Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems

Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, ***University of Wisconsin–Madison***

https://www.usenix.org/conference/osdi18/presentation/alagappan

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-931971-47-8

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

Fault-Tolerance, Fast and Slow: Exploiting Failure Asynchrony in Distributed Systems

Ramnatthan Alagappan, Aishwarya Ganesan, Jing Liu,
Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau

University of Wisconsin – Madison

Abstract

We introduce *situation-aware updates and crash recovery* (SAUCR), a new approach to performing replicated data updates in a distributed system. SAUCR adapts the update protocol to the current situation: with many nodes up, SAUCR buffers updates in memory; when failures arise, SAUCR flushes updates to disk. This situation-awareness enables SAUCR to achieve high performance while offering strong durability and availability guarantees. We implement a prototype of SAUCR in ZooKeeper. Through rigorous crash testing, we demonstrate that SAUCR significantly improves durability and availability compared to systems that always write only to memory. We also show that SAUCR's reliability improvements come at little or no cost: SAUCR's overheads are within 0%-9% of a purely memory-based system.

1 Introduction

The correctness and performance of a fault-tolerant system depend, to a great extent, upon its underlying replication protocols. In the modern data center, these protocols include Paxos [45], Viewstamped Replication [48], Raft [57], and ZAB [39]. If these protocols behave incorrectly, reliability goals will not be met; if they perform poorly, excess resources and cost will be incurred.

A key point of differentiation among these protocols relates to how they store system state (§2). In one approach, which we call *disk durable*, critical state is replicated to persistent storage (i.e., hard drives or SSDs) within each node of the system [13, 14, 17, 37, 57]. In the contrasting *memory durable* approach, the state is replicated only to the (volatile) memory of each machine [48, 55]. Unfortunately, neither approach is ideal.

With the disk-durable approach, safety is paramount. When correctly implemented, by committing updates to disks within a majority of nodes, the disk-durable approach offers excellent durability and availability. Specifically, data will not be lost if the nodes crash and recover; further, the system will remain available if a bare majority of nodes are available. Unfortunately, the cost of safety is performance. When forcing updates to hard drives, disk-durable methods incur a 50× overhead; even when using flash-based SSDs, the cost is high (roughly 2.5×).

With the memory-durable approach, in contrast, performance is generally high, but at a cost: durability. In the presence of crash scenarios where a majority of nodes crash (and then recover), existing approaches can lead to data loss or indefinite unavailability.

The distributed system developer is thus confronted with a vexing quandary: choose safety and pay a high performance cost, or choose performance and face a potential durability problem. A significant number of systems [17, 41, 48, 55, 61] lean towards performance, employing memory-durable approaches and thus risking data loss or unavailability. Even when using a system built in the disk-durable manner, performance concerns can entice the unwary system administrator towards disaster; for instance, the normally reliable disk-durable ZooKeeper can be configured to run in a memory-durable mode [5], leading (regrettably) to data loss [30].

In this paper, we address this problem by introducing *situation-aware updates and crash recovery* or SAUCR (§3), a hybrid replication protocol that aims to provide the high performance of memory-durable techniques while offering strong guarantees similar to disk-durable approaches. The key idea underlying SAUCR is that the mode of replication should depend upon the situation the distributed system is in at a given time. In the common case, with many (or all) nodes up and running, SAUCR runs in memory-durable mode, thus achieving excellent throughput and low latency; when nodes crash or become partitioned, SAUCR transitions to disk-durable operation, thus ensuring safety at a lower performance level.

SAUCR applies several techniques to achieve high performance and safety. For example, a *mode-switch technique* enables SAUCR to transition between the fast memory-durable and the safe disk-durable modes. Next, given that SAUCR can operate in two modes, a node recovering from a crash performs *mode-aware crash recovery*; the node recovers the data from either its local disk or other nodes depending on its pre-crash mode. Finally, to enable a node to safely recover from a fast-mode crash, the other nodes store enough information about the node's state within them in the form of *replicated last-logged entry (LLE) maps*.

The effectiveness of SAUCR depends upon the simultaneity of failures. Specifically, if a window of time ex-

ists between individual node failures, the system can detect and thus react to failures as they occur. SAUCR takes advantage of this window in order to move from its fast mode to its slow-and-safe mode.

With *independent* failures, such a time gap between failures exists because the likelihood of many nodes failing together is negligible. Unfortunately, failures can often be *correlated* as well, and in that case, many nodes can fail together [31, 35, 42, 64]. Although many nodes fail together, a correlated failure does not necessarily mean that the nodes fail at the same instant: the nodes can fail either non-simultaneously or simultaneously. With non-simultaneous correlated failures, a time gap (ranging from a few milliseconds to a few seconds) exists between the individual failures; such a gap allows SAUCR to react to failures as they occur. With simultaneous failures, in contrast, such a window does not exist. However, we conjecture that such truly simultaneous failures are extremely rare; we call this the Non-Simultaneity Conjecture (NSC). While we cannot definitively be assured of the veracity of NSC, our analysis (§2.3) of existing data [31, 33] hints at its likely truth.

Compared to memory-durable systems, SAUCR improves reliability under many failure scenarios. Under independent and non-simultaneous correlated failures, SAUCR always preserves durability and availability, offering the same guarantees as a disk-durable system; in contrast, memory-durable systems can lead to data loss or unavailability. Additionally, if NSC holds, SAUCR always provides the same guarantees as a disk-durable system. Finally, when NSC does not hold and if more than a majority of nodes crash in a truly simultaneous fashion, SAUCR remains unavailable, but preserves safety.

We implement (§4) and evaluate (§5) a prototype of SAUCR in ZooKeeper [4]. Through rigorous fault injection, we demonstrate that SAUCR remains durable and available in hundreds of crash scenarios, showing its robustness. This same test framework, when applied to existing memory-durable protocols, finds numerous cases that lead to data loss or unavailability. SAUCR's reliability improvements come at little or no performance cost: SAUCR's overheads are within 0%-9% of memory-durable ZooKeeper across six different YCSB workloads. Compared to the disk-durable ZooKeeper, with a slight reduction in availability in rare cases, SAUCR improves performance by 25× to 100× on HDDs and 2.5× on SSDs.

2 Distributed Updates and Recovery

In this section, we first describe the disk-durable and memory-durable protocols. We then describe the characteristics of different kinds of failures. Finally, we draw attention to the non-reactiveness to failures and the static nature of existing protocols.

	Mode	Avg. Latency (μs)	Throughput (ops/s)
HDD	`fsync`-s disabled	327.86	3050.1
cluster1	disk durability	16665.18 (50.8× ↑)	60.0 (50.8× ↓)
SSD	`fsync`-s disabled	461.2	2168.34
cluster2	disk durability	1027.3 (2.3× ↑)	973.4 (2.3× ↓)

Table 1: **Disk Durability Overheads.** *The table shows the overheads of disk durability. The experimental setup is detailed in §5.2.*

2.1 Disk-Durable Protocols

Disk-durable protocols always update the disk on a certain number of nodes upon every data modification. For example, ZooKeeper [4], etcd [24], and other systems [14, 49, 53, 62] persist every update on a majority of nodes before acknowledging clients.

With the exception of subtle bugs [2], disk-durable protocols offer excellent durability and availability. Specifically, committed data will never be lost under any crash failures. Further, as long as a majority of nodes are functional, the system will remain available. Unfortunately, such strong durability and availability guarantees come at a cost: poor performance.

Disk-durable protocols operate with caution and pessimistically flush updates to the disk (e.g., by invoking the `fsync` system call [11, 60]). Such forced writes in the critical path are expensive, often prohibitively so. To highlight these overheads, we conduct a simple experiment with ZooKeeper in the following modes: first, in the disk-durable configuration in which the `fsync` calls are enabled; second, with `fsync` calls disabled. A client sends update requests in a closed loop to the leader which then forwards the requests to the followers. We run the experiment on a five-node cluster and thus at least three servers must persist the data before acknowledgment.

As shown in Table 1, on HDDs, forced writes incur a 50× performance overhead compared to the `fsync`-disabled mode. Even on SSDs, the cost of forced writes is high (2.3×). While batching across many clients may alleviate some overheads, disk-durable protocols are fundamentally limited by the cost of forced writes and thus suffer from high latencies and low throughput.

A disk-durable update protocol is usually accompanied by a disk-based recovery protocol. During crash-recovery, a node can immediately join the cluster just after it recovers the data from its disk. A recovering node can completely trust its disk because the node would not have acknowledged any external entity before persisting the data. However, the node may be lagging: it may not contain some data that other nodes might have stored after it crashed. Even in such cases, the node can immediately join the cluster; if the node runs for an election, the leader-election protocol will preclude this node from becoming the leader because it has not stored some data that the other nodes have [2, 57]. If a leader already exists, the node fetches the missed updates from the leader.

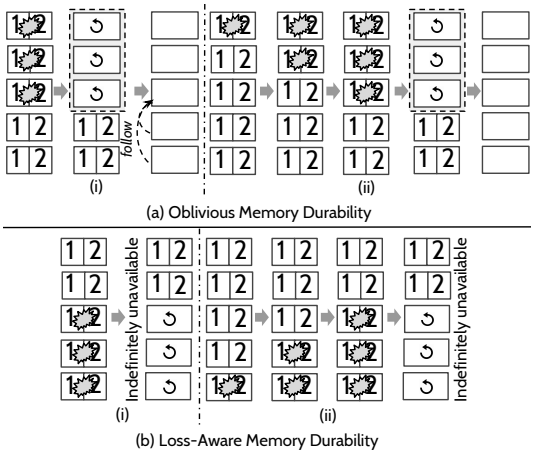


Figure 1: **Problems in Memory-Durable Approaches.** *(a) and (b) show how a data loss or an unavailability can occur with oblivious and loss-aware memory durability, respectively. In (i), the nodes fail simultaneously; in (ii), they fail non-simultaneously, one after the other.*

2.2 Memory-Durable Protocols

Given the high overheads imposed by a disk-durable protocol, researchers and practitioners alike have proposed alternative protocols [17, 55], in which the data is always buffered in memory, achieving good performance. We call such protocols *memory-durable* protocols.

2.2.1 Oblivious Memory Durability

The easiest way to achieve memory "durability" is *oblivious memory durability*, in which any forced writes in the protocol are simply disabled, unaware of the risks of only buffering the data in memory. Most systems provide such a configuration option [8, 22, 27, 62]; for example, in ZooKeeper, turning off the *forceSync* flag disables all `fsync` calls [5]. Turning off forced writes increases performance significantly, which has tempted practitioners to do so in many real-world deployments [29, 38, 59].

Unfortunately, disabling forced writes might lead to a data loss [5, 43] or sometimes even an unexpected data corruption [68]. Developers and practitioners have reported several instances where this unsafe practice has led to disastrous data-loss events in the real world [7, 30].

Consider the scenarios shown in Figure 1(a), in which ZooKeeper runs with *forceSync* disabled. If a majority of nodes crash and recover, data could be silently lost. Specifically, the nodes that crash could form a majority and elect a leader among themselves after recovery; however, this majority of nodes have lost their volatile state and thus do not know of the previously committed data, causing a silent data loss. The intact copies of data on other nodes (servers 4 and 5) can be overwritten by the new leader because the followers always follow the leader's state in ZooKeeper [2, 57].

2.2.2 Loss-Aware Memory Durability

Given that naïvely disabling forced writes may lead to a silent data loss, researchers have examined more careful approaches. In these approaches, a node, after a crash and a subsequent reboot, realizes that it might have lost its data; thus, a recovering node first runs a distinct recovery protocol. We call such approaches *loss-aware memory-durable* approaches.

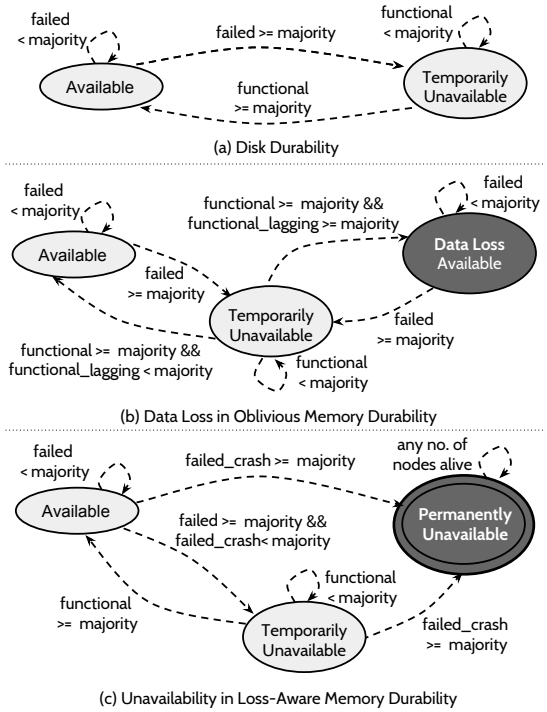


Figure 2: **Summary of Protocol Behaviors and Guarantees.** *The figure shows how the disk-durable and memory-durable protocols behave under failures and the guarantees they provide.*

The view-stamped replication (VR) protocol [55] is an example of this approach. Similarly, researchers at Google observed that they could optimize their Paxos-based system [17] by removing disk flushes, given that the nodes run a recovery protocol. For simplicity, we use only VR as an example for further discussion.

In VR, when a node recovers from a crash, it first marks itself to be in a *recovering* state, in which the node can neither participate in replication nor give votes to elect a new leader (i.e., a view change) [48]. Then, the node sends a recovery message to other servers. A node can respond to this message if it is *not* in the *recovering* state; the responding node sends its data to the recovering node. Once the node collects responses from a majority of servers (including the leader of the latest view), it can fix its data. By running a recovery protocol, this approach prevents a silent data loss.

Unfortunately, the loss-aware approach can lead to unavailability in many failure scenarios. Such an unavailability event could be *permanent*: the system may remain unavailable indefinitely even after all nodes have recovered from failures. For example, in Figure 1(b), a majority of nodes crash and recover. However, after re-

covering from the crash, none of the nodes will be able to collect recovery responses from a majority of nodes (because nodes in the *recovering* state cannot respond to the recovery messages), leading to permanent unavailability.

Protocols Summary. Figure 2 summarizes the behaviors of the disk-durable and memory-durable protocols. A node either could be functional or could have failed (crashed or partitioned). Disk-durable protocols remain available as long as a majority are functional. The system becomes *temporarily* unavailable if a majority fail; however, it becomes available once a majority recover. Further, the protocol is durable at all times.

The oblivious memory-durable protocol becomes temporarily unavailable if a majority fail. After recovering from a failure, a node could be lagging: it either recovers from a crash, losing all its data, or it recovers from a partition failure, and so it may not have seen updates. If such functional but lagging nodes form a majority, the system can silently lose data.

The loss-aware memory-durable approach becomes temporarily unavailable if the system is unable to form a majority due to partitions. However, the system becomes permanently unavailable if a majority or more nodes crash at any point; the system cannot recover from such a state, regardless of how many nodes recover.

2.3 Failures and Failure Asynchrony

Given that existing approaches compromise on either performance or reliability, our goal is to design a distributed update protocol that delivers high performance while providing strong guarantees. Such a design needs a careful understanding of how failures occur in datacenter distributed systems, which we discuss next.

Similar to most distributed systems, our goal is to tolerate only fail-recover failures [34, 36, 45, 57] and not Byzantine failures [16, 46]. In the fail-recover model, nodes may fail any time and recover later. For instance, a node may crash due to a power loss and recover when the power is restored. When a node recovers, it loses all its volatile state and is left only with its on-disk data. We assume that persistent storage will be accessible after recovering from the crash and that it will not be corrupted [32]. In addition to crashing, sometimes, a node could be partitioned and may later be able to communicate with the other nodes; however, during such partition failures, the node does not lose its volatile state.

Sometimes, node failures are *independent*. For example, in large deployments, single-node failure events are often independent: a crash of one node (e.g., due to a power failure) does not affect some other node. It is unlikely for many such independent failures to occur together, especially given the use of strategies such as failure-domain-aware placement [3, 44, 50].

With independent failures, the likelihood that a majority of nodes fail together is negligible. Under such a condition, designing a protocol that provides both high performance and strong guarantees is fairly straightforward: the protocol can simply buffer updates in memory always. Given that a majority will not be down at any point, the system will always remain available. Further, at least one node in the alive majority will contain all the committed data, preventing a data loss.

Unfortunately, in reality, such a failure-independence assumption is rarely justified. In many deployments, failures can be correlated [12, 20, 25, 35, 64, 66]. During such correlated crashes, several nodes fail together, often due to the same underlying cause such as rolling reboots [31], bad updates [54], bad inputs [26], or data-center-wide power outages [42].

Given that failures can be correlated, it is likely that the above naïve protocol may lose data or become unavailable. An ideal protocol must provide good performance and strong guarantees in the presence of correlated failures. However, designing such a protocol is challenging. At a high level, if the updates are buffered in memory (aiming to achieve good performance), a correlated failure may take down all the nodes together, causing the nodes to lose the data, affecting durability.

Although many or all nodes fail together, a correlated failure does not mean that the nodes fail at the same instant; the nodes can fail either non-simultaneously or simultaneously. With non-simultaneous correlated crashes, a time gap between the individual node failures exists. For instance, a popular correlated-crash scenario arises due to bad inputs: many nodes process a bad input and crash together [26]. However, such a bad input is not applied at exactly the same time on all the nodes (for instance, a leader applies an input before its followers), causing the individual failures to be non-simultaneous.

In contrast, with simultaneous correlated crashes, such a window between failures does not exist; all nodes may fail before any node can detect a failure and react to it. However, we conjecture that such truly simultaneous crashes are extremely rare; we call this the Non-Simultaneity Conjecture (NSC). Publicly available data supports NSC. For example, a study of failures in Google data centers [31] showed that in most correlated failures, nodes fail one after the other, usually a few seconds apart.

We also analyze the time gap between failures in the publicly available Google cluster data set [33]. This data set contains traces of machine events (such as the times of node failures and restarts) of about 12K machines over 29 days and contains about 10K failure events. From the traces, we randomly pick five machines (without considering failure domains) and examine the timestamps of their failures. We repeat this 1M times (choosing different sets of machines). We find that the time between two failures among the picked machines is greater than 50

ms in 99.9999% of the cases. However, we believe the above percentage is a conservative estimate, given that we did not pick the machines across failure domains; doing so is likely to increase the time between machine failures. Thus, we observe that truly simultaneous machine failures are rare: a gap of 50 ms or more almost always exists between the individual failures.

Given that in most (if not all) failure scenarios, a window of time exists between the individual failures, a system can take advantage of the window to react and perform a preventive measure (e.g., flushing to disk). A system that exploits this asynchrony in failures can improve durability and availability significantly.

2.4 Non-Reactiveness and Static Nature

We observe that existing update protocols do not react to failures. While it may be difficult to react to truly simultaneous failures, with independent and non-simultaneous failures, an opportunity exists to detect failures and perform a corrective step. However, existing protocols do not react to *any* failure.

For example, the oblivious memory-durable protocol can lose data, regardless of the simultaneity of the failures. Specifically, a data loss occurs both in Figure 1(a)(i) in which the nodes crash simultaneously and (a)(ii) in which they fail non-simultaneously. Similarly, the loss-aware approach can become unavailable, regardless of the simultaneity of the failures (as shown in Figure 1(b)). This is the reason we do not differentiate simultaneous and non-simultaneous failures in Figure 2; the protocols behave the same under both failures.

Next, we note that the protocols are *static* in nature: they always update and recover in a constant way, regardless of the situation; this situation-obliviousness is the cause for poor performance or reliability. For example, the disk-durable protocol constantly anticipates failures, forcing writes to disk even when nodes never or rarely crash; this unnecessary pessimism leads to poor performance. In contrast, when nodes rarely crash, a situation-aware approach would buffer updates in memory, achieving high performance. Similarly, the memory-durable protocol always optimistically buffers updates in memory even when only a bare majority are currently functional; this unwarranted optimism results in poor durability or availability. In contrast, when only a bare majority are alive, a situation-aware approach would safely flush updates to disk, improving durability and availability.

Our approach, *situation-aware updates and crash recovery* or SAUCR, reacts to failures quickly with corrective measures, and adapts to the current situation of the system. Such reactiveness and situation-awareness enables SAUCR to achieve high performance similar to a memory-durable protocol while providing strong guarantees similar to a disk-durable protocol.

3 Situation-Aware Updates and Recovery

The main idea in SAUCR is that of situation-aware operation, in which the system operates in two modes: *fast* and *slow*. In the common case, with many or all nodes up, SAUCR operates in the fast mode, buffering updates in memory and thus achieving high performance. When failures arise, SAUCR quickly detects them and performs two corrective measures. First, the nodes flush their data to disk, preventing an imminent data loss or unavailability. Second, SAUCR commits subsequent updates in slow mode, in which the nodes synchronously write to disk, sacrificing performance to improve reliability.

When a node recovers from a crash, it performs mode-aware recovery. The node recovers its data either from its local disk or from other nodes depending on whether it operated in slow or fast mode before it crashed.

We first outline SAUCR's guarantees (§3.1) and provide an overview of SAUCR's modes (§3.2). Next, we discuss how SAUCR detects and reacts to failures (§3.3), and describe the mechanisms that enable mode-aware recovery (§3.4). We then explain how crash recovery works and show its safety (§3.5). Finally, we summarize SAUCR's key aspects and describe the guarantees in detail (§3.6).

3.1 Guarantees

We consider three kinds of failures: independent, correlated non-simultaneous, and correlated simultaneous failures. SAUCR can tolerate any number of independent and non-simultaneous crashes; under such failures, SAUCR always guarantees durability. As long as a majority of servers eventually recover, SAUCR guarantees availability. Under simultaneous correlated failures, if a majority or fewer nodes crash, and if eventually a majority recover, SAUCR will provide durability and availability. However, if *more than* a majority crash simultaneously, then SAUCR cannot guarantee durability and so will remain unavailable. However, we believe such truly simultaneous crashes are extremely rare. We discuss the guarantees in more detail later (§3.6).

3.2 SAUCR Modes Overview

We first describe some properties common to many majority-based systems. We then highlight how SAUCR differs from existing systems in key aspects.

Most majority-based systems are *leader-based* [6, 57]; the clients send updates to the leader which then forwards them to the followers. The updates are first stored in a log and are later applied to an application-specific data structure. A leader is associated with an *epoch*: a slice of time; for any given epoch, there could be at most one leader [6, 57]. Because only the leader proposes an update, each update is uniquely qualified by the epoch in which the leader proposed it and the index of the update in the log. The leader periodically checks if a follower

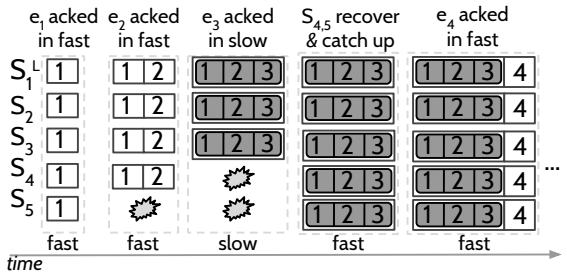


Figure 3: **Saucr Modes.** *The figure shows how* SAUCR*'s modes work. S_1 is the leader. Entries in a white box are committed but are only buffered (e.g., e_1 and e_2 in the first and second states). Entries shown grey denote that they are persisted (e.g., $e_1 - e_3$ in the third state). In fast mode, a node loses its data upon a crash and is annotated with a crash symbol (e.g., S_5 has lost its data in the second state).*

is alive or not via heartbeats. If the followers suspect that the leader has failed, they compete to become the new leader in a new epoch. Most systems guarantee the *leader-completeness* property: a candidate can become the leader only if it has stored all items that have been acknowledged as committed [2, 57]. SAUCR retains all the above properties of majority-based systems.

In a memory-durable system, the nodes always buffer updates in memory; similarly, the updates are always synchronously persisted in a disk-durable system. SAUCR changes this fundamental attribute: SAUCR either buffers the updates or synchronously flushes them to disk depending on the situation. When more nodes than a bare minimum to complete an update are functional, losing those additional nodes will not result in an immediate data loss or unavailability; in such situations, SAUCR operates in fast mode. Specifically, SAUCR operates in fast mode if *more than* a bare majority are functional (i.e., functional $\geq \lceil n/2 \rceil + 1$, where n is the total nodes, typically a small odd number). If nodes fail and only a bare majority ($\lceil n/2 \rceil$) are functional, losing even one additional node may lead to a data loss or unavailability; in such situations, SAUCR switches to the slow mode. Because the leader continually learns about the status of the followers, the leader determines the mode in which a particular request must be committed.

We use Figure 3 to give an intuition about how SAUCR's modes work. At first, all the nodes are functional and hence the leader S_1 replicates entry e_1 in fast mode. The followers acknowledge e_1 before persisting it (before invoking `fsync`); similarly, the leader also only buffers e_1 in memory. In fast mode, the leader acknowledges an update only after $\lceil n/2 \rceil + 1$ nodes have buffered the update. Because at least four nodes have buffered e_1, the leader acknowledges e_1 as committed. Now, S_5 crashes; the leader detects this but remains in fast mode and commits e_2 in fast mode.

Next, S_4 also crashes, leaving behind a bare majority; the leader now immediately initiates a switch to slow mode and replicates all subsequent entries in slow mode. Thus, e_3 is replicated in slow mode. Committing an entry in slow mode requires at least a bare majority to persist the entry to their disks; hence, when e_3 is persisted on three nodes, it is committed. Further, the first entry persisted in slow mode also persists all previous entries buffered in memory; thus, when e_3 commits, e_1 and e_2 are also persisted. Meanwhile, S_4 and S_5 recover and catch up with other nodes; therefore, the leader switches back to fast mode, commits e_4 in fast mode, and continues to commit entries in fast mode until further failures.

3.3 Failure Reaction

In the common case, with all or many nodes alive, SAUCR operates in fast mode. When failures arise, the system needs to detect them and switch to slow mode or flush to disk. The basic mechanism SAUCR uses to detect failures is that of heartbeats.

Follower Failures and Mode Switches. If a follower fails, the leader detects it via missing heartbeats. If the leader suspects that only a bare majority (including self) are functional, it immediately initiates a switch to slow mode. The leader sends a special request (or uses an outstanding request such as e_3 in the above example) on which it sets a flag to indicate to the followers that they must respond only after persisting the request; this also ensures that all previously buffered data will be persisted. All subsequent requests are then replicated in slow mode. When in fast mode, the nodes periodically flush their buffers to disk in the background, without impacting the client-perceived performance. These background flushes reduce the amount of data that needs to be written when switching to slow mode. Once enough followers recover, the leader switches back to fast mode. To avoid fluctuations, the leader switches to fast mode after confirming a handful number of times that it promptly gets a response from more than a bare majority; however, a transition to slow mode is immediate: the first time the leader suspects that only a bare majority of nodes are alive.

Leader Failures and Flushes. The leader takes care of switching between modes. However, the leader itself may fail at any time. The followers quickly detect a failed leader (via heartbeats) and flush all their buffered data to disk. Again, the periodic background flushes reduce the amount of data that needs to be written.

3.4 Enabling Safe Mode-Aware Recovery

When a node recovers from a crash, it may have lost some data if it had operated in fast mode; in this case, the node needs to recover its lost data from other nodes. In contrast, the node would have all the data it had logged on its disk if it had crashed in slow mode or if it had flushed after detecting a failure; in such cases, it recovers the data only from its disk. Therefore, a recovering node first needs to determine the mode in which it last

operated. Moreover, if a node recovers from a fast-mode crash, the other nodes should maintain enough information about the recovering node. We now explain how SAUCR satisfies these two requirements.

3.4.1 Persistent Mode Markers

The SAUCR nodes determine their pre-crash mode as follows. When a node processes the first entry in fast mode, it synchronously persists the epoch-index pair of that entry to a structure called the *fast-switch-entry*. Note that this happens only for the first entry in the fast mode. In the slow mode or when flushing on failures, in addition to persisting the entries, the nodes also persist the epoch-index pair of the latest entry to a structure called the *latest-on-disk-entry*. To determine its pre-crash mode, a recovering node compares the above two on-disk structures. If its *fast-switch-entry* is ahead[1] of its *latest-on-disk-entry*, then the node concludes that it was in the fast mode. Conversely, if the *fast-switch-entry* is behind the *latest-on-disk-entry*, then the node concludes that it was in the slow mode or it had safely flushed to disk.

3.4.2 Replicated LLE Maps

Once a node recovers from a crash, it must know how many entries it had logged in memory or disk before it crashed. We refer to this value as the *last logged entry* or LLE of that node. The LLE-recovery step is crucial because only if a node knows its LLE, it can participate in elections. Specifically, a candidate requests votes from other nodes by sending its LLE. A participant grants its vote to a candidate if the participant's LLE and current epoch are not ahead of the candidate's LLE and current epoch, respectively [57] (provided the participant had not already voted for another candidate in this epoch).

In a majority-based system, as long as a majority of nodes are alive, the system must be able to elect a leader and make progress [10, 57]. It is possible that the system only has a bare majority of nodes including the currently recovering node. Hence, it is crucial for a recovering node to immediately recover its LLE; if it does not, it cannot participate in an election or give its vote to other candidates, rendering the system unavailable.

If a node recovers from a slow-mode crash, it can recover its LLE from its disk. However, if a node recovers from a fast-mode crash, it would *not* have its LLE on its disk; in this case, it has to recover its LLE from other nodes. To enable such a recovery, as part of the replication request, the leader sends a map of the last (potentially) logged entry of each node to every node. The leader constructs the map as follows: when replicating an entry at index i in epoch e, the leader sets the LLE of all the functional followers and self to e.i and retains the last successful value of LLE for the crashed or partitioned followers. For instance, if the leader (say, S_1) is replicating an entry at index 10 in epoch e to S_2, S_3, and S_4, and if S_5 has crashed after request 5, then the map will be $\langle S_1$:e.10, S_2:e.10, S_3:e.10, S_4:e.10, S_5:e.5$\rangle$. We call this map the *last-logged entry map* or LLE-MAP. In the fast mode, the nodes maintain the LLE-MAP in memory; in slow mode, the nodes persist the LLE-MAP to the disk.

[1] An entry a is ahead of another entry b if (a.epoch $>$ b.epoch) or (a.epoch $==$ b.epoch and a.index $>$ b.index).

3.5 Crash Recovery

In a disk-durable system, a node recovering from a crash performs three distinct recovery steps. First, it recovers its LLE from its disk. Second, it competes in an election with the recovered LLE. The node may either become the leader or a follower depending on its LLE's value. Third, the node recovers any missed updates from other nodes. If the node becomes the leader after the second step, it is guaranteed to have all the committed data because of the leader-completeness property [2, 57], skipping the third step. If the node becomes a follower, it might be lagging and so fetches the missed updates from the leader.

In SAUCR, a node recovering from a crash could have operated either in slow or fast mode before it crashed. If the node was in slow mode, then its recovery steps are identical to the disk-durable recovery described above; we thus do not discuss slow-mode crash recovery any further. A fast-mode crash recovery, however, is more involved. First, the recovering node would not have its LLE on its disk; it has to carefully recover its LLE from the replicated LLE-MAPs on other nodes. Second, it has to recover its lost data irrespective of whether it becomes the leader or a follower. We explain how a node performs the above crash-recovery steps.

Max-Among-Minority. A SAUCR node recovering from a fast-mode crash recovers its LLE using a procedure that we call *max-among-minority*. In this procedure, the node first marks itself to be in a state called *recovering* and then sends an LLE query to all other nodes. A node may respond to this query only if it is in a recovered (*not recovering*) state; if it is not, it simply ignores the query. Note that a node can be in the recovered state in two ways. First, it could have operated in fast mode and not crashed yet; second, it could have last operated in slow mode and so has the LLE-MAP on its disk. The recovering node waits to get responses from at least a bare minority of nodes, where $bare\text{-}minority = \lceil n/2 \rceil - 1$; once the node receives a bare-minority responses, it picks the maximum among the responses as its LLE. Finally, the node recovers the actual data up to the recovered LLE. For now, we assume that at least a bare minority will be in recovered state; we soon discuss cases where only fewer than a bare minority are in the recovered state (§3.6).

We argue that the max-among-minority procedure guarantees safety, i.e., it does not cause a data loss. To do so, let us consider a node N that is recovering from a

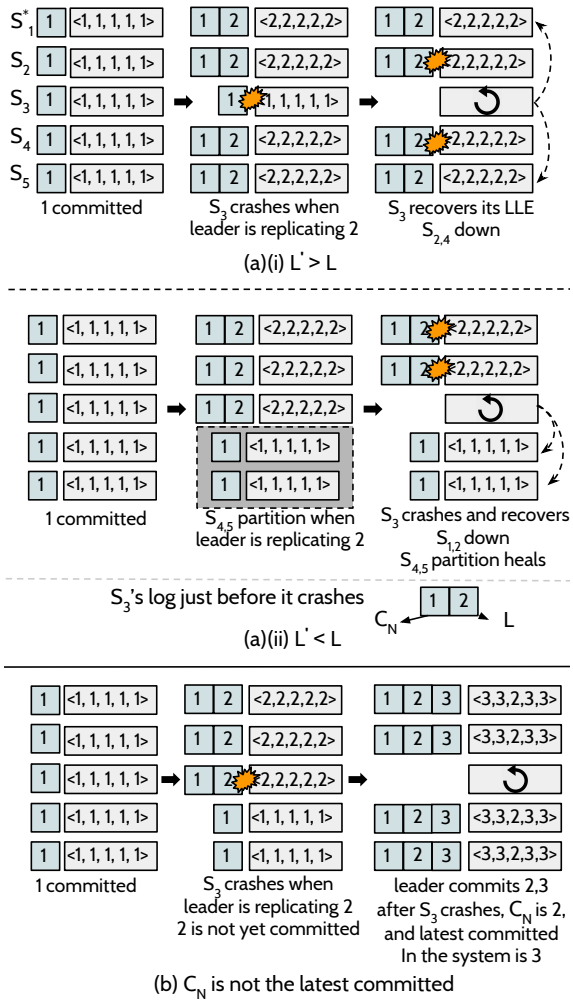


Figure 4: **LLE Recovery.** *The figure shows how L′ may not be equal to L. For each node, we show its log and LLE-MAP. The leader (S_1) is marked *; crashed nodes are annotated with a crash symbol; nodes partitioned are shown in a dotted box; epochs are not shown.*

fast-mode crash and let its actual last-logged entry (LLE) be L. When N runs the max-among-minority procedure, it retrieves L′ as its LLE and recovers all entries up to L′.

If N recovers exactly all entries that it logged before crashing (i.e., L′=L), then it is as though N had all the entries on its local disk (similar to how a node would recover in a disk-durable protocol, which is safe). Therefore, if the retrieved L′ is equal to the actual last-logged entry L, the system would be safe.

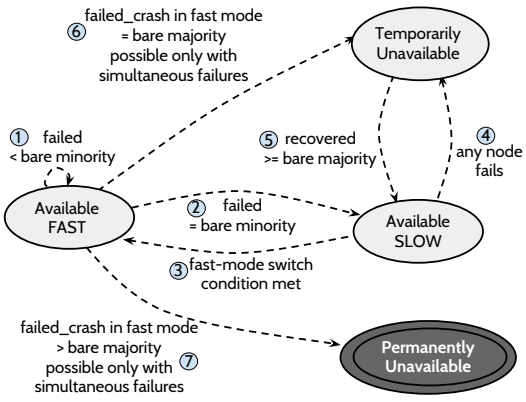
However, in reality, it may not be possible for N to retrieve an L′ that is exactly L. If N crashes after the leader sends a replication request but before N receives it, N may retrieve an L′ that is greater than L. For example, consider the case shown in Figure 4(a)(i). The leader (S_1) has successfully committed entry-1 in fast mode and now intends to replicate entry-2; hence, the leader populates the LLE-MAP with 2 as the value for all the nodes. However, S_3 crashes before it receives entry-2; consequently, its LLE is 1 when it crashed. However, when S_3 recovers its LLE from LLE-MAPs of S_1 and S_5 using the max-among-minority algorithm, the recovered L′ will be 2 which is greater than 1. Note that if L′ is greater than L, it means that N will recover additional entries that were not present in its log, which is safe. Similarly, it is possible for N to retrieve an L′ that is smaller than L. For instance, in Figure 4(a)(ii), S_3 has actually logged two entries; however, when it recovers, its L′ will be 1 which is smaller than the actual LLE 2. $L' < L$ is the only case that needs careful handling.

We now show that the system is safe even when the recovered L′ is smaller than L. We first establish a lower bound for L′ that guarantees safety. Then, we show that max-among-minority ensures that the recovered L′ is at least as high as the established lower bound.

Lower bound for L′. Let N's log when it crashed be D and let C_N be the last entry in D that is committed. For example, in Figure 4(a)(ii), for S_3, D contains entries 1 and 2, and the last entry in D that was committed is 1. Note that C_N need not be the latest committed entry; the system might have committed more entries after N crashed but none of these entries will be present in N's log. For example, in Figure 4(b), for S_3, C_N is 2 while the latest committed entry in the system is 3.

For the system to be safe, all *committed* entries must be recovered, while the *uncommitted* entries need *not* be recovered. For example, in Figure 4(a)(ii), it is safe if S_3 does not recover entry-2 because entry-2 is uncommitted. However, it is unsafe if N does not recover entry-1 because entry-1 is committed. For instance, imagine that S_3 runs an incorrect recovery algorithm that does not recover entry-1 in Figure 4(a)(ii). Now, if S_1 and S_2 also run the incorrect algorithm, then it is possible for S_1, S_2, and S_3 to form a majority and lose committed entry-1. Therefore, if the recovery ensures that N recovers all the entries up to C_N, committed data will not be lost, i.e., L′ must be at least as high as the last entry in N's log that is committed. In short, the lower bound for L′ is C_N. Next, we show that indeed the L′ recovered by max-among-minority is equal to or greater than C_N.

Proof Sketch for $L' \geq C_N$. We prove by contradiction. Consider a node N that is recovering from a fast-mode crash and that C_N is the last entry in N's log that was committed. During recovery, N queries a bare minority. Let us suppose that N recovers an L′ that is less than C_N. This condition can arise if a bare minority of nodes hold an LLE of N in their LLE-MAPs that is less than C_N. This is possible if the bare minority crashed long ago and recently recovered, or they were partitioned. However, if a bare minority had crashed or partitioned, it is *not* possible for the remaining bare majority to have committed C_N in fast mode (recall that a fast-mode commitment requires at least $\textit{bare-majority} + 1$ nodes to have bufferred



$bare\ majority = \lceil n/2 \rceil, bare\ minority = \lceil n/2 \rceil - 1$

Figure 5: **SAUCR Summary and Guarantees.** *The figure summarizes how* SAUCR *works under failures and the guarantees it provides.*

C_N and updated their LLE-MAPs). Therefore, C_N could have either been committed only in slow mode or not committed at all. However, if C_N was committed in slow mode, then N would be recovering from a slow-mode crash which contradicts the fact that N is recovering from a fast-mode crash. The other possibility that C_N could not have been committed at all directly contradicts the fact that C_N is committed. Therefore, our supposition that L' is less than C_N must be false.

Once a node has recovered its LLE, it can participate in elections. If an already recovered node or a node that has not failed so far becomes the leader (for example, S_1 or S_5 in Figure 4(a)(i)), it will already have the LLE-MAP, which it can use in subsequent replication requests. On the other hand, if a recently recovered node becomes the leader (for example, S_3 in Figure 4(a)(i)), then it needs to construct the LLE-MAP values for other nodes. To enable this construction, during an election, the voting nodes send their LLE-MAP to the candidate as part of the vote responses. Using these responses, the candidate constructs the LLE-MAP value for each node by picking the maximum LLE of that node from a bare-minority responses.

Data recovery. Once a node has successfully recovered its LLE, it needs to recover the actual data. If the recovering node becomes the follower, it simply fetches the latest data from the leader. In contrast, if the recovering node becomes the leader, it recovers the data up to the recovered LLE from the followers.

3.6 Summary and Guarantees

We use Figure 5 to summarize how SAUCR works and the guarantees it offers; a node fails either by crashing or by becoming unreachable over the network. We guide the reader through the description by following the sequence numbers shown in the figure. ① At first, we assume all nodes are in recovered state; in this state, SAUCR operates in the fast mode; when nodes fail, SAUCR stays in the fast mode as long as the number of nodes failed is less than a bare minority. ② After a bare minority of nodes fail, SAUCR switches to slow mode. ③ Once in slow mode, if one or more nodes recover and respond promptly for a few requests, SAUCR transitions back to fast mode. ④ In slow mode, if any node fails, SAUCR becomes temporarily unavailable. ⑤ Once a majority of nodes recover, the system becomes available again.

To explain further transitions, we differentiate non-simultaneous and simultaneous crashes and network partitions. In the presence of non-simultaneous crashes, nodes will have enough time to detect failures; the leader can detect follower crashes and switch to slow mode and followers can detect the leader's crash and flush to disk. Thus, despite any number of non-simultaneous crashes, SAUCR always transitions through slow mode. Once in slow mode, the system provides strong guarantees.

However, in the presence of simultaneous crashes, many nodes could crash instantaneously while in fast mode; in such a scenario, SAUCR cannot always transition through slow mode. ⑥ If the number of nodes that crash in fast mode does not exceed a majority, SAUCR will only be temporarily unavailable; in this case, at least a bare minority will be in recovered state or will have previously crashed in slow mode making crash recovery possible (as described in §3.5). ⑦ In rare cases, more than a bare majority of nodes may crash in fast mode, in which case, crash recovery is not possible: the number of nodes that are in recovered state or previously crashed in slow mode will be less than a bare minority. During such simultaneous crashes, which we believe are extremely rare, SAUCR remains unavailable.

In the presence of partitions, all nodes could be alive, but partitioned into two; in such a case, the minority partition would be temporarily unavailable while the other partition will safely move to slow mode if a bare majority are connected within the partition. The nodes in the minority partition would realize they are not connected to the leader and flush to disk. Both of these actions guarantee durability and prevent future unavailability.

4 Implementation

We have implemented situation-aware updates and crash recovery in Apache ZooKeeper (v3.4.8). We now describe the most important implementation details.

Storage layer. ZooKeeper maintains an on-disk log to which the updates are appended. ZooKeeper also maintains snapshots and meta information (e.g., current epoch). We modified the log-update protocol to not issue `fsync` calls synchronously in fast mode. Snapshots are periodically written to disk; because the snapshots are taken in the background, foreground performance is unaffected. The meta information is always synchronously updated. Fortunately, such synchronous updates happen rarely (only when the leader changes), and thus do

not affect common-case performance. In addition to the above structures, SAUCR maintains the *fast-switch-entry* in a separate file and synchronously updates it the first time when the node processes an entry in the fast mode. In slow mode, the LLE-MAP is synchronously persisted. SAUCR maintains the map at the head of the log file. The *latest-on-disk-entry* for a node is its own entry in the persistent LLE-MAP (LLE-MAP is keyed by node-id).

Replication. We modified the *QuorumPacket* [9] (which is used by the leader for replication) to include the mode flag and the LLE-MAP. The leader transitions to fast mode after receiving three consecutive successful replication acknowledgements from more than a bare majority.

Failure Reaction. In our implementation, the nodes detect failures through missing heartbeats, missing responses, and broken socket connections. Although quickly reacting to failures and flushing or switching modes is necessary to prevent data loss or unavailability, hastily declaring a node as failed might lead to instability. For example, if a follower runs for an election after missing just one heartbeat from the leader, the system may often change leader, affecting progress. SAUCR's implementation avoids this scenario as follows. On missing the first heartbeat from the leader, the followers *suspect* that the leader might have failed and so quickly react to the suspected failure by flushing their buffers. However, they conservatively wait for a handful of missing heartbeats before *declaring* the leader as failed and running for an election. Similarly, while the leader initiates a mode switch on missing the first heartbeat response, it waits for a few missing responses before declaring the follower as failed. If a majority of followers have not responded to a few heartbeats, the leader steps down and becomes a candidate.

Recovery Protocol. We modified the leader election protocol so that a node recovering from a fast-mode crash first recovers its LLE before it can participate in elections. A responding node correctly handles LLE-query from a node and replication requests from the leader that arrive concurrently. If a node that recovers from a fast-mode crash becomes the leader, it fetches the data items up to its LLE from others. However, due to the background flushes, several items might already be present on the disk; the node recovers only the missing items. The responding node batches several items in its response.

5 Evaluation

We now evaluate the durability, availability, and performance of our SAUCR implementation.

5.1 Durability and Availability

To evaluate the guarantees of SAUCR, we developed a cluster crash-testing framework. The framework first generates a graph of all possible cluster states as shown in Figure 6. Then, it generates a set of *cluster-state sequences*. For instance, $12345 \rightarrow 345 \rightarrow 45 \rightarrow 1245 \rightarrow 1 \rightarrow 13 \rightarrow 12345$ is one such sequence. In this sequence, at first, all five nodes are alive; then, two nodes (1 and 2) crash; then, 3 crashes; next, 1 and 2 recover; then 2, 4, 5 crash; 3 recovers; finally, 2, 4, 5 recover. To generate a sequence, we start from the root state where all nodes are alive. We visit a child with a probability that decreases with the length of the path constructed so far, and the difference in the number of alive nodes between the parent and the child. We produced 1264 such sequences (498 and 766 for a 5-node and 7-node cluster, respectively).

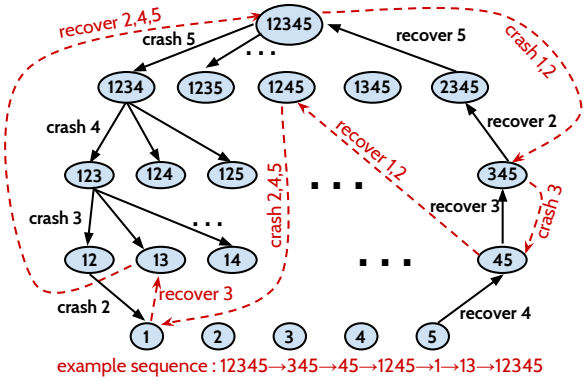


Figure 6: **Cluster-State Sequences.** *The figure shows the possible cluster states for a five-node cluster and how cluster-state sequences are generated. One example cluster-state sequence is traced.*

The cluster-state sequences help test multiple update and recovery code paths in SAUCR. For example, in the above sequence, 12345 would first operate in fast mode; then 345 would operate in slow mode; then 1245 would operate in fast mode; 1 would flush to disk on detecting that other nodes have crashed; in the penultimate state, 3 would recover from a slow-mode crash; in the last state, 2, 4, and 5 would recover from a fast-mode crash.

Within each sequence, at each intermediate cluster state, we insert new items if possible (if a majority of nodes do not exist, we cannot insert items). $12345^a \rightarrow 345^b \rightarrow 45 \rightarrow 1245^c \rightarrow 1 \rightarrow 13 \rightarrow 12345^d$ shows how entries *a-d* are inserted at various stages. In the end, the framework reads all the acknowledged items. If the cluster does not become available and respond to the queries, we flag the sequence as unavailable for the system under test. If the system silently loses the committed items, then we flag the sequence as data loss.

We subject the following four systems to the cluster-crash sequences: memory-durable ZK (ZooKeeper with the *forceSync* flag turned off), VR (viewstamped replication), disk-durable ZK (ZooKeeper with *forceSync* turned on), and finally SAUCR. Existing VR implementations [65] do not support a read/write interface, preventing us from directly applying our crash-testing framework to them. Therefore, we developed an *ideal* model of VR that resembles a perfect implementation.

System	Nodes	Non-simultaneous				Simultaneous				
		Total	Correct	Unavailable	Data loss	Scenario	Total	Correct	Unavailable	Data loss
ZK-mem	5	498	248	0	250	n/a	498	248	0	250
	7	766	455	0	311	n/a	766	455	0	311
VR-ideal	5	498	28	470	0	n/a	498	28	470	0
	7	766	189	577	0	n/a	766	189	577	0
ZK-disk	5	498	498	0	0	n/a	498	498	0	0
	7	766	766	0	0	n/a	766	766	0	0
SAUCR	5	498	498	0	0	other	475	475	0	0
						!min-rec	23	0	23	0
	7	766	766	0	0	other	725	725	0	0
						!min-rec	41	0	41	0

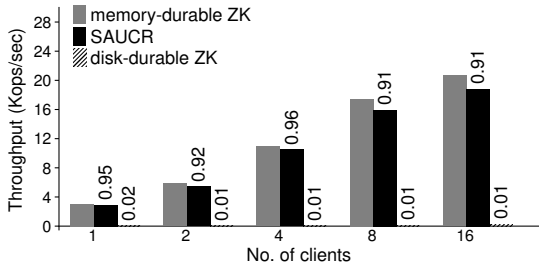
Table 2: **Durability and Availability.** *The table shows the durability and availability of memory-durable ZK (ZK-mem), VR (VR-ideal), disk-durable ZK (ZK-disk), and* SAUCR. *!min-rec denotes that only less than a bare minority are in recovered state.*

5.1.1 Non-simultaneous Crashes

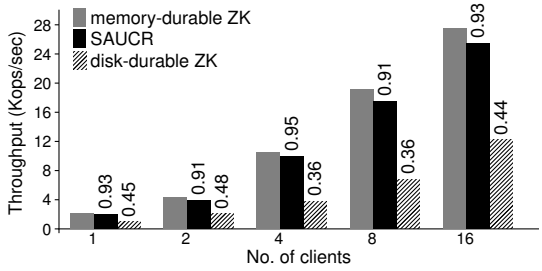
We first test all sequences considering that failures are non-simultaneous. For example, when the cluster transitions from 12345 to 345, we crash nodes 1 and 2 one after the other (with a gap of 50 ms). Table 2 shows the results. As shown, the memory-durable ZK loses data in about 50% and 40% of the cases in the 5-node and 7-node tests, respectively. The ideal VR model does not lose data; however, it leads to unavailability in about 90% and 75% of the cases in the 5-node and 7-node tests, respectively. As expected, disk-durable ZooKeeper is safe. In contrast to memory-durable ZK and VR, SAUCR remains durable and available in all cases. Because failures are non-simultaneous in this test, the leader detects failures and switches to slow mode; similarly, the followers quickly flush to disk if the leader crashes, leading to correct behavior.

5.1.2 Simultaneous Crashes

We next assume that failures are simultaneous. For example, if the cluster state transitions from 124567 to 12, we crash all four nodes at the same time, without any gap. Note that during such a failure, SAUCR would be operating in fast mode and suddenly many nodes would crash simultaneously, leaving behind less than a bare minority. In such cases, less than a bare minority would be in the *recovered* state; SAUCR cannot handle such cases. Table 2 shows the results. As shown, memory-durable ZK loses data in all cases in which it lost data in the non-simultaneous test. This is because memory-durable ZK loses data, irrespective of the simultaneity of the crashes. Similarly, VR is unavailable in all the cases where it was unavailable in the non-simultaneous crash tests. As expected, disk-durable ZK remains durable and available. SAUCR remains unavailable in a few cases by its design.



(a) HDD (cluster-1)



(b) SSD (cluster-2)

Figure 7: **Micro-benchmarks.** *(a) and (b) show the update throughput on memory-durable ZK,* SAUCR, *and disk-durable ZK on HDDs and SSDs, respectively. Each request is 1KB in size. The number on top of each bar shows the performance normalized to that of memory-durable ZK.*

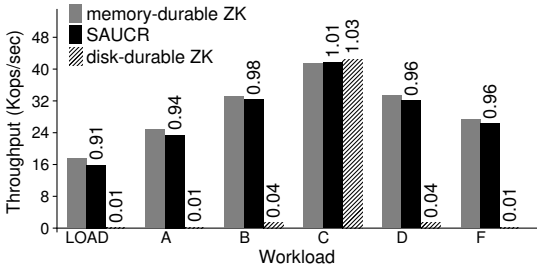
5.2 Performance

We conducted our performance experiments on two clusters (cluster-1: HDD, cluster-2: SSD), each with five machines. The HDD cluster has a 10 Gb network, and each node is a 20-core Intel Xeon CPU E5-2660 machine with 256 GB memory running Linux 4.4, with a 1-TB HDD. The SSD cluster has 10 Gb network, and each node is a 20-core Intel E5-2660 machine with 160 GB memory running Linux 4.4, with a 480-GB SSD. Numbers reported are the average over five runs.

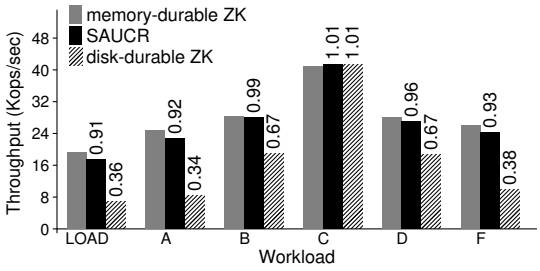
5.2.1 Update Micro-benchmark

We now compare SAUCR's performance against memory-durable ZK and disk-durable ZK. We conduct this experiment for an update-only micro-benchmark.

Figure 7(a) and (b) show the results on HDDs and SSDs, respectively. As shown in the figure, SAUCR's performance is close to the performance of memory-durable ZK (overheads are within 9% in the worst case). Note that SAUCR's performance is close to memory-durable ZK but not equal; this small gap exists because, in the fast mode, SAUCR commits a request only after four nodes (majority + 1) acknowledge, while memory-durable ZK commits a request after three nodes (a bare majority) acknowledge. Although the requests are sent to the followers in parallel, waiting for acknowledgment from one additional follower adds some delay. Compared to disk-durable ZK, as expected, both memory-durable ZK and SAUCR are significantly faster. On HDDs, they are about 100× faster. On SSDs, however,



(a) HDD (cluster-1)



(b) SSD (cluster-2)

Figure 8: **Macro-benchmarks.** *The figures show the throughput under various YCSB workloads for memory-durable ZK,* SAUCR*, and disk-durable ZK for eight clients. The number on top of each bar shows the performance normalized to that of memory-durable ZK.*

the performance gap is less pronounced. For instance, with a single client, memory-durable ZK and SAUCR are only about 2.1× faster than disk-durable ZK. We found that this inefficiency arises because of software overheads in ZooKeeper's implementation that become dominant atop SSDs.

5.2.2 YCSB Workloads

We now compare the performance of SAUCR against memory-durable ZK and disk-durable ZK across the following six YCSB [23] workloads: load (all writes), A (w:50%, r:50%), B (w:5%, r:95%), C (only reads), D (read latest, w:5%, r:95%), F (read-modify-write, w:50%, r:50%). We use 1KB requests.

Figure 8(a) and (b) show the results on HDDs and SSDs, respectively. For all workloads, SAUCR closely matches the performance of memory-durable ZK; again, the small overheads are a result of writing to one additional node. For write-heavy workloads (load, A, F), SAUCR's performance overheads are within 4% to 9% of memory-durable ZK. For such workloads, memory-durable ZK and SAUCR perform notably better than disk-durable ZK (about 100× and 2.5× faster on HDDs and SSDs, respectively). For workloads that perform mostly reads (B and D), SAUCR's overheads are within 1% to 4% of memory-durable ZK. For such read-heavy workloads, memory-durable ZK and SAUCR are about 25× and 40% faster than disk-durable ZK on HDDs and SSDs, respectively. For the read-only workload (C), all three systems perform the same on both HDDs and SSDs because reads are served only from memory.

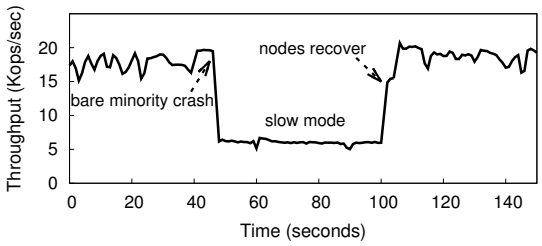


Figure 9: **Performance Under Failures.** *The figure shows* SAUCR*'s performance under failures; we conduct this experiment with eight clients running an update-only workload on SSDs.*

5.2.3 Performance Under Failures

In all our previous performance experiments, we showed how SAUCR performs in its fast mode (without failures). When failures arise and if only a bare majority of nodes are alive, SAUCR switches to the slow mode until enough nodes recover. Figure 9 depicts how SAUCR detects failures and switches to slow mode when failures arise. However, when enough nodes recover from the failure, SAUCR switches back to fast mode.

5.3 Heartbeat Interval vs. Performance

SAUCR uses heartbeats to detect failures. We now examine how varying the heartbeat interval affects workload performance. Intuitively, short and aggressive intervals would enable quick detection but lead to worse performance. Short intervals may degrade performance for two reasons: first, the system would load the network with more packets; second, the SAUCR nodes would consider a node as failed upon a missing heartbeat/response when the node was merely slow and thus react spuriously by flushing to disk or switching to slow mode.

To tackle the first problem, when replication requests are flowing actively, SAUCR treats the requests themselves as heartbeats; further, we noticed that even when the heartbeat interval is lower than a typical replication-request latency, the additional packets do not affect the workload performance significantly. The second problem of spurious reactions can affect performance.

For the purpose of this experiment, we vary the heartbeat interval from a small (and unrealistic) value such as 1 μs to a large value of 1 second. We measure three metrics: throughput, the number of requests committed in slow mode (caused by the leader suspecting follower failures), and the number of flushes issued by a follower (caused by followers suspecting a leader failure). Figure 10 shows the result. As shown, when the interval is equal to or greater than 1 ms, the workload performance remains mostly unaffected. As expected, with such reasonably large intervals, even if the nodes are slow occasionally, the likelihood that a node will not receive a heartbeat or a response is low; thus, the nodes do not react spuriously most of the times. As a result, only a few spurious flushes are issued by the followers, and very few

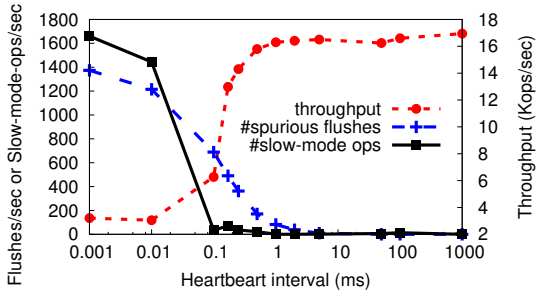


Figure 10: **Heartbeat Interval vs. Performance.** *The figure shows how varying the heartbeat interval affects performance. The left y-axis shows the average number of flushes issued by a follower per second or the average number of requests committed in slow mode by the leader per second. We measure the performance (right y-axis) by varying the heartbeat interval (x-axis). We conduct this experiment with eight clients running the YCSB-load workload on SSDs.*

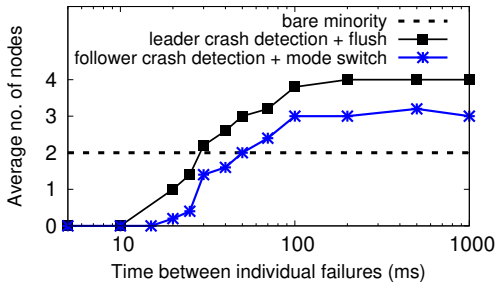


Figure 11: **Correlated Failure Reaction.** *The figure shows how quickly* SAUCR *reacts to correlated failures; the y-axis denotes the number of nodes that detect and flush to disk before all nodes crash when we vary the time between the individual failures (x-axis). We conduct this experiment on SSDs.*

requests are committed in slow mode. In contrast, when the interval is less than 1 ms, the SAUCR nodes react more aggressively, flushing more often and committing many requests in slow mode, affecting performance. In summary, for realistic intervals of a few tens of milliseconds (used in other systems [28]) or even for intervals as low as 1 ms, workload performance remains unaffected.

Finally, although the nodes react aggressively (with short intervals), they do not declare a node as failed because there are no actual failures in this experiment. As a result, we observe that the leader does not step down and the followers do not run for an election.

5.4 Correlated Failure Reaction

We now test how quickly SAUCR detects and reacts to a correlated failure that crashes all the nodes. On such a failure, if at least a bare minority of nodes flush the data to disks before all nodes crash, SAUCR will be able to provide availability and durability when the nodes later recover. For this experiment, we use a heartbeat interval value of 50 ms. We conduct this experiment on a five-node cluster in two ways.

First, we crash the active leader and then successively crash all the followers. We vary the time between the individual failures and observe how many followers detect and flush to disk before all nodes crash. For each failure-gap time, we run the experiment five times and report the average number of nodes that safely flush to disk. Figure 11 shows the result: if the time between the failures is greater than 30 ms, then at least a bare minority of followers always successfully flush the data, ensuring availability and durability.

Second, we crash the followers, one after the other. In this case, the leader detects the failures and switches to slow mode. As shown in the figure, if the time between the individual failures is greater than 50 ms, the system will be available and durable after recovery. As we discussed earlier (§2.3), in a real deployment, the time between individual failures is almost always greater than 50 ms; therefore, in such cases, with a heartbeat interval of 50 ms, SAUCR will always remain safe.

Note that we run this experiment with a 50-ms heartbeat interval; shorter intervals (such as 1 ms used in Figure 10) will enable the system to remain durable and available (i.e., a bare minority or more nodes would safely flush or switch to slow mode) even when the failures are only a few milliseconds apart.

6 Discussion

We now discuss two concerns related to SAUCR's adoption in practice. First, we examine whether SAUCR will offer benefits with the advent of faster storage devices such as non-volatile memory (NVM). Second, we discuss whether applications will be tolerant of having low throughput when SAUCR operates in slow mode.

Faster Storage Devices. The reliability of memory-durable approaches can be significantly improved if every update is forced to disk. However, on HDDs or SSDs, the overhead of such synchronous persistence is prohibitively expensive. New storage devices such as NVMe-SSDs and NVM have the potential to reduce the cost of persistence and thus improve reliability with low overheads. However, even with the advent of such faster storage, we believe SAUCR has benefits for two reasons.

First, although NVMe-SSDs are faster than HDDs and SSDs, they are not as fast as DRAM. For example, a write takes 30 μs on Micron NVMe-SSDs which is two orders of magnitude slower than DRAM [19] and thus SAUCR will have performance benefits compared to NVMe-SSDs. While NVM and DRAM exhibit the same latencies for reads, NVM writes are more expensive (roughly by a factor of 5) [40, 67]. Further, writing a few tens of kilobytes (as a storage system would) will be slower than published numbers that mostly deal with writing cachelines. Hence, even with NVMs, SAUCR will demonstrate benefit.

Second, and more importantly, given the ubiquity of DRAM and their lower latencies, many current systems

and deployments choose to run memory-only clusters for performance [17, 43], and we believe this trend is likely to continue. SAUCR would increase the durability and availability of such non-durable deployments significantly without affecting their performance at no additional cost (i.e., upgrading to new hardware).

Low Performance in Slow Mode. Another practical concern regarding SAUCR's use in real deployments is that of the low performance that applications may experience in slow mode. While SAUCR provides low performance in slow mode, we note that this trade-off is a significant improvement over other existing methods that can either lead to permanent unavailability or lose data. Further, in a shared-storage setting, we believe many applications with varying performance demands will coexist. While requests from a few latency-sensitive applications may time out, SAUCR allows other applications to make progress without any hindrance. Furthermore, in slow mode, only update requests pay the performance penalty, while most read operations can be served without any overheads (i.e., at about the same latency as in the fast mode). Finally, this problem can be alleviated with a slightly modified system that can be reconfigured to include standby nodes when in slow mode for a prolonged time. Such reconfiguration would enable the system to transition out of the slow mode quickly. We believe this extension could be an avenue for future work.

7 Related Work

We now discuss how prior systems and research efforts relate to various aspects of our work.

Situation-Aware Updates. The general idea of dynamically transitioning systems between different modes is common in real-time systems [15]. Similarly, the idea of fault-detection-triggered mode changes has been used in cyber-physical distributed systems [18]. However, we do not know of any previous work that dynamically adapts a distributed update protocol to the current situation. Many practical systems statically define whether updates will be flushed to disk or not [8, 22, 27, 62]. A few systems, such as MongoDB, provide options to specify the durability of a particular request [51]. However such dynamicity of whether the request will be persisted or buffered is purely client-driven: the storage system does not automatically make any such decisions, depending on the current failures.

Recovery. RAMCloud [58,64] has a similar flavor to our work. However, the masters always construct their data from remote backups, unlike SAUCR, which performs mode-specific recovery. SAUCR's recovery is similar to VR's recovery [48]. However, SAUCR's recovery differs from that of VR in two ways. First, in VR, a recovering node waits for a majority responses before it moves to the recovered state, while in SAUCR, a recovering node has to wait only for a bare minority responses. Second, and more importantly, in VR, a responding node can readily be in the recovered state only if it has not yet crashed. In contrast, in SAUCR, a node can readily be in the recovered state in two ways: either it could have operated in fast mode and not failed yet, or it might have operated in slow mode previously or flushed to disk. These differences improves SAUCR's availability. SAUCR's recovery is also similar to how replicated-state-machine (RSM) systems recover corrupted data from copies [1].

Performance Optimizations in RSM systems. Several prior efforts have optimized majority-based RSM systems by exploiting network properties [47, 61]; other optimizations have also been proposed [41, 52]. However, to our knowledge, most of these systems are only memory-durable. SAUCR can augment such systems to provide stronger guarantees while not compromising on performance. A few systems [14, 21, 56] realize that synchronous disk writes are a major bottleneck; these systems have proposed techniques (e.g., batching) that make disk I/O efficient. SAUCR's implementation includes such optimizations in its slow mode.

8 Conclusion

Fault-tolerant replication protocols are the foundation upon which many data-center systems and applications are built. Such a foundation needs to perform well, yet also provide a high level of reliability. However, existing approaches either suffer from low performance or can lead to poor durability and availability. In this paper, we have presented situation-aware updates and crash recovery (SAUCR), a new approach to replication within a distributed system. SAUCR reacts to failures and adapts to current conditions, improving durability and availability while maintaining high performance. We believe such a situation-aware distributed update and recovery protocol can serve as a better foundation upon which reliable and performant systems can be built.

Acknowledgments

We thank Andreas Haeberlen (our shepherd) and the OSDI reviewers for their thoughtful suggestions that improved the presentation of the content significantly. We thank the members of ADSL for their valuable feedback. We also thank CloudLab [63] for providing a great environment for running our experiments. This material was supported by funding from NSF grants CNS-1421033, CNS-1218405, and CNS-1838733, DOE grant DE-SC0014935, and donations from EMC, Huawei, Microsoft, NetApp, and VMware. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-Aware Recovery for Consensus-Based Storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST '18)*, Oakland, CA, February 2018.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated Crash Vulnerabilities. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[3] Amazon Elastic Compute Cloud. Regions and Availability Zones. `https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html`.

[4] Apache. ZooKeeper. `https://zookeeper.apache.org/`.

[5] Apache. ZooKeeper Configuration Parameters. `https://zookeeper.apache.org/doc/r3.1.2/zookeeperAdmin.html#sc_configuration`.

[6] Apache. ZooKeeper Leader Activation. `https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_leaderElection`.

[7] Apache Accumulo Users. Setting ZooKeeper forceSync=no. `http://apache-accumulo.1065345.n5.nabble.com/setting-zookeeper-forceSync-no-td7758.html`.

[8] Apache Cassandra. Cassandra Wiki: Durability. `https://wiki.apache.org/cassandra/Durability`.

[9] Apache ZooKeeper. QuorumPacket Class. `http://people.apache.org/~larsgeorge/zookeeper-1075002/build/docs/dev-api/org/apache/zookeeper/server/quorum/QuorumPacket.html`.

[10] Apache ZooKeeper. ZooKeeper Overview. `https://zookeeper.apache.org/doc/r3.5.0-alpha/zookeeperOver.html`.

[11] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.0 edition, May 2015.

[12] Mehmet Bakkaloglu, Jay J Wylie, Chenxi Wang, and Gregory R Ganger. On Correlated Failures in Survivable Storage Systems. Technical Report CMU-CS-02-129, School of computer science, Carnegie-Mellon University, 2002.

[13] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. CORFU: A Shared Log Design for Flash Clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI '12)*, San Jose, CA, April 2012.

[14] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. Paxos Replicated State Machines As the Basis of a High-performance Data Store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI '11)*, Boston, MA, April 2011.

[15] Alan Burns. System Mode Changes - General and Criticality-Based. In *Proc. of 2nd Workshop on Mixed Criticality Systems (WMC)*, pages 3–8, 2014.

[16] Miguel Castro and Barbara Liskov. Practical Byzantine Fault Tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, New Orleans, Louisiana, February 1999.

[17] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos Made Live: An Engineering Perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*, Portland, OR, August 2007.

[18] Ang Chen, Hanjun Xiao, Andreas Haeberlen, and Linh Thi Xuan Phan. Fault Tolerance and the Five-second Rule. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*, Kartause Ittingen, Switzerland, May 2015.

[19] Chris Mellor. Storage with the speed of memory? XPoint, XPoint, that's our plan. `https://www.theregister.co.uk/2016/04/21/storage_approaches_memory_speed_with_xpoint_and_storageclass_memory/`.

[20] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data

Loss in Cloud Storage. In *Proceedings of the USENIX Annual Technical Conference (USENIX '13)*, San Jose, CA, June 2013.

[21] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright Cluster Services. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, Big Sky, Montana, October 2009.

[22] CockroachDB. CockroachDB Cluster Settings. `https://www.cockroachlabs.com/docs/stable/cluster-settings.html`.

[23] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC '10)*, Indianapolis, IA, June 2010.

[24] CoreOS. etcd Guarantees. `https://coreos.com/blog/etcd-v230.html`.

[25] DataCenterKnowledge. Lightning Disrupts Google Cloud Services. `http://www.datacenterknowledge.com/archives/2015/08/19/lightning-strikes-google-data-center-disrupts-cloud-services/`.

[26] Jeffrey Dean and Luiz André Barroso. The Tail at Scale. *Communications of the ACM*, 56(2):74–80, 2013.

[27] Elasticsearch. Translog Settings. `https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-translog.html#_translog_settings`.

[28] Etcd. Etcd Tuning. `https://coreos.com/etcd/docs/latest/tuning.html`.

[29] Flavio Junqueira. Transaction Logs and Snapshots. `https://mail-archives.apache.org/mod_mbox/zookeeper-user/201504.mbox/%3CDA045626-54A4-4F8A-96C0-69DA574D9807@yahoo.com%3E`.

[30] Flavio Junqueira. [ZooKeeper-user] forceSync=no. `http://grokbase.com/p/zookeeper/user/126g0063x4/forcesync-no`.

[31] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, December 2010.

[32] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, Santa Clara, CA, February 2017.

[33] Google. Google Cluster Data. `https://github.com/google/cluster-data`.

[34] Google Code University. Introduction to Distributed System Design. `http://www.hpcs.cs.tsukuba.ac.jp/~tatebe/lecture/h23/dsys/dsd-tutorial.html`.

[35] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI '05)*, Boston, MA, May 2005.

[36] Henry Robinson. Consensus Protocols: Paxos. `http://the-paper-trail.org/blog/consensus-protocols-paxos/`.

[37] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX '10)*, Boston, MA, June 2010.

[38] Jay Kreps. Using forceSync=no in Zookeeper. `https://twitter.com/jaykreps/status/363720100332843008`.

[39] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance Broadcast for Primary-backup Systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '11)*, Hong Kong, China, June 2011.

[40] Sudarsun Kannan, Nitish Bhat, Ada Gavrilovska, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Redesigning LSMs for Nonvolatile Memory with NoveLSM. In *Proceedings of the USENIX Annual Technical Conference (USENIX '18)*, Boston, MA, July 2018.

[41] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All About Eve: Execute-verify Replication for Multi-core Servers. In *Proceedings of the 10th Symposium on Operating Systems Design and Implementation (OSDI '12)*, Hollywood, CA, October 2012.

[42] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, April 2004.

[43] Ken Birman. What we can learn about specifications from ZooKeeper's asynchronous mode, and its unsafe ForceSync=no option? `http://thinkingaboutdistributedsystems.blogspot.com/2017/09/what-we-can-learn-from-zookeepers.html`.

[44] Madhukar Korupolu and Rajmohan Rajaraman. Robust and Probabilistic Failure-Aware Placement. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 213–224. ACM, 2016.

[45] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 32(4):18–25, 2001.

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

[47] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just Say No to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

[48] Barbara Liskov and James Cowling. Viewstamped Replication Revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT CSAIL, 2012.

[49] LogCabin. LogCabin. `https://github.com/logcabin/logcabin`.

[50] Microsoft Azure. Azure Availability Sets. `https://docs.microsoft.com/en-us/azure/virtual-machines/windows/tutorial-availability-sets#availability-set-overview`.

[51] MongoDB. MongoDB Write Concern. `https://docs.mongodb.com/manual/reference/write-concern/`.

[52] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There is More Consensus in Egalitarian Parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, Nemacolin Woodlands Resort, Farmington, Pennsylvania, October 2013.

[53] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. Extracting More Concurrency from Distributed Transactions. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[54] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI '06)*, San Jose, CA, May 2006.

[55] Brian M Oki and Barbara H Liskov. 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*, ON, Canada, August 1988.

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

[57] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the USENIX Annual Technical Conference (USENIX '14)*, Philadelphia, PA, June 2014.

[58] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[59] Parsely Inc. Streamparse: Configuring Zookeeper with forceSync = no. `https://github.com/Parsely/streamparse/issues/168`.

[60] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[61] Dan R. K. Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing Distributed Systems Using Approximate Synchrony in Data Center Networks. In *Proceedings of the 12th Symposium on Networked Systems Design and Implementation (NSDI '15)*, Oakland, CA, May 2015.

[62] RethinkDB. RethinkDB Settings - Durability. `https://rethinkdb.com/docs/consistency/`.

[63] Robert Ricci, Eric Eide, and CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), 2014.

[64] Ryan Scott Stutsman. *Durability and Crash Recovery in Distributed In-Memory Storage Systems*. PhD thesis, Stanford University, 2013.

[65] UWSysLab. VR Implementation. `https://github.com/UWSysLab/NOPaxos/tree/master/vr`.

[66] Hakim Weatherspoon, Tal Moscovitz, and John Kubiatowicz. Introspective Failure Analysis: Avoiding Correlated Failures in Peer-to-Peer Systems. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems*, pages 362–367. IEEE, 2002.

[67] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, Santa Clara, CA, February 2016.

[68] Zookeeper User Mailing List. Unavailability Issues due to Setting forceSync=no in ZooKeeper. `http://zookeeper-user.578899.n2.nabble.com/forceSync-no-td7577568.html`.