensive systems [11, 32, 33].

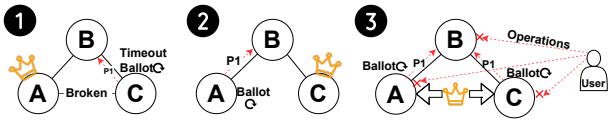


Figure 3: The Leader-Churning partition. Nodes A and C disconnect from each other. ① C times out and starts an election. B becomes a follower of C. ② A learns C's leadership via B and becomes a follower, but soon times out and starts another election. ③ A and C repeat leader elections, and no node can maintain the leadership.

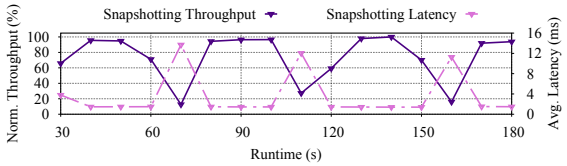


Figure 4: Fluctuating performance due to periodic snapshots.

Typically, detection of gray failures and slowdowns requires external tools [28, 33] as opposed to the built-in detectors. A notable exception is a Copilots Paxos [38] that relies on two co-leaders to process the same request and monitor each other for signs of a slowdown. The requirement for co-leaders substantially complicates the SMR protocol, as both leaders work independently and, when both are healthy, must reconcile their logs to arrive at a final order of requests. A slowdown is a huge problem if the leader node becomes slow, although recent research has shown that even a follower's slowdown may have negative performance implications [55].

2.2.2 Log Management & Performance Fluctuations. Log management is an essential bolt-on module in practical implementations. Applications replicate millions of log entries, and storing such large logs is a significant challenge. Academic studies often omit log management or propose impractical methods [1, 15, 27, 36]. In contrast, many industrial implementations rely on *compaction* – periodic snapshotting of the log and deletion of log entries captured in a snapshot. Snapshotting plays a crucial role in failure recovery. When a node joins the cluster or requires recovery, it can catch up without needing the complete log from other nodes by fetching the latest snapshot and the log of changes after the snapshot.

Periodic snapshotting requires exclusive access to the state machine (or some part of the state machine, depending on lock granularity), pausing the execution to ensure the snapshot consistently captures the current state and version. These periodic delays create short-lived availability pauses during the compaction, resulting in higher request latency or even timeouts. In Figure 4, we illustrate performance fluctuations of etcd [17] as it undergoes periodic compactions every 500,000 log entries. Furthermore, snapshotting requires excessive disk I/O and temporarily increases memory usage, leading to other issues such as higher garbage collection pressure.

3 HOLIPAXOS: PREDICTABLE MULTIPAXOS

To address performance predictability issues, we propose HoliPaxos, a holistic approach to SMR that integrates failure/slowdown

detection and replication log management into the core MultiPaxos-based [49] SMR. Like MultiPaxos, HoliPaxos requires a majority quorum and operates under the crash fault tolerance assumption. We also assume a semi-reliable network – a message from node n_1 to n_2 will eventually be delivered after an unspecified delay as long as n_1 is connected to n_2. In this section, we focus our narrative on changes and additions to MultiPaxos.

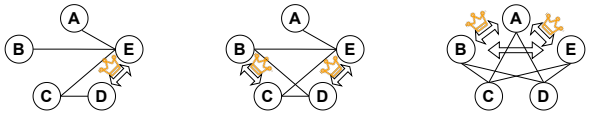
3.1 Integrated Failure Detector

Like traditional ad-hoc-style solutions, HoliPaxos uses independent failure detectors at every node, allowing any node to initiate a failover in case of a suspected leader malfunction. While the ability for a unilateral leader change is counterproductive for maintaining leader stability, it also avoids the explicit cross-node coordination to decide the need for a leader change. However, our failure detector relies on other protocol information, such as ballots and past election observations, to guide the election of a more stable node.

3.1.1 Heartbeats. The failure detector uses heartbeat messages to assess whether the component issuing heartbeats is alive. In our case, the leader node dispatches periodic heartbeats. The followers monitor the arrival of heartbeats to decide whether to trigger a leader failover, typically initiating a leader election after missing several heartbeats. The actual timeout (i.e., the number of missed heartbeats) slightly differs at each node to avoid nodes performing simultaneous leader elections. Unlike the ad-hoc failure detector implementations, e.g., FrankenPaxos [51], Omni-Paxos [37], each HoliPaxos heartbeat includes a ballot number, indicating the freshness of the leadership, allowing each node to compare the ballot on each heartbeat and identify the heartbeats issued by old leaders. Crucially, the nodes normally reply to any such stale leader with a heartbeat rejection message containing a newer leader's ballot, effectively forcing the old leader to step down to a follower role.

This Raft-borrowed technique rejects an old leader who lost a connection to a quorum and cannot make progress, a situation we described earlier in Figure 2. In this example, node B cannot stand down as it never hears from other nodes trying for leadership, and node E never attempts to acquire leadership, as it keeps hearing from B. However, E knows of higher ballots from other nodes trying to become leaders. Propagating such a higher ballot to B from E upon a heartbeat rejection forces B to stand down, unlocking node E to become a leader. Unlike HoliPaxos, Raft may not recover from this partition due to the additional leadership requirements despite having a similar failure detector. In Raft, if node E has a shorter log than the majority of other nodes, it will fail to acquire leadership due to rejections based on log length. HoliPaxos, however, follows the MultiPaxos leader election, allowing E to learn and recover missing items during the leader election.

3.1.2 Avoiding Churn. Independent, uncoordinated failure detectors are prone to leader churn when compromised nodes who do not see each other detect that as failures and initiate elections, as described in Figure 3. The only solution to the problem, aside from fixing the network partition, is to prevent all but one churning nodes from running the leader election. Omni-Paxos [37] does so using its knowledge of quorum-connected nodes and gossiping to decide which nodes are allowed to participate fully in the election.



(a) Quorum disconnected node churn. **(b) Quorum connected nodes churn** **(c) Churn between three nodes**

Figure 5: Advanced Leader-churning partitions.

Similarly, HoliPaxos needs to recognize the churn and temporarily prevent some nodes from becoming leaders.

We mitigate the churn with simple rules based on observations of churn in the system. Our rules rely on *common nodes* to which any churning node must be connected. Some, but not necessarily all, common nodes are the intersection between the quorums churning nodes form. In other words, there will be at least one common node in each quorum of a quorum-capable churning node. Common nodes allow each such churning node to learn about new leadership and new ballot and then trigger an election upon the timeout (since churning nodes do not see each other and do timeout). For instance, node B in Figure 3 is *common* to *churning nodes* A and C.

Rule 1: Pause leader elections for churning nodes. This first rule penalizes churning nodes by prolonging their heartbeat intervals upon detecting churn. These large heartbeat intervals stop all churning nodes from running leader elections, allowing other nodes to become leaders. This rule alone solves the leader churn in the most common [10, 11] SMR size of 3 nodes (Figure 3), where two leaders churn and prevent a centrally located common node from becoming a leader. To implement this rule, HoliPaxos nodes count the number of elections they lead within a sliding time window. For nodes that initiate an excessive number of elections within the window, their leader heartbeat interval is multiplied by a penalty factor, resulting in less frequent heartbeats from churning leaders. Since other non-churning nodes still expect heartbeats at regular, shorter intervals, they will eventually deem the leaders with high enough heartbeat intervals as failed, initiating the election. The penalty is dynamic and grows as nodes perform more elections within the window. Once nodes stop initiating excessive leader elections, they reset the counter of leader elections and heartbeat interval, resuming normal operation.

To put this rule in Figure 3, Nodes A and C repeatedly start elections for leadership. They increase their heartbeat intervals by observing excessive elections initiated by themselves within a certain timeframe. Eventually, a high heartbeat interval in A and C causes node B, with an unchanged timeout/heartbeat interval, to see A and C as failed and start a new election. As A and C are still responsive, they confirm B's leadership. Crucially, B's heartbeats cause A and C to stop their leadership attempts as both see an active leader. Once the leader churn stops long enough, all nodes reset their counters and heartbeat intervals to unpenalized values.

To solve the leader churn in larger clusters, like the second most popular deployment option of 5 nodes, we must introduce two additional rules. In larger clusters, a network partition may isolate nodes from a quorum of connections, yet still allow a leader to be elected. If these "quorum-disconnected" nodes do not see a leader, they may initiate new elections with higher ballots. Consequently, such ballot

will propagate to a leader via heartbeat rejections from nodes mutually connected to a leader and the "quorum-disconnected" node, as shown in Figure 5a. Note that the original MultiPaxos without the heartbeat rejection is somewhat immune to this problem, at least if the number of mutual nodes is small. An extension of this problem (Figure 5b and Figure 5c) arises when quorum-connected nodes churn while having multiple common nodes in their quorums.

Rule 2: Prevent quorum-disconnected nodes from leader elections. This rule prevents nodes that cannot reach a quorum from participating in leader election, removing them from the potential churn pool. HoliPaxos achieves this by counting the unique responses each node receives (both acks and rejections) when running the leader election (i.e., phase-1 of MultiPaxos). If a node does not receive at least a quorum of responses, it enters a "passive" mode. Such a node does not stop leader election attempts. However, it runs the elections with the invalid ballot 0 (e.g., a ballot of 0 in a system with a minimal valid ballot of 1). This ensures that the node will receive rejections from any node it can reach, not impacting the leader elections. At the same time, continuous attempts serve as a check for quorum-connectedness. Once a node can communicate with a quorum, it will resume normal elections with valid ballot numbers. This rule avoids problems similar to Figure 5a, where node D does not have a quorum but, if left active, will initiate elections and cause a quorum-connected node E to step down.

Rule 3: Use common nodes to break churn. Rule 1 can break the churn between two quorum-connected nodes when there exists just one common quorum-connected node connected to both churners. When such a node does not exist, we need an additional mechanism to break the churning pattern. Consider an example in Figure 5b. There, node B churns with node C, and nodes E and D are both connected to B and C. When we apply Rule 1 to B and C churn, either node D or E will become a leader. However, these nodes are not stable. For instance, if D becomes a leader, node E, which is disconnected from D, will not receive the heartbeats and will ultimately start the election, beginning a churning cycle between D and E. As D and E churn, they have two common nodes in their quorums – B and C, repeating the entire process.

These common nodes can detect churn and deterministically prevent some churning nodes from succeeding in their elections. The simplest way to block nodes is by having the common nodes reject them during the leader election without giving a valid ballot. HoliPaxos nodes detect churn by remembering past leader election attempts. After a sufficient churn history or schedule has accumulated, common nodes independently reject all but the highest ID nodes, ideally leaving only one leader candidate to succeed. Rejected nodes may continue attempting elections again without the ability to out-ballot other nodes, ensuring that their leader elections will fail and not interfere with another leader, directly or indirectly. A rejected node can still win the election if it somehow learns of the current ballot and increases it. This can happen when the node has gained additional connectivity to have learned the ballot, making its leadership attempt valid due to a new connectivity configuration.

Rule 3, running on common nodes, is mutually exclusive to Rule 1, which detects churn on churning nodes. If Rule 3 triggers, it will stop churn, essentially stopping the accumulation of observations for Rule 1. Similarly, triggering Rule 1 generally prevents Rule 3 from triggering, as Rule 1 stops or changes the churn pattern, resetting any churn observations in the common nodes. However, Rule 1 is the best option for solving leader churn in 3-node clusters. Rule 3 (combined with Rule 2) is a more general solution suitable for 5-node clusters, but it may not select a more centrally connected node as a leader. As such, Rule 1 has benefits worth considering in larger clusters. The two rules may be combined as follows – Rule 1 takes priority, allowing churning nodes to disqualify themselves and make any common quorum-connected nodes attempt the leadership. However, if such common nodes cannot hold on to the leader role (due to churn, as in Figure 5b), the system will eventually make a cycle and return to the original churning nodes. At this point, the original churning nodes will pause their execution of Rule 1, eventually allowing Rule 3 to take effect. This requires the time for churn detection in Rule 1 to be substantially slower than that for Rule 3. Pausing Rule 1 should not interfere with the ability to avoid churn later, as Rule 3 is generally sufficient, albeit sometimes suboptimal. However, we can still resume Rule 1 after some time.

Additionally, more than two nodes may churn concurrently, as shown in Figure 5c. However, this does not prevent the system from eventually settling on a stable leader. With more than two nodes churning, their churn schedule is more random, potentially causing common nodes to see only some nodes churning and decide what nodes to reject based on partial information. For example, in Figure 5c, nodes B and E can churn for some time, while node A has a slow failure detector. This delay may result in common nodes C and D rejecting B's attempts, allowing E to become a leader. Then A's failure detector kicks in, causing a new round of churn between A and E. However, Rule 3 will apply again, making E the leader.

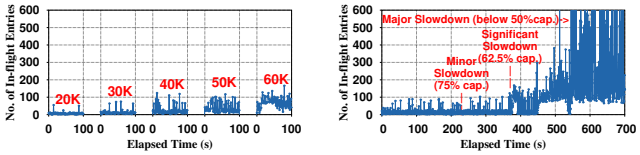
3.1.3 Correctness. Our heartbeat-based integrated failure detector only impacts when the leader election process begins and does not change the Paxos leader election algorithm, making it inherently safe. However, failure detection may affect liveness. Due to FLP [13], achieving both safety and liveness in consensus with asynchronous networks and the possibility of node failure is impossible. The liveness may be impacted by a deadlock – inability to elect a leader.

Theorem 1 (Failure Detector Liveness): *failure detector does not disqualify all eligible nodes from becoming leaders.*

Proof Sketch: Our detector avoids churn by altering the nodes' ability to run leader elections. Sometimes, we need to stop an eligible node from attempting to become a leader. However, we should never disqualify *all* eligible leaders. Rule 1 temporarily stops churning nodes from issuing heartbeats and running elections, allowing other nodes to run the election. This rule, as described, solves the only possible partition-induced leader churn scenario in 3-node clusters. In large clusters, Rule 1 is superseded by Rule 3. Rule 2 removes non-quorum-connected nodes from leadership attempts; these nodes could not have become leaders. Therefore, this rule does not disqualify any potential leader nodes.

As it stands, Rule 3 is the only rule that may impact liveness. Assume a Rule 3 can enter a deadlock state. This requires common nodes to reject every churning node. Assume we have a set C of n common nodes: $C = \{c1, ..., c_n\}$. We have two broad cases for how nodes in C with the churning nodes:

Case 1. The entire set C is in the leader election quorum of all churning nodes, ensuring that every node in C must answer every leader election. As a result, all nodes in C will observe the same churning



(a) The number of in-flight operations under increasing workloads from 20 K to 60 K ops/s.

(b) Gradual leader slowdown. The CPU capacity signifies the severity of a slowdown.

Figure 6: Number of in-flight operations.

schedule. Since our rule for selecting nodes is deterministic – keep the highest ID churning node eligible, nodes in C will all select the same node to remain eligible. This contradicts the hypothesis that common nodes reject every churning node.

Case 2. Only a subset of C is needed for each leader election quorum by churning nodes. As a result, not all nodes in C will make each leader election quorum, and some nodes may be slower to respond and not be a part of a quorum. However, by our networking assumption, all nodes in C will eventually receive all the messages as long as they remain connected to the senders of these messages. As a result, all nodes in C will eventually observe all leader election attempts and eventually construct the same churning schedule, making this case equivalent to Case 1 and contradicting the hypothesis of rejecting all churning nodes.

Finally, Rule 2 and Rule 3 allow nodes to be requalified upon changes in network connectivity, ensuring that the system does not get stuck in the older configuration when the network improves.

3.2 Integrated Slowdown Detection

In addition to a failure detector, HoliPaxos incorporates a slowdown detector to mitigate issues arising from a slow or underperforming leader. Like the current state-of-the-art [38], we target a case with at most one slow node – the leader (a quorum masks a slow follower node). Unlike the current state-of-the-art, we seek not to mask the faulty leader but instead focus on prompt leader change. At the high level, the integrated slowdown detector aims to identify *possible* leader slowdown and notify the followers to initiate a leader election via a standard leader change protocol. Our slowdown detector relies on self-monitoring, as a slow leader has not failed completely and may perform some limited self-diagnostics. Note that if the leader has failed or becomes too slow, our integrated failure detector on the follower side will catch it and initiate the leader election.

To detect potential leader slowdowns, HoliPaxos' leader monitors its outstanding, or "in-flight" operations—the operations received from clients but not yet applied to the state machine. The size of this in-flight queue is a proxy for tracking the leader's processing latency, which typically varies depending on the workload. These size changes are predictable, as workloads with more requests issued per second tend to have more outstanding operations in replication and awaiting execution. Figure 6a shows this observation as we subject HoliPaxos SMR to varying loads.

Whenever a leader becomes overloaded, as it would happen when a workload is too high for the leader's capacity due to a slowdown, the number of outstanding operations not only rises but also begins to fluctuate greatly, as illustrated in Figure 6b. In this figure, we gradually restrict the leader's CPU until it can no longer handle the workload, causing its outstanding queue to grow and fluctuate. Our slowdown detector targets this growth and fluctuation pattern, capturing the variance in the number of outstanding operations over a fixed time frame to gauge fluctuation intensity.

In particular, we use the Cumulative Sum (CUSUM) change point detection algorithm [43, 56] to detect the significant changes in the number of outstanding requests. Our CUSUM-based detector continuously monitors shifts in the variance of the number of outstanding operations. Every 0.5 seconds, it records the number of outstanding operations and calculates the variance of outstanding operations based on the previous 7.5 seconds of observations (i.e., 15 samples). We then compute low and high CUSUM on this variance (lines 8-10 in Algorithm 1). We use the variance of in-flight requests instead of the actual counts of outstanding operations to ensure that changes in workload (as in Figure 6b) do not automatically trigger a change point. We have tuned the critical level parameter ω to a low constant value based on our observations.

When the leader operates at full capacity, the variance in outstanding operations remains within a stable, low range, keeping the cumulative deviation close to its mean over time. However, once the variance shows a sustained fluctuation beyond normal levels, this deviation accumulates and crosses a threshold, triggering a change point. The CUSUM algorithm identifies two types of change points: increasing or decreasing. An increasing change point flags a potential slowdown event (lines 11-13 in Algorithm 1), while a decreasing change point signals potential machine recovery.

We also monitor the system's throughput to improve detection reliability and prevent false positives due to rapid workload changes. If the increasing change point in the number of in-flight requests corresponds to a substantial increase in throughput, we ignore this change point. Our evaluation confirms this simple solution avoids the slowdown detector triggering upon rapid workload increases.

When the slowdown detector suspects its node is slow, it will declare the node overloaded. The leader then replicates and commits the *overload declaration marker* to the log to ensure this declaration is persistent and cannot be forgotten by subsequent leaders. The overload declaration marker contains the leader's identity and the time of the overload event. The followers can initiate the leader change process upon receiving the overload declaration marker if the last successful leader (i.e., the leader who was able to replicate data) is the same one as specified in the marker. Like a normal leader change, more than one follower may start the process, but a random backoff process should reduce concurrent attempts.

Eventually, a new leader is elected, and the system either recovers its performance if the problem is indeed due to a slow leader or remains overloaded. Consequently, a new leader invalidates the overload declaration marker, stopping further leader change attempts in response to the overload event. If a leader election has remedied the problem, the new leader can write a corresponding *overload remediation marker*. This marker effectively signals that the overload was fixed via a leader change. Afterward, the current leader (or any subsequent leader) can issue a new overload marker onto the log. However, a leader (or any subsequent leader) cannot issue a new overload declaration marker until the previous one is

Algorithm 1 Slow leader self detector

Leader's Self Monitor Thread

```
1: metrics: The current evaluation window for storing recorded runtime metrics;
2: metrics_last: Runtime metrics from previous window;
3: variance_queue: The queue for storing variances;
4: C+, C−: Cumulative sum of positive and negative deviations, respectively.
5: while leader do
6:     metris = append(metrics, inflight_cout)
7:     if full(metrics) then
8:         Calculate variance of metrics and push it to variance_queue.
9:         dev. ← variance − mean_of_variance_queue.
10:        C+ ← Max(0, C+ + dev. − ω), C− ← Min(0, C− + dev. − ω).
11:        if C+ > Pos_Thres & TP(metrics) ≤ TP(metrics_last) then
12:            is_slow ← true.
13:            Reset C+, C− to 0.
14:        else if C− < Neg_Thres then
15:            is_slow ← false
16:            Reset C+, C− to 0.
17:        end if
18:        if is_slow == true & overload declaration has not replicated then
19:            replicate the declaration to trigger a leader change.
20:        end if
21:        metrics_last = metrics
22:        metrics ← nil
23:    end if
24: end while
```

remedied or a substantial timeout is elapsed. This mechanism ensures that HoliPaxos avoids repeated leader changes due to overload until the workload intensity subsides.

The resulting protocol achieves detection and failover from a configuration with one slow node; if more than one node is slow, the success is probabilistic and depends on whether a non-slow follower overtakes a slow leader. We list the high-level algorithm for the slowdown detection in Algorithm 1.

3.2.1 Correctness. Similar to the failure detector, we maintain safety by relying on an unmodified MultiPaxos leader election process. The liveness may be compromised if the slowdown detector continuously forces the leaders to change or churn. This can happen in case of a general overload when a workload exceeds the healthy capacity of nodes in the cluster. The overload declaration marker serves to prevent the leader churn. While multiple followers may compete for leadership, this is no different than a typical leader election, and eventually (possibly over a few rounds), a successful leader will emerge. When a successful leader emerges, the followers lose the mandate to initiate a new election based on the old overload marker, essentially ensuring that there is only one leader election cycle (possibly with many rounds) per overload marker.

3.3 Integrated Adaptive Log Management

SMR solutions, such as HoliPaxos, use a replication log to order commands for execution. The log also acts as a source of data for certain recovery operations. For example, when a node has missed a command, let's say due to a dropped message, it can recover the missing command from the log of another node. Log management, however, goes beyond appending commands to the log. Without log cleanup, the system will eventually expand all resources needed to maintain the log (i.e., memory or storage) and crash. Traditional log cleanup and compaction rely on periodic snapshots to truncate logs. However, such snapshotting is not necessary in SMR.

During failure-free operation, HoliPaxos eliminates the need for snapshotting and performs cleaning continuously while retaining only the log entries needed for a potential recovery. To that order, the leader periodically collects the log status from all nodes and determines a safe log index for removal. Followers respond to the leader's request with their local log indexes of the latest executed entries (i.e., the last log instance applied to the local copy of a state machine), called Last Executed (*LE*). Since a state machine executes sequentially, an *LE* index tells that a particular node has applied all operations up to that instance and no longer needs them locally. Upon receiving responses from all nodes, the leader computes the global minimum of *LE* across all nodes, namely the Global Last Executed (*GLE*), and includes it in subsequent requests. This enables all nodes to safely trim their logs up to this *GLE* value. Accordingly, all nodes respond with their latest *LE* values, and the leader updates *GLE* for the next cleanup round.

For example, as shown in Figure 7, the leader calculates a *GLE* of 101 based on the Last Executed values across nodes. When a follower receives a message with *GLE* of 101, it trims all prior entries up to and including index 101. Even if a different node becomes a leader, it won't need any entries before index 102.

The log cleanup up to the *GLE* ensures that all nodes have applied each log entry to the state machine and no longer need it. This approach lets HoliPaxos tolerate occasional issues, like missing log entries due to message loss or transient node slowdowns. If a node is missing a log item, the *GLE* stalls, ensuring others have that item in their log for recovery. Unfortunately, this creates a problem with more extended failures, as node failure will freeze *GLE*, causing unbounded log growth. In this case, we defer to the reconfiguration option—an extended failure must be fixed by replacing a node. As such, HoliPaxos will reconfigure [26, 52] to exclude the failed node, resume cleanup, and reconfigure again to add a replacement. The reconfigurations must be announced before they happen, and these announcements must survive any potential failures.

In contrast to the periodic snapshots approach, node recovery or node addition in HoliPaxos will request an on-demand current snapshot from some other node (preferably not the leader to avoid overloading it). This snapshot is less stale than the periodic one, requiring fewer log items to catch up, but procuring the snapshot incurs costs during the recovery as opposed to normal operation. An important caveat in adding a replacement node is that while the node is recovering or building its state machine, the *GLE* must be frozen. This requirement is intuitive, as the recovering node is not fully caught up and should block any cleanup that impedes recovery. This *GLE* freeze upon node addition is possible because of reconfiguration announcements that are traditionally committed into the state machine's log. The recovering node initially has its *LE* set at the current cluster's *GLE*. Once that node has successfully installed the on-demand snapshot, its *LE* advances to the snapshot's version; as the node starts to consume the log after the snapshot, it can continue advancing its *LE*, allowing *GLE* to advance as well.

Another advantage of HoliPaxos log management and cleanup is that it avoids configuring the cleanup or compaction interval. Periodic compaction slows down the system, so doing it too often is costly, while infrequent compaction increases the storage/memory consumption and recovery time. HoliPaxos avoids this dilemma, as it automatically adjusts its cleanup speed based on performance and how stale or caught up the nodes are.

A naive alternative to waiting for all nodes to execute some log item is waiting for a majority quorum. In a *quorum trim* approach, nodes remove a log entry right after the execution since an executed

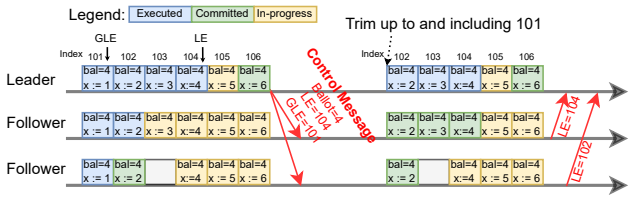


Figure 7: Control Phase. The life cycle of a log entry has three states in order: *In-Progress*, *Committed*, and *Executed*. On the left are logs of all nodes before the leader sends a control message with its *ballot*, *LE*, and *GLE*. On the right, logs change (log trimming and state changes) after the control message.

command must be in at least a quorum of nodes. This approach avoids the problem of stalled cleanup upon a single node failure and maintains a high cleanup rate. However, it also increases the possibility of on-demand snapshot requests for transient failures, as the quorum trim approach removes entries too quickly, leaving insufficient logs for any lagging followers to catch up.

3.3.1 Correctness. HoliPaxos log trim and cleanup have no impact on correctness and safety in a steady state, as all nodes remove the log entries only after the entire cluster has consumed the operations from the log and applied them to the state machine. We assume that nodes are either correct (in which case they participate in the protocol but may transiently go offline and then catch up) or recovering. Recovering nodes start with an empty state and an empty log. A node that has experienced permanent failure is removed from the cluster and replaced with a recovering node. Furthermore, we assume that only up to a minority of nodes can fail and consequently be replaced by recovering nodes.

Lemma 1: *The state machine of a correct node always includes the outcome of all cleaned-up log entries.*

Assume a set of $Nodes$ and node $r \in Nodes$; its state machine maintains some state S_r. As r executes a log instance i, its state transitions from S_{ri-1} to S_{ri}. After the execution, the instance i is no longer needed locally to node r since the outcome of i is preserved in S_{ri}. After executing i, r also advances its last executed frontier to $LE_r = i$. Since each replica r maintains its last executed (LE_r) frontier and the cleanup chooses the minimum LE from all nodes: $GLE = min(\{LE_0, LE_1, ..., LE_n\})$, for $n = Cardinality(Nodes)$, we ensure that $\forall r \in Nodes$: operation GLE is preserved in S_{rLE_r}. All instances up to GLE may be cleaned up, but their outcomes remain in the state machines of all correct nodes.

Theorem 2 (correctness): *The state machine of a recovering node is eventually caught up and identical to that of the correct nodes.*

Proof Sketch: Upon adding a new node z, the reconfiguration must be announced to at least the majority of the cluster. Initially, the new node z has its $LE_f = GLE$, blocking the GLE advancement upon z's announcement to the cluster. At the same time, node z requests an on-demand snapshot at version j from a correct node r. By Lemma 1, this snapshot S_{rj} must preserve all operations up to GLE: $j >= GLE$. After node z has installed the snapshot S_{rj}, it can start consuming log entries $(j, c]$, where c is the most recent log entry. Since these log entries are for instances after instance j and $j > GLE$, correct nodes must have these log instances available and not cleaned up. As a result, node z starts its recovery from a correct snapshot that captures the log up to instance j and is guaranteed to have access to log instances $(j, c]$ to eventually recover up to the most recent log entry and catch up with the rest of the cluster.

3.4 Integrated Control Messages

HoliPaxos relies on several control messages outside Paxos' normal phases to operate. The leader periodically broadcasts heartbeats and log status updates to all followers, and these control messages originating from the leader are frequent. Furthermore, the commit messages a leader sends to followers to tell follower nodes to advance their state machines can also be considered control messages, as they do not affect user-perceived performance. While having a distinct commit message for each committed log instance seems a straightforward option, it results in increased network traffic, especially when the throughput is high. Following our holistic, integrated approach, HoliPaxos combines all these types of periodic control information into a single control message from the leader.

The control message is lightweight and includes just two log indices to indicate the cluster's overall progress and a ballot. The leader includes its latest executed frontier LE in the message as a batch commit marker. Every log instance up to and including LE must have been a quorum committed for a leader to execute it, making it safe for the followers to change the states of all log instances up to and including LE to committed and eventually execute them as well. The leader also includes a GLE marker that indicates the global execution progress and, as discussed in subsection 3.3, allows safe log cleanup. Finally, the mere existence of the control message acts as a leader's heartbeat. The ballot number ensures that the heartbeat comes from a legitimate leader.

Figure 7 illustrates the control phase. The leader sends an integrated control message with $LE = 104$ and $GLE = 101$. Upon receiving it, followers validate the ballot to ensure the message is from the correct leader. Once the leader validation passes, the followers clean up all log instances up to and including instance 101. Followers also mark all instances up to LE (e.g., 104) as committed. If some instances are missing before the LE, such as at Follower 2, the follower pauses the local commit process. Although all existing entries between the gap and LE are safe to be committed (e.g., 104), they cannot be yet applied to the state machine. Thus, we leave these entries locally uncommitted until gaps are filled, and a future control message with a higher LE resumes the commit process.

4 EVALUATION

To assess the performance of HoliPaxos, we implemented it in Go as a modular and self-contained component providing interfaces to upper application layers. We then built a distributed linearizable in-memory Key-Value store on top. We used gRPC [16] with Protobuf [34] encoding for communication within the cluster. HoliPaxos also supports reconfiguration and snapshot creation.

In our evaluation of HoliPaxos, we assess whether our integrated components (failure and slowdown detectors, log management, and combined control messaging) improve performance stability and resource usage. We compare HoliPaxos against classical SMR solutions, such as MultiPaxos and Raft, and two state-of-the-art protocols: Omni-Paxos [37] and Copilots Paxos [38], which address

partial network partitions and leader slowdowns, respectively. We implemented MultiPaxos in Go, following the same design principles and standards as HoliPaxos. For Raft, we used etcd's Raft library [17]. It features a CheckQuorum mechanism to handle partition problems [41]. Since etcd includes many components not related to SMR directly, for a fair comparison, we used our key-value store with etcd's Raft, as described in the official example guidelines [12]. For Omni-Paxos and Copilots Paxos, we used their respective open-source implementations (Rust for Omni-Paxos and Go for Copilots Paxos). Omni-Paxos codebase requires a separate networking implementation; we used the networking implementation from Rust's version of our MultiPaxos code [30].

Omni-Paxos and Copilots Paxos are substantially more complicated SMR protocols than HoliPaxos. Omni-Paxos separates Paxos' leader into several stages and introduces a concept of quorum-connected servers (e.g., nodes know whether they could reach a quorum in the past) to facilitate leader election during network partitions by ensuring that only nodes that can reach a quorum attempt to become the next leaders. Omni-Paxos avoids interference from multiple quorum-connected nodes by preventing these nodes from continuously proposing with ever higher ballot numbers. This approach solves many possible leader election issues due to bad connectivity, albeit sometimes picking less than ideal nodes for a leader. Like HoliPaxos, it falls short of a more general solution to fixing partial connectivity with the help of message relaying [2].

Copilots Paxos deploys two leaders to handle the same commands from the clients; in case one leader becomes slow, the fast one can detect that and take over the ongoing work of a slow leader, thereby tolerating one slow machine. Having two leaders for the same operations requires the protocol to continuously resolve ordering ambiguities using an approach borrowed from EPaxos [36].

4.1 Experimental Setup

We ran all experiments on AWS m5.2xlarge virtual machines with 8 vCPUs and 32 GiB RAM located in the same subnet of a single region. For partition experiments, we present the results in 3 and 5 nodes, depending on the partition scenario. We focus on the 3-node cluster for all other experiments, as the larger cluster has similar trends across various experimental settings. An additional machine triggered partitioning, adding new nodes and disconnections. Node disconnections were simulated using the iptables command to control packet drops to/from specific IP addresses [18].

We used a YCSB [9] benchmark with a Zipfian request distribution with key and value sizes set at 23 B and 500 B, respectively [54]. The original Copilots Paxos supports only 8-byte keys and values, representing an unrealistic workload. We extended it to use YCSB with arbitrary key and object sizes. Before running the experiments, we populated the database with 1 million key-value pairs and ran a 10-second warm-up. We tried YCSB workload A (50% reads and 50% writes), workload B (95% reads and 5% writes), and workload C (100% reads) [5]. Given that the protocols exhibited similar performance patterns, we only present results for workload A.

4.2 Throughput vs. Latency

We first examine the general performance of HoliPaxos by comparing it to several established protocols, including Omni-Paxos, Copilots Paxos, and Paxi [1] from academia and production systems like etcd and TiKV. The primary objective of this evaluation is to demonstrate that HoliPaxos offers reasonable and competitive performance that is on par with these well-established systems. To ensure a fair comparison, we configured etcd and TiKV, which persist every entry in their logs to disk, to use a RAM disk [3].

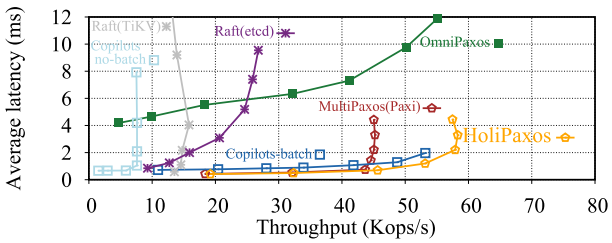


Figure 8: Throughput vs. Latency in different SMR.

The results in Figure 8 show throughput and average latency across runs with an increasing number of concurrent YCSB closed-loop clients, until throughput shows no significant further increase. The throughput of Copilots Paxos was severely limited on the client side, requiring us to use the largest available AWS instance. This limitation arises from more complicated client-side logic, as clients must send commands, and receive and interpret replies from both co-leaders. While etcd and TiKV exhibit similar throughput (15-25 Kops/s), HoliPaxos achieves nearly 60 Kops/s at comparable latency. This difference in performance is expected due to the additional features in both production systems, which may impact their throughput. HoliPaxos also outperforms Copilots Paxos and Paxi in maximum throughput while maintaining a similar or better average latency. Copilots Paxos performs extremely poorly without batching, reaching only 7 Kops/s in throughput. Despite not batching in HoliPaxos, we use batching with Copilots Paxos in the remaining experiments. Adding batching support to HoliPaxos is trivial and would substantially boost its throughput. Omni-Paxos has a similar maximum throughput but 4-6 times higher average latency than HoliPaxos, reflecting internal overhead due to the protocol's complexity. These failure-free results indicate that HoliPaxos generally outperforms the traditional and state-of-the-art protocols. Only one HoliPaxos contribution—better log management—helps its performance in a failure-free case. Both integrated failure and slowdown detectors do not contribute to better performance in this experiment nor harm it due to their lightweight nature.

4.3 Resilience Under Partial Partitions

Next, we analyze the performance under partial network partitions. Specifically, we used network partition scenarios described in section 3 and omitted the case of churning between three nodes due to its similar results to other examples. We initiated a 20-second network partition at the 80th second of a 180-second run. This duration is sufficient, as all protocols either regain availability within 20 seconds or remain unavailable indefinitely. HoliPaxos was tuned to count leader elections initiated by itself in a 3-second window. As the node sees 5 election attempts within the window, the heartbeat penalty of Rule 1 starts to increment, typically making the node churn for 1.5 seconds before its heartbeat interval is too high. Rule

(a) 5 nodes cluster with leader-quorum-loss partition. **(b) 3 nodes cluster with leader-churning partition.** **(c) 5 nodes churn with quorum-disconnected node (Figure 5a).** **(d) 5 nodes churn with quorum-connected nodes (Figure 5b).**

Figure 9: Normalized throughput under partial network partitions. Red dashed lines indicate the partition start and end. Raft+ is etcd with Pre-Vote and CheckQuorum. The dashed box highlights the performance drop after the network resumes.

3 window for collecting churn schedule is 5 seconds, giving priority to Rule 1. We use Omni-Paxos and two different versions of the Raft for comparison – the original one [42], and the one with *Pre-vote* and *CheckQuorum* features [41] to help with network partitions. Our experiments designate the latter Raft version as *Raft+*. Pre-vote requires a successful pre-election before starting the actual one, and CheckQuorum allows the leader to step down when it does not receive a quorum of heartbeat responses. Notably, with CheckQuorum enabled, etcd prevents followers from answering any votes if they still receive messages from the leader.

Given each implementation's varying maximum throughput, we normalize throughput by the protocol's respective maximums to emphasize the relative effect of partitions on performance. Figure 9 shows average normalized throughput over tumbling 10-second windows. HoliPaxos has relatively small performance degradation under these network partition cases and briefly loses up to 42% of its capacity. Omni-Paxos and both Raft variants experience comparable or worse degradation. MultiPaxos becomes unavailable.

Focusing on a Quorum-Loss Partition (Figure 9a), we see that HoliPaxos' stable node, having enough connections to form a quorum, quickly becomes a new leader. In our observations, it typically takes at least two rounds of election timeouts (one for other followers initiating the election and another for the stable node). Once the stable leader is elected, throughput rebounds to a nominal level. Omni-Paxos and Raft perform comparably to HoliPaxos in most runs. When the partition emerges, the stable node updates its ballot and disregards heartbeats from the old leader. Like HoliPaxos, no other node becomes the leader, and the stable node eventually initiates the election. However, in rare cases, Raft may become unavailable when the stable node lacks recent log entries and cannot become a leader. With CheckQuorum enabled, Raft's performance variance widens significantly as a result of multiple rounds of leader elections. As the existing leader rescinds the leadership actively due to no quorum, the stable node may immediately start the leader election and secure leadership within two timeouts, degrading throughput by as little as 20%. Conversely, the stable node may spend more time detecting the leader stepping down, causing its ballot number to lag behind other candidates. Multiple election rounds may occur, resulting in as much as 90% degradation. Compared to HoliPaxos, Raft+'s unavailability depends on how soon the old leader steps down and the stable node detects it, which varies during runtime.

Moving to the 3-node leader-churning partition (Figure 9b), HoliPaxos again shows greater resilience. It experiences partial unavailability during leadership churning but immediately resumes upon the election of a stable leader before the partition is resolved. HoliPaxos exhibits the best performance among the four evaluated protocols, with less than a 30% reduction in maximum throughput.

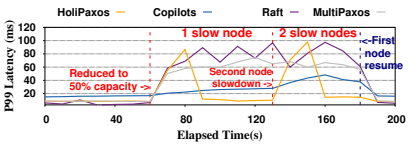
Omni-Paxos and both Raft versions perform differently in this scenario. Due to frequent leader election attempts, Raft's performance degradation varies widely, from 20% to 90%. The disconnected follower broadcasts a higher ballot but lacks the most recent log to become the leader. However, this higher ballot forces repeated elections. Unlike MultiPaxos, the stable node in Raft also participates in repeated elections and eventually wins, restoring availability. Raft+'s throughput remains unaffected during the partition, as the stable node ignores votes from the disconnected follower and avoids disruption. At the same time, OmniPaxos still suffers from degradation due to requiring at least one election. After the partition is resolved, the disconnected follower initiates elections with a higher ballot in both Omni-Paxos and Raft+, leading to several rounds of elections, causing a 30%-40% performance drop. At the same time, HoliPaxos elects a stable leader during the partition, which persists after the partition is fixed.

As for 5-node leader churn cases in Figure 9c and Figure 9d, all protocols experience some degradation. HoliPaxos quickly detects a quorum-disconnected node and prevents it from pulling a follower away from the new leader, allowing stability to resume swiftly in Figure 9c. However, it has a slightly higher performance loss with the quorum-connected nodes churning (Figure 9d), where HoliPaxos requires more time to block churner nodes. OmniPaxos behaves similarly to its performance in simpler 3-node configurations. Raft+ avoids the elections entirely when the original leader is one of the churners, keeping its performance degradation minimal. Basic Raft and MultiPaxos suffer the most because they cannot stop the leader churn. After both partitions are restored, the leader must help previously partitioned nodes catch up, impacting the performance of all protocols, including Raft+ and HoliPaxos.

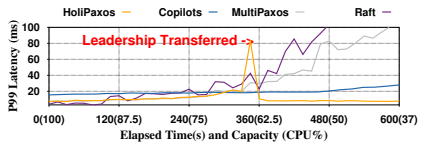
Under all partition scenarios, Omni-Paxos and Raft are comparable to HoliPaxos most of the time (except for occasional "hiccups"). However, Omni-Paxos and Raft have more complex designs that hurt steady-state performance.

4.4 Slowdown Detection Performance

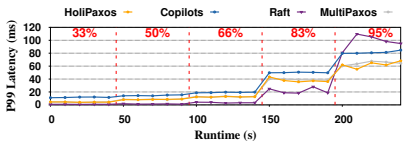
To validate the effectiveness of HoliPaxos's slowdown detection, we simulated a leader slowdown by reducing the CPU resources available to the leader's Docker container [39]. We used the same detector settings as outlined in subsection 3.2, detection threshold on in-flight request variance set to 3000 (# of ops)^2 and ω set to



(a) Rapid leader slowdown from 100% to 50% CPU. Red lines – CPU reduction on the leader; blue line – CPU capacity restoration.



(b) Gradual slowdown from 100% to 37% CPU. HoliPaxos's slowdown detector triggers when CPU capacity is reduced to 68.75%.



(c) Rapid workload increases every 50 seconds. HoliPaxos does not trigger slowdown detection.

Figure 10: 99th percentile latency with machine slowdown or an increasing workload.

100 (# of ops)^2. These values were set empirically based on the observations for our deployment. Throughout the experiment, we monitored the impact of slowdowns on the 99th percentile client-side latency. We used a workload with a target throughput of 60% of each protocol's maximum capacity. Figure 10 shows P99 latency over the 10-second tumbling window as the leader becomes slow.

In Figure 10a, we illustrate a significant rapid slowdown by halving the CPU capacity. MultiPaxos and Raft exhibit a sharp rise in tail latency immediately following the slowdown. HoliPaxos initially shows the same increase but manages to limit its duration and subsequently returns to normal latency after the slowdown detector switches the leader to a healthy node. When we cut half of the CPU capacity from the new leader, HoliPaxos re-elects another leader again, restoring stable performance. MultiPaxos and Raft only return to normal latency levels when the original leader is fully restored. Similar to HoliPaxos, Copilots Paxos demonstrates resilience with a single slow leader, showing a moderate latency increase. However, this latency increase persists for the entire duration of a slowdown event while HoliPaxos restores to nominal latency. When a second leader becomes slow, Copilots Paxos expectedly fails to prevent a significant rise in tail latency.

Next, we evaluate the gradual slowdown as we decrease CPU capacity by 6% every minute. Unlike sudden and significant slowdowns, these marginal reductions present a more challenging detection scenario. Figure 10b shows that tail latency slowly increases across all protocols as CPU capacity falls. When the CPU allocation drops below 70%, HoliPaxos's detector reaches its threshold and triggers a leader election to restore performance. Although the leadership transition incurs a temporary tail latency spike, the protocol gains long-term stability and reduced latency. This experiment highlights that our integrated self-slowdown-detection feature effectively captures the leader's creeping degradation. Unlike MultiPaxos and Raft, Copilots Paxos avoids latency spikes but still experiences a noticeable increase in tail latency when a machine becomes significantly slow. However, this stable latency increase comes at the cost of a complex protocol that requires twice the network traffic and duplicated processing at clients as they send, receive, and process every command twice, once for each leader.

Finally, in Figure 10c, we show that the HoliPaxos slowdown detector does not trigger upon sudden workload increases. We perform a sequence of rapid workload increases, starting at 30% of capacity and increasing it by 15% every 50 seconds. For instance, HoliPaxos can sustain roughly 60 Kops/s, making its initial starting workload of 18 Kops/s with a 9 Kops/s increase every 50 seconds. Such a workload change increases the in-progress queue size we use for slowdown detection. However, the system sees the corresponding increase in served requests and dismisses the change point event as a false positive. Once the workload reaches 75%, queuing effects become more pronounced, causing an increase in tail latency. If the slowdown detection was triggered, we would have seen a bigger initial spike of up to 100 ms for leader election.

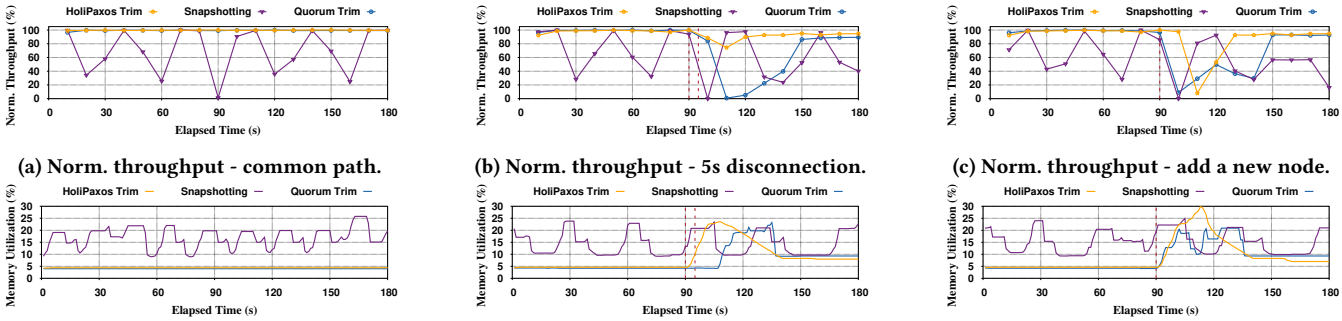
4.5 Log Management Performance

To evaluate the impact of log management on performance and resource utilization, we look at three scenarios: the common path, a short-lived temporary node disconnection (5 seconds), and the addition of a new node. We trigger configuration changes (adding or disconnecting nodes) at the 90-second mark in the figures, denoted by vertical red lines. We use periodic snapshotting from Raft and extend HoliPaxos to support the quorum-trim for comparison.

Figure 11 shows the throughput and memory utilization, measured for every 10-second tumbling window. Once again, we normalize throughput to the respective maximums of each implementation. Figure 11a reveals a remarkably stable performance of HoliPaxos integrated log management (HoliPaxos Trim) and the Quorum Trim for the common case without node failures or additions. In contrast, the throughput of periodic snapshotting exhibits a highly volatile pattern, with sharp peaks and valleys where degradation reaches as much as 98%. The variability underscores the considerable performance cost associated with snapshot-based log compaction. Additionally, as shown in Figure 11d, our approach has a low and stable memory usage due to the consistent pace of trimming the log entries. Conversely, snapshotting exhibits notable fluctuations in memory utilization. The peaks correspond to snapshot generation, during which a significant amount of memory is allocated before being persisted to a disk and reclaimed.

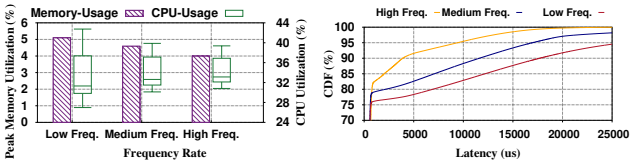
As for the node disconnection, the snapshotting behavior is similar to the steady operation, as shown in Figure 11b. This aligns with our expectations since the cost of snapshotting gets amortized throughout the runtime duration. Our log trimming mechanism is also affected, but any degradation is transient. The memory utilization spike arises from the suspension of log cleanup, jumping from 5% to nearly 25%, and this increase persists for over 5 seconds (the disconnection duration), as shown in Figure 11e. This is because the re-connected follower requires several rounds of Commit messages to catch up on the progress and complete the batch commit before the leader can update the *GLE* and start log trimming.

When adding nodes, the cost of taking a snapshot of the current state machine is visible for HoliPaxos and the Quorum Trip in Figure 11c and Figure 11f. However, Quorum Trim takes twice the



(a) Norm. throughput - common path.
(b) Norm. throughput - 5s disconnection.
(c) Norm. throughput - add a new node.
(d) Mem. Utilization - common path.
(e) Mem. utilization - 5s disconnection.
(f) Mem. utilization - add a new node.

Figure 11: The impact of different log cleanup methods. (a), (b), and (c) show normalized throughput. (d), (e), and (f) show the runtime memory utilization. The two red dashed lines in (b) and (e) refer to the disconnection for 5 seconds, and the red dashed line in (c) and (f) stands for the event of a new node joining. Both events start at the 90th second.



(a) The effect of trimming frequency on resource utilization.
(b) Request latency CDF for different control message frequencies.

Figure 12: The impact of log management frequency on performance. The log management frequency is adjusted via the control message intervals: *Low Freq.* (2400 ms), *Medium Freq.* (800 ms); *High Freq.* (200 ms). In (a), bars in box plots, from bottom to top, represent minimum, 25th percentile, median, 75th percentile, and maximum CPU utilization.

time as HoliPaxos to return to the nominal performance, as depicted in Figure 11c. This happens because bootstrapping a node may need more than one snapshot – when a new node restores from a snapshot, the leader may have already moved on and cleaned up the logs needed to catch up, requiring yet another snapshot and restore. The periodic snapshotting approach uses existing snapshots instead. Transmitting these existing snapshots still takes time and network bandwidth. Furthermore, the new node requires a longer catch-up period as it is restoring from an older state machine version compared to HoliPaxos. This, and the continuing cost of periodic snapshotting, results in choppier performance for a prolonged time.

4.5.1 Log Management Frequency. Next, we examine the overhead of HoliPaxos's log management, which operates based on integrated control messages. With each leader's control messages, followers commit log entries in batches and remove old log entries. We can change the frequency of log management activities by altering the control message interval. We try three different configurations—a large interval of 2400ms (low frequency), a medium one of 800ms (medium frequency), and a small one of 200ms (high frequency). As only followers perform batch commits and log trimming (the leader only trims the log), we present the results for followers.

Figure 12a shows the follower's peak memory utilization and the range of CPU utilization. Frequent log management results in lower peak memory usage as old log entries depart often, reclaiming memory. Frequent log cleanups also result in a smaller variance in CPU utilization. Although log management works more frequently, each cleanup is shorter, leading to a less bursty cleanup and more predictable CPU usage. In contrast, when the log management operates less frequently, it needs to handle more log entries at once, causing more fluctuation in CPU usage. Figure 12b further illustrates this overhead. The log management does not compete for resources and mutexes with replication requests, thereby keeping latency steady until the commit and trimming features are triggered. During the log trimming, however, lower-frequency log management has a larger backlog of accumulated log entries to process, impacting the request latency and driving the tail latency higher.

5 CONCLUSION

In this paper, we presented HoliPaxos, an enhanced version of the MultiPaxos consensus algorithm with a focus on integrated failure and slowdown detectors and efficient log management. Our failure detection addresses the limitations of previous MultiPaxos implementations that struggled under partial network partitions, achieving better resilience and performance predictability. Similarly, our self-monitoring slowdown detector allows a leader to declare itself slow and prompts other nodes to change leadership for better performance characteristics in slowdown cases. Our adaptive log management approach allows for efficient log trimming while minimizing the need for costly snapshotting, maintaining more stable performance, and less fluctuation in memory utilization. We further integrate all periodic metadata information into one control message to reduce the overhead of resource utilization. Overall, HoliPaxos offers a robust and stable solution for the real-world deployment of MultiPaxos, providing a more predictable runtime performance compared to traditional implementations.

ACKNOWLEDGMENTS

This project is in part sponsored by the National Science Foundation (NSF) under awards CNS-2149443, CNS-2149389, and CNS-2440896.

REFERENCES

[1] Ailidani Ailijiang, Aleksey Charapko, and Murat Demirbas. 2019. Dissecting the Performance of Strongly-Consistent Replication Protocols. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1696–1710. https://doi.org/10.1145/3299869.3319893

[2] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. https://www.usenix.org/conference/osdi20/presentation/alfatafta

[3] Kingston Authors. [n.d.]. What is a RAM Disk? https://www.kingston.com/en/blog/pc-performance/what-is-ram-disk. Last accessed on 05/10/2025.

[4] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 221–227.

[5] Sean Busbey. 2015. Core Workloads. https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. Last accessed on 05/10/2025.

[6] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 143–157.

[7] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos Made Live: An Engineering Perspective. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 398–407. https://doi.org/10.1145/1281100.1281103

[8] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. 2019. Linearizable quorum reads in Paxos. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC 2010)*. ACM, 143–154. https://doi.org/10.1145/1807128.1807152

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems (TOCS)* 31, 3, Article 8 (Aug. 2013), 22 pages. https://doi.org/10.1145/2491245

[11] Mostafa Elhemali, Niall Gallagher, Nick Gordon, Joseph Idziorek, Richard Krog, Colin Lazier, Erben Mo, Akhilesh Mritunjai, Somasundaram Perianayagam, Tim Rath, Swami Sivasubramanian, James Christopher Sorenson III, Sroaj Sosothikul, Doug Terry, and Akshat Vig. 2022. Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 1037–1048. https://www.usenix.org/conference/atc22/presentation/elhemali

[12] etcd-raft-example Authors. 2023. raftexample is an example usage of etcd's raft library. https://github.com/etcd-io/etcd/tree/main/contrib/raftexample. Last accessed on 05/10/2025.

[13] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)* 32, 2 (1985), 374–382.

[14] Pedro Fouto, Nuno Preguiça, and João Leitão. 2022. High Throughput Replication with Integrated Membership Management. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. 575–592.

[15] Aishwarya Ganesan, Ramnatthan Alagappan, Anthony Rebello, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2022. Exploiting Nil-external Interfaces for Fast Replicated Storage. *ACM Transactions on Storage (TOS)* 18, 3 (2022), 1–35.

[16] Google. 2023. gRPC: A high performance, open source universal RPC framework. https://grpc.io/.

[17] Red Hat. 2019. etcd. A distributed, reliable key-value store for the most critical data of a distributed system. https://coreos.com/etcd/. Last accessed on 05/10/2025.

[18] Ken Hess. 2020. Sysadmin tools: How to use iptables. https://www.redhat.com/sysadmin/iptables. Last accessed on 05/10/2025.

[19] Heidi Howard and Ittai Abraham. 2020. Raft does not Guarantee Liveness in the face of Network Faults. https://decentralizedthoughts.github.io/2020-12-12-raft-liveness-full-omission/.

[20] Heidi Howard and Richard Mortier. 2020. Paxos vs Raft: Have we reached consensus on distributed consensus?. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data*. 1–9.

[21] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 73–90.

[22] 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* (Whistler, BC, Canada) *(HotOS '17)*. Association for Computing Machinery, New York, NY, USA, 150–155. https://doi.org/10.1145/3102980.3103005

[23] Chris Jensen, Heidi Howard, and Richard Mortier. 2021. Examining Raft's behaviour during partial network failures. In *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems* (Online, United Kingdom) *(HAOC '21)*. Association for Computing Machinery, New York, NY, USA, 11–17. https://doi.org/10.1145/3447851.3458739

[24] Leslie Lamport. 1998. The Part-Time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (may 1998), 133–169. https://doi.org/10.1145/279227.279229

[25] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (December 2001), 51–58. https://www.microsoft.com/en-us/research/publication/paxos-made-simple/

[26] Leslie Lamport, Dahlia Malkhi, and Lidong Zhou. 2010. Reconfiguring a state machine. *SIGACT News* 41, 1 (mar 2010), 63–73. https://doi.org/10.1145/1753171.1753191

[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 2016)*. USENIX Association, 467–483.

[28] Xiaoyun Li, Pengfei Chen, Linxiao Jing, Zilong He, and Guangba Yu. 2020. Swisslog: Robust and unified deep learning based log anomaly detection for diverse faults. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 92–103.

[29] Zhiying Liang, Vahab Jabrayilov, Aleksey Charapko, and Abutalib Aghayev. [n.d.]. HoliPaxos Implementation. https://github.com/Zhiying12/holipaxos.. Last accessed on 05/19/2025.

[30] Zhiying Liang, Vahab Jabrayilov, Aleksey Charapko, and Abutalib Aghayev. 2024. MultiPaxos Made Complete. *arXiv preprint arXiv:2405.11183* (2024).

[31] Tom Lianza and Chris Snook. 2021. A Byzantine failure in the real world. https://blog.cloudflare.com/a-byzantine-failure-in-thereal-world/. Last accessed on 05/10/2025.

[32] Chang Lou, Peng Huang, and Scott Smith. 2020. Understanding, detecting and localizing partial failures in large system software. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 559–574.

[33] 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 *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 49–64.

[34] Protobuf Maintainers. 2023. Protocol Buffers Documentation. https://protobuf.dev/.

[35] Yanhua Mao, Flavio P. Junqueira, and Keith Marzullo. 2008. Mencius: Building Efficient Replicated State Machines for WANs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI 2008)*. USENIX Association, 369–384.

[36] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 358–372.

[37] Harald Ng, Seif Haridi, and Paris Carbone. 2023. Omni-Paxos: Breaking the Barriers of Partial Connectivity. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 314–330.

[38] Khiem Ngo, Siddhartha Sen, and Wyatt Lloyd. 2020. Tolerating Slowdowns in Replicated State Machines using Copilots. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 583–598. https://www.usenix.org/conference/osdi20/presentation/ngo

[39] Authors of Docker Resource constraints. [n.d.]. Resource constraints. https://docs.docker.com/engine/containers/resource_constraints/. Last accessed on 05/10/2025.

[40] 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 Seventh Annual ACM Symposium on Principles of Distributed Computing* (Toronto, Ontario, Canada) *(PODC '88)*. Association for Computing Machinery, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

[41] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph.D. Dissertation. Stanford, CA, USA. Advisor(s) K., Ousterhout, John and David, Mazières, and Mendel, Rosenblum,. AAI28121474.

[42] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC 2014)*. USENIX Association, 305–319.

[43] Ewan S Page. 1954. Continuous inspection schemes. *Biometrika* 41, 1/2 (1954), 100–115.

[44] Haochen Pan, Jesse Tuglu, Neo Zhou, Tianshu Wang, Yicheng Shen, Xiong Zheng, Joseph Tassarotti, Lewis Tseng, and Roberto Palmieri. 2021. Rabia: Simplifying

State-Machine Replication Through Randomization. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 472–487. https://doi.org/10.1145/3477132.3483582

[45] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD 2020)* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1493–1509. https://doi.org/10.1145/3318464.3386134

[46] YugabyteDB Team. 2021. YugabyteDB: cloud native distributed SQL database for mission-critical applications. https://www.yugabyte.com/yugabytedb/.

[47] 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*. 281–297.

[48] Sarah Tollman, Seo Jin Park, and John Ousterhout. 2021. {EPaxos} Revisited. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 613–632.

[49] Robbert Van Renesse and Deniz Altinbuken. 2015. Paxos Made Moderately Complex. *ACM Comput. Surv.* 47, 3, Article 42 (feb 2015), 36 pages. https://doi.org/10.1145/2673577

[50] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. 2019. On the Parallels between Paxos and Raft, and How to Port Optimizations. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 445–454. https://doi.org/10.1145/3293611.3331595

[51] Michael Whittaker. 2021. FrankenPaxos. https://github.com/mwhittaker/frankenpaxos. Last accessed on 05/10/2025.

[52] Michael Whittaker, Neil Giridharan, Adriana Szekeres, Joseph M Hellerstein, Heidi Howard, Faisal Nawab, and Ion Stoica. 2020. Matchmaker paxos: A reconfigurable consensus protocol [technical report]. *arXiv preprint arXiv:2007.09468* (2020).

[53] Andy Woods and Daniel Harrison. 2018. Geo-Partitioning: What Global Data Actually Looks Like. https://www.cockroachlabs.com/blog/geo-partitioning-one/.

[54] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 191–208. https://www.usenix.org/conference/osdi20/presentation/yang

[55] Andrew Yoo, Yuanli Wang, Ritesh Sinha, Shuai Mu, and Tianyin Xu. 2021. Fail-slow fault tolerance needs programming support. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 228–235.

[56] Xiaofeng Yu and Ying Cheng. 2022. A comprehensive review and comparison of CUSUM and change-point-analysis methods to detect test speededness. *Multivariate Behavioral Research* 57, 1 (2022), 112–133.

[57] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: high-availability storage made practical in WeChat. *Proceedings of the VLDB Endowment* 10, 12 (2017), 1730–1741.

[58] Siyuan Zhou and Shuai Mu. 2021. {Fault-Tolerant} Replication with {Pull-Based} Consensus in {MongoDB}. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. 687–703.