Protocol-Aware Recovery for Consensus-Based Distributed Storage

RAMNATTHAN ALAGAPPAN and AISHWARYA GANESAN,
University of Wisconsin – Madison
ERIC LEE, University of Texas – Austin
AWS ALBARGHOUTHI, University of Wisconsin – Madison
VIJAY CHIDAMBARAM, University of Texas – Austin
ANDREA C. ARPACI-DUSSEAU and REMZI H. ARPACI-DUSSEAU,
University of Wisconsin – Madison

We introduce *protocol-aware recovery* (PAR), a new approach that exploits protocol-specific knowledge to correctly recover from storage faults in distributed systems. We demonstrate the efficacy of PAR through the design and implementation of corruption-tolerant replication (CTRL), a PAR mechanism specific to replicated state machine (RSM) systems. We experimentally show that the CTRL versions of two systems, LogCabin and ZooKeeper, safely recover from storage faults and provide high availability, while the unmodified versions can lose data or become unavailable. We also show that the CTRL versions achieve this reliability with little performance overheads.

CCS Concepts: • **General and reference → Reliability**; • **Information systems → Distributed storage**; • **Computer systems organization → Redundancy**; • **Software and its engineering → File systems management**;

Additional Key Words and Phrases: Storage faults, data corruption, fault tolerance, consensus

ACM Reference format:
Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. Protocol-Aware Recovery for Consensus-Based Distributed Storage. *ACM Trans. Storage* 14, 3, Article 21 (October 2018), 30 pages.
https://doi.org/10.1145/3241062

This material was supported by funding from NSF grants CNS-1421033 and CNS-1218405, DOE grant DE-SC0014935, and donations from EMC, Huawei, Microsoft, 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. This article is an extended version of a FAST '18 paper by Alagappan et al. [7]. The additional material here includes a discussion on how PAR can be applied to other systems, a proof of why crashes and corruptions cannot be always disentangled, an overview diagram that summarizes the entire recovery protocol, new performance experiments, new figures explaining leader-initiated snapshots, and many other small updates.

Authors' addresses: R. Alagappan and A. Ganesan, 1210 W. Dayton St., Madison, WI 53706; emails: {ra, ag}@cs.wisc.edu; E. Lee, 2317 Speedway, Austin, TX 78712; email: ericlee123@utexas.edu; A. Albarghouthi, 1210 W. Dayton St., Madison, WI 53706; email: aws@cs.wisc.edu; V. Chidambaram, 2317 Speedway, Austin, TX 78712; email: vijay@cs.utexas.edu; A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau, 1210 W. Dayton St., Madison, WI 53706; emails: dusseau@cs.wisc.edu, remzi@cs.wisc.edu.

1553-3077/2018/10-ART21 $15.00
https://doi.org/10.1145/3241062

1 INTRODUCTION

Failure recovery using redundancy is central to improved reliability of distributed systems [15, 23, 33, 37, 65, 71]. Distributed systems recover from node crashes and network failures using copies of data and functionality on several nodes [7, 49, 59]. Similarly, bad or corrupted data on one node should be recovered from redundant copies.

In a *static* setting where all nodes always remain reachable and where clients do not actively update data, recovering corrupted data from replicas is straightforward; in such a setting, a node could repair its state by simply fetching the data from any other node.

In reality, however, a distributed system is a *dynamic* environment, constantly in a state of flux. In such settings, orchestrating recovery correctly is surprisingly hard. As a simple example, consider a quorum-based system, in which a piece of data is corrupted on one node. When the node tries to recover its data, some nodes may fail and be unreachable, and some nodes may have recently recovered from a failure and so lack the required data or hold a stale version. If enough care is not exercised, the node could "fix" its data from a stale node, overwriting the new data, potentially leading to a data loss.

To correctly recover corrupted data from redundant copies in a distributed system, we propose that a recovery approach should be *protocol-aware*. A *protocol-aware recovery* (PAR) approach is carefully designed based on how the distributed system performs updates to its replicated data, elects the leader, and so on. For instance, in the previous example, a PAR mechanism would realize that a faulty node has to query at least R (read quorum) other nodes to safely and quickly recover its data.

In this article, we apply PAR to replicated state machine (RSM) systems. We focus on RSM systems for two reasons. First, correctly implementing recovery is most challenging for RSM systems because of the strong consistency and durability guarantees they provide [62]; a small misstep in recovery could violate the guarantees. Second, the reliability of RSM systems is crucial: many systems entrust RSM systems with their critical data [47]. For example, Bigtable, GFS, and other systems [8, 28] store their metadata on RSM systems such as Chubby [17] or ZooKeeper [5]. Hence, protecting RSM systems from storage faults such as data corruption will improve the reliability of many dependent systems.

We first characterize the different approaches to handling storage faults by developing the *RSM recovery taxonomy*, through experimental and qualitative analysis of practical systems and methods proposed by prior research (Section 2). Our analyses show that most approaches employed by currently deployed systems do not use any protocol-level knowledge to perform recovery, leading to disastrous outcomes such as data loss and unavailability.

Thus, to improve the resiliency of RSM systems to storage faults, we design a new protocol-aware recovery approach that we call corruption-tolerant replication or CTRL (Section 3). CTRL constitutes two components: a *local storage layer* and a *distributed recovery protocol*; while the storage layer reliably detects faults, the distributed protocol recovers faulty data from redundant copies. Both the components carefully exploit RSM-specific knowledge to ensure safety (e.g., no data loss) and high availability.

CTRL applies several novel techniques to achieve safety and high availability. For example, a *crash-corruption disentanglement* technique in the storage layer distinguishes corruptions caused by crashes from disk faults; without this technique, safety violations or unavailability could result. Next, a *global-commitment determination* protocol in the distributed recovery separates committed items from uncommitted ones; this separation is critical: while recovering faulty committed items is necessary for safety, discarding uncommitted items quickly is crucial for availability. Finally, a novel *leader-initiated snapshotting* mechanism enables identical snapshots across nodes to greatly simplify recovery.

We implement Ctrl in two storage systems that are based on different consensus algorithms: LogCabin [45] (based on Raft [53]) and ZooKeeper [5] (based on ZAB [41]) (Section 4). Through experiments, we show that Ctrl versions provide safety and high availability in the presence of storage faults, while the original systems remain unsafe or unavailable in many cases; we also show that Ctrl induces minimal performance overhead (Section 5).

2 BACKGROUND AND MOTIVATION

We first provide background on storage faults and RSM systems. We then present the taxonomy of different approaches to handling storage faults in RSM systems.

2.1 Storage Faults in Distributed Systems

Disks and flash devices exhibit a subtle and complex failure model: a few blocks of data could become inaccessible or be silently corrupted [9, 10, 34, 63]. Although such storage faults are rare compared to whole-machine failures, in large-scale distributed systems, even rare failures become prevalent [64, 66]. Thus, it is critical to reliably detect and recover from storage faults.

Storage faults occur due to several reasons: media errors [11], program/read disturbance [64], and bugs in firmware [10], device drivers [70], and file systems [29, 30]. Storage faults manifest in two ways: block *errors* and *corruption*. Block errors (or latent sector errors) arise when the device internally detects a problem with a block and throws an error upon access. Studies of both flash [35, 64] and hard drives [11, 63] show that block errors are common. Corruption could occur due to lost and misdirected writes that may not be detected by the device. Studies [10, 54] and anecdotal evidence [38, 39, 61] show the prevalence of data corruption in the real world.

Many local file systems, on encountering a storage fault, simply propagate the fault to applications [12, 57, 68]. For example, ext4 silently returns corrupted data if the underlying device block is corrupted. In contrast, a few file systems transform an underlying fault into a different one; for example, btrfs returns an error to applications if the accessed block is corrupted on the device. In either case, storage systems built atop local file systems should handle corrupted data and storage errors to preserve end-to-end data integrity.

One way to tackle storage faults is to use RAID-like storage to maintain multiple copies of data on each node. However, many distributed deployments would like to use inexpensive disks [23, 33]. Given that the data in a distributed system is inherently replicated, it is wasteful to store multiple copies on each node. Hence, it is important for distributed systems to use the inherent redundancy to recover from storage faults.

2.2 RSM-Based Storage Systems

Our goal is to harden RSM systems to storage faults. In an RSM system, a set of nodes compute identical states by executing commands on a state machine (an in-memory data structure on each node) [62]. Typically, clients interact with a single node (the leader) to execute operations on the state machine. Upon receiving a command, the leader durably writes the command to an on-disk *log* and replicates it to the followers. When a majority of nodes have durably persisted the command in their logs, the leader applies the command to its state machine and returns the result to the client; at this point, the command is committed. The commands in the log have to be applied to the state machine *in-order*. Losing or overwriting committed commands violates the safety property of the state machine. The replicated log is kept consistent across nodes by a consensus protocol such as Paxos [43], ZAB [41], or Raft [53].

Because the log can grow indefinitely and exhaust disk space, periodically, a *snapshot* of the in-memory state machine is written to disk and the log is garbage collected. When a node restarts after a crash, it restores the system state by reading the latest on-disk snapshot and the log. The node

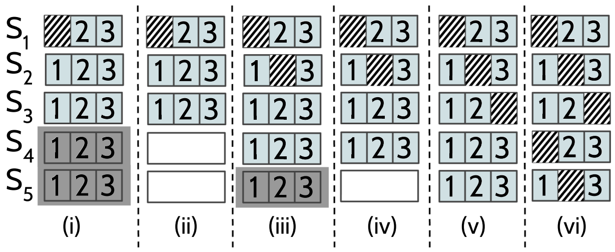


Fig. 1. **Sample Scenarios.** The figure shows sample scenarios in which current approaches fail. Faulty entries are striped. Crashed and lagging nodes are shown as gray and empty boxes, respectively.

also recovers its critical metadata (e.g., log start index) from a structure called *metainfo*. Thus, each node maintains three critical persistent data structures: the *log*, the *snapshots*, and the *metainfo*.

These persistent data structures could be corrupted due to storage faults. Practical systems try to safely recover the data and remain available under such failures [16, 18]. However, as we will show, none of the current approaches correctly recover from storage faults, motivating the need for a new approach.

2.3 RSM Recovery Taxonomy

To understand the different possible ways to handle storage faults in RSM systems, we analyze a broad range of approaches. We perform this analysis by two means: first, we analyze practical systems including ZooKeeper, LogCabin, etcd [27], and a Paxos-based system [26] using a fault-injection framework we developed (Section 5); second, we analyze techniques proposed by prior research or used in proprietary systems [16, 18].

Through our analysis, we classify the approaches into two categories: *protocol-oblivious* and *protocol-aware*. The oblivious approaches do not use any protocol-level knowledge to perform recovery. Upon detecting a fault, these approaches take a recovery action locally on the faulty node; such actions interact with the distributed protocols in unsafe ways, leading to data loss. The protocol-aware approaches use some RSM-specific knowledge to recover; however, they do not use this knowledge correctly, leading to undesirable outcomes. Our taxonomy is *not* complete in that there may be other techniques; however, to the best of our knowledge, we have not observed other approaches apart from those in our taxonomy.

To illustrate the problems, we use Figure 1. In all cases, log entries[1] 1, 2, and 3 are committed; losing these items will violate safety. Table 1 shows how each approach behaves in Figure 1's scenarios. As shown in the table, all current approaches lead to safety violation (e.g., data loss), low availability, or both. A recovery mechanism that effectively uses redundancy should be safe and available in all cases. Table 1 also compares the approaches along other axes such as performance, maintenance overhead (intervention and extra nodes), recovery time, and complexity. Although Figure 1 shows only faults in the *log*, the taxonomy applies to other structures including the snapshots and the metainfo.

NoDetection. The simplest reaction to storage faults is none at all: to trust every layer in the storage stack to work reliably. For example, a few prototype Paxos-based systems [26] do not use checksums for their on-disk data; similarly, LogCabin does not protect its snapshots with checksums. *NoDetection* trivially violates safety; corrupted data can be obliviously served to clients. However, deployed systems do use checksums and other integrity strategies for most of their on-disk data.

[1]A log entry contains a state-machine command and data.

Table 1. Recovery Taxonomy

Class	Approach	Safety	Availability	Performance	No Intervention	No extra nodes	Fast Recovery	Low Complexity	(i)	(ii)	(iii)	(iv)	(v)	(vi)
Protocol Oblivious	*NoDetection*	×	√	√	√	√	na	√	E	E	E	E	E	E
	Crash	√	×	√	×	√	na	√	U	C	U	C	U	U
	Truncate	×	√	√	√	√	×	√	C	L	C	L	L	L
	DeleteRebuild	×	√	√	×	√	×	√	C	L	C	L	L	L
Protocol Aware	*MarkNonVoting*	×	×	√	√	√	×	√	U	C	U	C	U	U
	Reconfigure	√	×	√	×	×	×	√	U	C	U	C	U	U
	Byzantine FT	√	×	×	√	×	na	×	U	C	U	U	U	U
	CTRL	√	√	√	√	√	√	√	C	C	C	C	C	C

E- Return Corrupted, L- Data Loss, U- Unavailable, C- Correct.
The table shows how different approaches behave in Figure 1 scenarios. While all approaches are unsafe or unavailable, CTRL ensures safety and high availability.

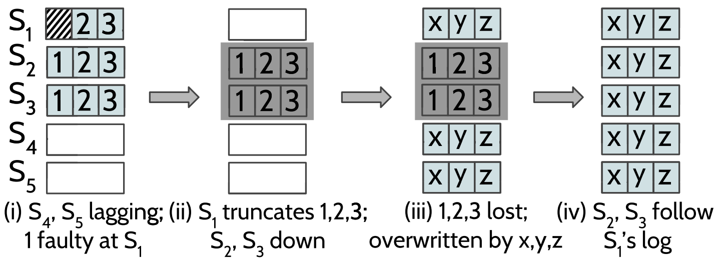

S_1
S_2
S_3
S_4
S_5
1 2 3
x y z
(i) S_4, S_5 lagging; 1 faulty at S_1
(ii) S_1 truncates 1,2,3; S_2, S_3 down
(iii) 1,2,3 lost; overwritten by x,y,z
(iv) S_2, S_3 follow S_1's log

Fig. 2. **Safety Violation Example.** The figure shows the sequence of events which exposes a safety violation with the Truncate approach.

Crash. A better strategy is to use checksums and handle I/O errors, and crash the node on detecting a fault. *Crash* may seem like a good strategy because it intends to prevent any damage that the faulty node may inflict on the system. Our experiments show that the *Crash* approach is common: LogCabin, ZooKeeper, and etcd crash sometimes when their logs are faulty. Also, ZooKeeper crashes when its snapshots are corrupted.

Although *Crash* preserves safety, it suffers from severe unavailability. Given that nodes could be unavailable due to other failures, even a single storage fault results in unavailability, as shown in Figure 1(i). Similarly, a single fault even in different portions of data on a majority (e.g., Figure 1(v)) renders the system unavailable. Note that simply restarting the node does not help; storage faults, unlike other faults, could be persistent: the node will encounter the same fault and crash again until manual intervention, which is error-prone and may cause a data loss. Thus, it is desirable to recover automatically.

Truncate. A more sophisticated action is to truncate (possibly faulty) portions of data and continue operating. The intuition behind *Truncate* is that if the faulty data is discarded, the node can continue to operate (unlike *Crash*), improving availability.

However, we find that *Truncate* can cause a safety violation (data loss). Consider the scenario shown in Figure 2 in which entry 1 is corrupted on S_1; S_4, S_5 are lagging and do not have any entry.

Assume S_2 is the leader. When S_1 reads its log, it detects the corruption; however, S_1 truncates its log, losing the corrupted entry and all subsequent entries (Figure 2(ii)). Meanwhile, S_2 (leader) and S_3 crash. S_1, S_4, and S_5 form a majority and elect S_1 the leader. Now the system does not have any knowledge of committed entries 1, 2, and 3, resulting in a *silent data loss*. The system also commits new entries x, y, and z in the place of 1, 2, and 3 (Figure 2(iii)). Finally, when S_2 and S_3 recover, they follow S_1's log (Figure 2(iv)), completely removing entries 1, 2, and 3.

In summary, although the faulty node detects the corruption, it truncates its log, losing the data locally. When this node forms a majority along with other nodes that are lagging, data is silently lost, violating safety. We find this safety violation in ZooKeeper and LogCabin.

Further, *Truncate* suffers from *inefficient recovery*. For instance, in Figure 1(i), S_1 truncates its log after a fault, losing entries 1, 2, and 3. Now to fix S_1's log, the leader needs to transfer *all* entries, increasing S_1's recovery time and wasting network bandwidth. ZooKeeper and LogCabin suffer from this slow recovery problem.

DeleteRebuild. Another commonly employed action is to manually delete all data on the faulty node and restart the node. Unfortunately, similar to *Truncate*, *DeleteRebuild* can violate safety: a node whose data is deleted could form a majority along with the lagging nodes, leading to a silent data loss. Surprisingly, administrators often use this approach hoping that the faulty node will be "simply fixed" by fetching the data from other nodes [67, 69, 77]. *DeleteRebuild* also suffers from the slow recovery problem similar to *Truncate*.

MarkNonVoting. In this approach, used by a Paxos-based system at Google [18], a faulty node deletes all its data on a fault and marks itself as a non-voting member; the node does not participate in elections until it observes one round of consensus and rebuilds its data from other nodes. By marking a faulty node as nonvoting, safety violations such as the one in Figure 2 are avoided. However, *MarkNonVoting* can sometimes violate safety as noted by prior work [74]. The underlying reason for unsafety is that a corrupted node deletes all its state including the promises[2] given to leaders. Once a faulty node has lost its promise given to a new leader, it could accept an entry from an old leader (after observing a round of consensus on an earlier entry). The new leader, however, still believes that it has the promise from the faulty node and so can overwrite the entry, previously committed by the old leader.

Further, this approach suffers from unavailability. For example, when only a majority of nodes are alive, a single fault can cause unavailability because the faulty node cannot vote; other nodes cannot now elect a leader.

Reconfigure. In this approach, a faulty node is removed and a new node is added. However, to change the configuration, a configuration entry needs to be committed by a majority. Hence, the system remains unavailable in many cases (e.g., when a majority are alive but one node's data is corrupted). Although *Reconfigure* is not used in practical systems to tackle storage faults, it has been suggested by prior research [16, 46].

BFT. An extreme approach is to use a Byzantine-fault-tolerant algorithm which should theoretically tolerate storage faults. However, *BFT* is expensive to be used in practical storage systems; specifically, *BFT* can achieve only half the throughput of what a crash-tolerant protocol can achieve [22]. Moreover, *BFT* requires $3f + 1$ nodes to tolerate f faults [1], thus remaining unavailable in most scenarios in Figure 1.

[2]In Paxos, a promise for a proposal numbered p is a guarantee given by a follower (acceptor) to the leader (proposer) that it will not accept a proposal numbered less than p in the future [43].

Taxonomy Summary. None of the current approaches effectively use redundancy to recover from storage faults. Most approaches do not use any protocol-level knowledge to recover; for example, *Truncate* and *DeleteRebuild* take actions locally on the faulty node and so interact with the distributed protocol in unsafe ways, causing a global data loss. Although some approaches (e.g., *MarkNonVoting*) use some RSM-specific knowledge, they do not do so correctly, causing data loss or unavailability. Thus, to ensure safety and high availability, a recovery approach should effectively use redundancy by exploiting protocol-specific knowledge. Further, it is beneficial to avoid other problems such as manual intervention and slow recovery. Our protocol-aware approach, Ctrl, aims to achieve these goals.

3 CORRUPTION-TOLERANT REPLICATION

Designing a correct recovery mechanism needs a careful understanding of the underlying protocols of the system. For example, the recovery mechanism should be cognizant of how updates are performed on the replicated data and how the leader is elected. We base Ctrl's design on the following protocol-level observations common to most RSM systems.

Leader-Based. A single node acts as the leader; all data updates to the replicated data flow only through the leader.

Epochs. RSM systems partition time into logical units called *epochs*. For any given epoch, only one leader is guaranteed to exist. Every data item is associated with the epoch in which it was appended and its *index* in the log. Since the entries could only be proposed by the leader and only one leader could exist for an epoch, an ⟨*epoch*, *index*⟩ pair uniquely identifies a log entry.

Leader Completeness. A node will not vote for a candidate if it has more up-to-date data than the candidate. Since committed data is present at least in a majority of nodes and a majority vote is required to win the election, the leader is guaranteed to have all the committed data. Although not explicitly specified in some protocols, this property is satisfied by most systems as confirmed by prior research [3, 53].

The above-listed attributes are common to most RSM system implementations. Ctrl exploits these common protocol-level attributes to correctly recover from storage faults. However, Ctrl cannot be readily applied to a few consensus approaches. For example, a few implementations of Paxos [50] allow updates to flow through multiple leaders at the same time. We believe Ctrl can be extended to work with such RSM variants as well. We leave this extension as an avenue for future work.

Ctrl divides the recovery responsibility between two components: the *local storage layer* and the *distributed recovery protocol*; while the storage layer reliably detects faulty data on a node, the distributed protocol recovers the data from redundant copies. Both the components use RSM-specific knowledge to perform their functions.

In this section, we first describe Ctrl's fault model (Section 3.1) and safety and availability guarantees (Section 3.2). We then describe the local storage layer (Section 3.3). Finally, we describe Ctrl's distributed recovery in two parts: first, we show how faulty *logs* are recovered (Section 3.4) and then we explain how faulty *snapshots* are recovered (Section 3.5).

3.1 Fault Model

Our fault model includes the standard failure assumptions made by crash-tolerant RSM systems: nodes could crash at any time and recover later, and nodes could be unreachable due to network failures [22, 44, 53]. Our model adds another realistic failure scenario where persistent data on the individual nodes could be corrupted or inaccessible. Table 2 shows a summary of our storage fault model. Our model includes faults in both user data and the file-system metadata blocks.

Table 2. Storage Fault Model

	Fault Outcome	Possible Causes
Data	corrupted data	misdirected and lost writes in ext
	inaccessible data	LSE, corruptions in ZFS and btrfs
FS Metadata	missing files/directories	directory entry corrupted, fsck may remove a faulty inode
	unopenable files/directories	sanity check fails after inode corruption, permission bits corrupted
	files with more or fewer bytes	*i_size* field in the inode corrupted
	file system read-only	journal corrupted; fsck not run
	file system unmountable	superblock corrupted; fsck not run

The table shows storage faults included in our model and possible causes that lead to a fault outcome.

User data blocks in the files that implement the system's persistent structures could be affected by errors or corruption. A number of (possibly contiguous) data blocks could be faulty as shown by studies [13, 63]. Also, a few bits/bytes of a block could be corrupted. Depending on the local file system in use, corrupted data may be returned obliviously or transformed into errors.

File-system metadata blocks can also be affected by faults; for example, the inode of a log file could be corrupted. Our fault model considers the following outcomes that can be caused by file-system metadata faults: files/directories may go missing, files/directories may be unopenable, a file may appear with fewer or more bytes, the file system may be mounted read-only, and in the worst case, the file system may be unmountable. Some file systems such as ZFS may mask most of the above outcomes from applications [76]; however, our model includes these faulty outcomes because they could realistically occur on other file systems that provide weak protection against corruption (e.g., ext2/3/4). Through fault-injection tests, we have verified that the metadata fault outcomes shown in Table 2 do occur on ext4.

3.2 Safety and Availability Guarantees

Ctrl guarantees that if there exists at least one correct copy of a *committed* data item, it will be recovered or the system will wait for that item to be fixed; committed data will never be lost. In unlikely cases where all copies of a committed item are faulty, the system will correctly remain unavailable. Ctrl also guarantees that the system will make a decision about an *uncommitted* faulty item as early as possible, ensuring high availability.

3.3 CTRL Local Storage Layer

To reliably recover, the storage layer (Clstore) needs to satisfy three key requirements. First, Clstore must be able to reliably detect a storage fault. Second, Clstore must correctly distinguish crashes from corruptions; safety can be violated otherwise. Third, Clstore must identify which pieces of data are faulty; only if Clstore identifies which pieces have been affected, can the distributed protocol recover those pieces.

3.3.1 Persistent Structures Overview. As we discussed, RSM systems maintain three persistent structures: the log, the snapshots, and the metainfo. Clstore uses RSM-specific knowledge of how these structures are used and updated, to perform its functions. For example, Clstore detects faults at a different granularity depending on the RSM data structure: faults in the log are detected at the granularity of individual entries, while faults in the snapshot are detected at the granularity of chunks. Similarly, Clstore uses the RSM-specific knowledge that a log entry is uniquely qualified by its $\langle$*epoch*, *index*$\rangle$ pair to identify faulty log entries.

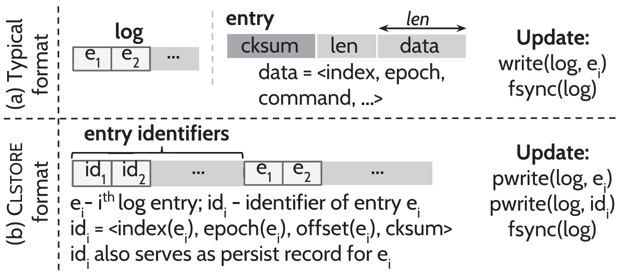


Fig. 3. **Log Format.** (a) shows the format of the log in a typical RSM system and the protocol used to update the log; (b) shows the same for CLSTORE.

Log. The log is a set of files containing a sequence of entries. The format of a typical RSM log is shown in Figure 3(a). The log is updated synchronously in the critical path; hence, changes made to the log format should not affect its update performance. CLSTORE uses a modified format as shown in Figure 3(b) which achieves this goal. A corrupted log is recovered at the granularity of individual entries.

Snapshots. The in-memory state machine is periodically written to a snapshot. Since snapshots can be huge, CLSTORE splits them into chunks; a faulty snapshot is recovered at the granularity of individual chunks.

Metainfo. The metainfo is special in that faulty metainfo *cannot* be recovered from other nodes. This is because the metainfo contains information unique to a node (e.g., its current epoch, votes given to candidates); recovering metainfo obliviously from other nodes could violate safety. CLSTORE uses this knowledge correctly and so maintains two copies of the metainfo locally; if one copy is faulty, the other copy is used. Fortunately, the metainfo is only a few tens of bytes in size and is updated infrequently; therefore, maintaining two copies does not incur significant overheads.

3.3.2 Detecting Faulty Data. CLSTORE uses well-known techniques for detection: *inaccessible* data is detected by catching return codes (e.g., EIO) and *corrupted* data is detected by a checksum mismatch. CLSTORE assumes that if an item and its checksum agree, then the item is not faulty. In the log, each entry is protected by a checksum; similarly, each chunk in a snapshot and the entire metainfo are checksummed.

CLSTORE also handles file-system metadata faults. Missing and unopenable files/directories are detected by handling error codes upon open. Log and metainfo files with fewer or more bytes are detected easily because these files are preallocated and are of a fixed size; snapshot sizes are stored separately, and CLSTORE cross-checks the stored size with the file-system reported size to detect discrepancies. A read-only/unmountable file system is equivalent to a missing data directory. In most cases of file-system metadata faults, CLSTORE crashes the nodes. Crashing reliably on a metadata fault preserves safety but compromises on availability. However, we believe this is an acceptable behavior because there are far more data blocks than metadata blocks; therefore, the probability of faults is significantly less for metadata than data blocks.

3.3.3 Disentangling Crashes and Corruption in Log. An interesting challenge arises when detecting corruptions in the log. A checksum mismatch for a log entry could occur due to two different situations. First, the system could have *crashed* in the middle of an update; in this case, the entry would be partially written and hence cause a mismatch. Second, the entry could be safely persisted but *corrupted* at a later point. Most log-based systems conflate these two cases: they treat a mismatch as a crash [32]. On a mismatch, they discard the corrupted entry and all subsequent entries, losing the data. Discarding entries due to such conflation introduces the possibility of a global data loss (as shown earlier in Figure 2).

Note that if the mismatch were really due to a crash, it is safe to discard the partially written entry. It is safe because the node would not have acknowledged to any external entity that it has written the entry. However, if an entry is *corrupted*, the entry cannot be simply discarded since it could be globally committed. Further, if a mismatch can be correctly attributed to a crash, the faulty entry can be quickly discarded locally, avoiding the distributed recovery. Hence, it is important for the local storage layer to distinguish the two cases.

To denote the completion of an operation, many systems write a commit record [14, 19]. Similarly, CLSTORE writes a persist record, p_i, after writing an entry e_i. For now, assume that e_i is ordered before p_i, i.e., the sequence of steps to append an entry e_i is $write(e_i)$, $fsync()$, $write(p_i)$, $fsync()$. On a checksum mismatch for e_i, if p_i is not present, we can conclude that the system crashed during the update. Conversely, if p_i is present, we can conclude that the mismatch was caused due to a corruption and *not* due to a crash. p_i is checksummed and is very small; it can be atomically written and thus cannot be "corrupted" due to a crash. If p_i is corrupted in addition to e_i, we can conclude that it is a corruption and not a crash.

The above logic works when e_i is ordered before p_i. However, such ordering requires an (additional) expensive *fsync* in the critical path, affecting log-update performance. For this reason, CLSTORE does not order e_i before p_i; thus, the append protocol is $\mathbf{t_1}$:$write(e_i)$, $\mathbf{t_2}$:$write(p_i)$, $\mathbf{t_3}$:$fsync()$.[3] Given this update sequence, assume a checksum mismatch occurs for e_i. If p_i is *not present*, CLSTORE can conclude that it is a crash (before $\mathbf{t_2}$) and discard e_i. Contrarily, if p_i is *present*, there are two possibilities: either e_i could be affected by a corruption after $\mathbf{t_3}$ or a crash could have occurred between $\mathbf{t_2}$ and $\mathbf{t_3}$ in which p_i hit the disk while e_i was only partially written. The second case is possible because file systems can reorder writes between two fsync operations and e_i could span multiple sectors [3, 20, 55, 56]. CLSTORE can still conclude that it is a corruption if e_{i+1} or p_{i+1} is present. However, if e_i is the *last entry*, then we cannot determine whether it was a crash or a corruption. A proof of this claim can be found in Appendix A.

The inability to disentangle the last entry when its persist record is present is not specific to CLSTORE, but rather a fundamental limitation in log-based systems. For instance, in ext4's journal_async_commit mode (where a transaction is not ordered before its commit record), a corrupted last transaction is assumed to be caused due to a crash, possibly losing data [40, 73]. Even if crashes and corruptions can be disentangled, there is little a single-machine system can do to recover the corrupted data. However, in a distributed system, redundant copies can be used to recover. Thus, when the last entry cannot be disentangled, CLSTORE safely marks the entry as *corrupted* and leaves it to the distributed recovery to fix or discard the entry based on the global commitment.

The entanglement problem does not arise for snapshots or metainfo. These files are first written to a temporary file and then atomically renamed. If a crash happens before the rename, the partially written temporary file is discarded. Thus, the system will never see a corrupted snapshot or metainfo due to a crash; if these structures are corrupted, it is because of a storage corruption.

3.3.4 Identifying Faulty Data. Once a faulty item is detected, it has to be *identified*; only if CLSTORE can identify a faulty item, the distributed layer can recover the item. For this purpose, CLSTORE redundantly stores an *identifier* of an item apart from the item itself; duplicating only the identifier instead of the whole item obviates the (2×) storage and performance overhead. However,

[3]The final *fsync* is required for durability.

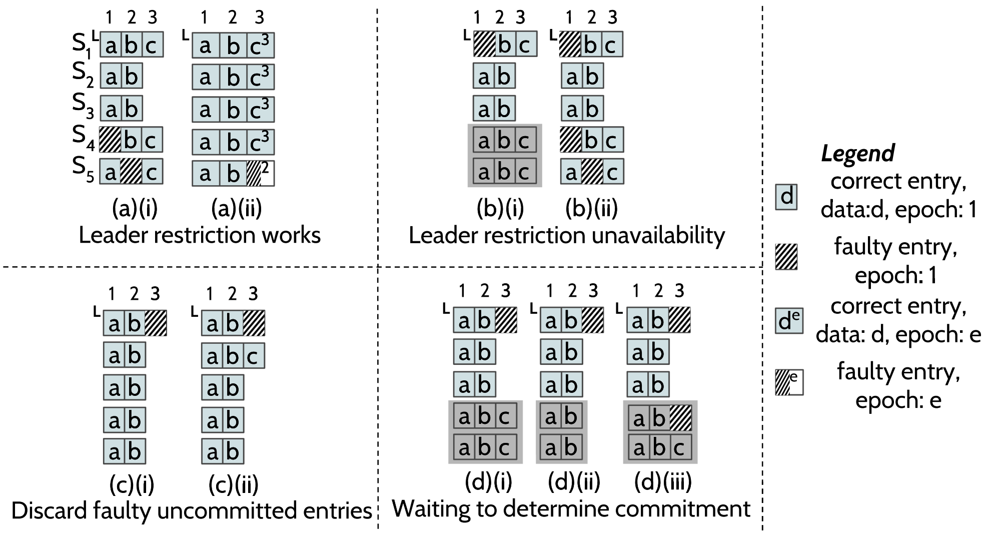

(a)(i)
(a)(ii)
Leader restriction works
(b)(i)
(b)(ii)
Leader restriction unavailability
(c)(i)
(c)(ii)
Discard faulty uncommitted entries
(d)(i)
(d)(ii)
(d)(iii)
Waiting to determine commitment
Legend
correct entry, data:d, epoch: 1
faulty entry, epoch: 1
correct entry, data: d, epoch: e
faulty entry, epoch: e

Fig. 4. **Distributed Log Recovery.** The figure shows how CTRL's log recovery operates. All entries are appended in epoch 1 unless explicitly mentioned. For entries appended in other epochs, the epoch number is shown in the superscript. Entries shown as striped boxes are faulty. A gray box around a node denotes that it is down or extremely slow. The leader is marked with L on the left. Log indexes are shown at the top.

storing the identifier near the item is less useful; a misdirected write can corrupt both the item and its identifier [10, 11]. Hence, identifiers are physically separated from the items they identify.

The ⟨*epoch*, *index*⟩ pair serves as the identifier for a log entry and is stored separately at the head of the log, as shown in Figure 3(b). The offset of an entry is also stored as part of the identifier to enable traversal of subsequent entries on a fault. The identifier of a log entry also conveniently serves as its persist record. Similarly, for a snapshot chunk, the ⟨*snap-index*, *chunk#*⟩ pair serves as the identifier; the *snap-index* and the snapshot size are stored in a separate file than the snapshot file. The identifiers have a nominal storage overhead (32 bytes for log entries and 12 bytes for snapshots), can be atomically written, and are also protected by a checksum.

It is highly unlikely an item and its identifier will both be faulty since they are physically separated [10, 11, 13, 63]. In such unlikely and unfortunate cases, CLSTORE crashes the node to preserve safety. Table 3 (second column) summarizes CLSTORE's key techniques.

3.4 CTRL Distributed Log Recovery

The local storage layer detects faulty data items and passes on their identifiers to the distributed recovery layer. We now describe how the distributed layer recovers the identified faulty items from redundant copies using RSM-specific knowledge. We first describe how *log entries* are recovered and subsequently describe *snapshot* recovery. As we discussed, metainfo files are recovered locally and so we do not discuss them any further. We use Figure 4 to illustrate how log recovery works.

Naive Approach: Leader Restriction. RSM systems do not allow a node with an incomplete log to become the leader. A naive approach to recovering from storage faults could be to impose an additional constraint on the election: *a node cannot be elected the leader if its log contains a faulty entry*. The intuition behind the naive approach is as follows: since the leader is guaranteed to have all committed data and our new restriction ensures that the leader is not faulty, faulty log entries on other nodes could be fixed using the corresponding entries on the leader. Cases (a)(i) and (a)(ii) in Figure 4 show scenarios where the naive approach could elect a leader. In (a)(i), only S_1 can become the leader because other nodes are either lagging or have at least one faulty entry. Assume S_1 is the leader also in case (a)(ii).

Fixing Followers' Logs. When the leader has no faulty entries, fixing the followers is straightforward. For example, in case (a)(i), the followers inform S_1 of their faulty entries; S_1 then supplies the correct entries. However, sometimes the leader might not have any knowledge of an entry that a follower is querying for. For instance, in case (a)(ii), S_5 has a faulty entry at index 3 but with a *different epoch*. This situation is possible because S_5 could have been the leader for epoch 2 and crashed immediately after appending an entry. As discussed earlier, an entry is uniquely identified by its ⟨*epoch*, *index*⟩; thus, when querying for faulty entries, a node needs to specify the epoch of the entry in addition to its index. Thus, S_5 informs the leader that its entry ⟨*epoch*:2, *index*:3⟩ is faulty. However, S_1 does not have such an entry in its log. If the leader does not have an entry that the follower has, then the entry *must be an uncommitted entry* because the leader is guaranteed to have all committed data; thus, the leader instructs S_5 to truncate the faulty entry and also replicates the correct entry.

Although the naive approach guarantees safety, it has availability problems. The system will be unavailable in cases such as the ones shown in (b): a leader cannot be elected because the logs of the alive nodes are either faulty or lagging. Note that even a single storage fault can cause an unavailability as shown in (b)(i). It is possible for a carefully designed recovery protocol to provide better availability in these cases. Specifically, since at least one intact copy of all committed entries exists, it is possible to collectively reconstruct the log.

3.4.1 *Removing the Restriction Safely.*

To recover from scenarios such as those in Figure 4(b), we remove the additional constraint on the election. Specifically, any node that has a more up-to-date log can now be elected the leader even if it has faulty entries. This relaxation improves availability; however, two key questions arise: first, when can the faulty leader proceed to accept new commands? Second, and more importantly, is it safe to elect a faulty node as the leader?

To accept a new command, the leader has to append the command to its log, replicate it, and apply it to the state machine. However, before applying the new command, *all* previous commands must be applied. Specifically, faulty commands cannot be skipped and later applied when they are fixed; such out-of-order application would violate safety. Hence, it is required for the leader to fix its faulty entries before it can accept new commands. Thus, for improved availability, the leader needs to fix its faulty entries as early as possible.

The crucial part of the recovery to ensure safety is to fix the leader's log using the redundant copies on the followers. In simple cases such as (b)(i) and (b)(ii), the leader S_1 could fix its faulty entry ⟨*epoch*:1, *index*:1⟩ using the correct entries from the followers and proceed to normal operation. However, in several scenarios, the leader cannot immediately recover its faulty entries; for example, none of the reachable followers might have any knowledge of the entry to be recovered or the entry to be recovered could also be faulty on the followers.

3.4.2 *Determining Commitment.*

The main insight to fix the leader's faulty log safely and quickly is to distinguish *uncommitted* entries from *possibly committed* ones; while recovering the committed entries is necessary for safety, uncommitted entries can be safely discarded. Further, discarding uncommitted faulty entries immediately is crucial for availability. For instance, in case (c)(i), the faulty entry on S_1 cannot be fixed since there are no copies of it; waiting to fix that entry results in indefinite unavailability. Sometimes, an entry could be partially replicated but remain uncommitted; for example, in case (c)(ii), the faulty entry on S_1 is partially replicated but is not committed. Although there is a possibility of recovering this entry from the other node (S_2), this is *not* necessary for safety; it is completely safe for the leader to discard this uncommitted entry.

To determine the commitment of a faulty entry, the leader queries the followers. If a majority of the followers respond that they do *not* have the entry (negative acknowledgment), then the leader concludes that the entry is uncommitted. In this case, the leader safely discards that entry and all subsequent entries; it is safe to discard the subsequent entries because entries are committed only in order (i.e., if an entry at index i is an uncommitted entry, then all entries after i must also be uncommitted entries). Conversely, if the entry were committed, at least one node in this majority would have that entry and inform the leader of it; in this case, the leader can fix its faulty entry using that response.

Waiting to Determine Commitment. Sometimes, it may be impossible for the leader to quickly determine commitment. For instance, consider the cases in Figure 4(d) in which S_4 and S_5 are down or slow. S_1 queries the followers to recover its entry ⟨*epoch*:1, *index*:3⟩. S_2 and S_3 respond that they do not have such an entry (negative acknowledgment). S_4 and S_5 do not respond because they are down or slow. The leader, in this case, has to wait for either S_4 or S_5 to respond; discarding the entry without waiting for S_4 or S_5 could violate safety. However, once S_4 or S_5 responds, the leader will make a decision immediately. In (d)(i), S_4 or S_5 would respond with the correct entry, fixing the leader. In (d)(ii), S_4 or S_5 would respond that it does not have the entry, accumulating three (a majority out of five) negative acknowledgments; hence, the leader can conclude that the entry is uncommitted, discard it, and continue to normal operation. In (d)(iii), S_4 would respond that it has the entry but is faulty in its log too. In this case, the leader has to wait for the response from S_5 to determine commitment. In the unfortunate and unlikely case where all copies of an entry are faulty, the system will remain unavailable.

3.4.3 The Complete Log Recovery Protocol. We now assemble the pieces of the log recovery protocol. First, fixing faulty followers is straightforward; the committed faulty entries on the followers can be eventually fixed by the leader because the leader is guaranteed to have all committed data. Faulty entries on followers that the leader does not know about are uncommitted; hence, the leader instructs the followers to discard such entries.

The main challenge is thus fixing the leader's log. The leader queries the followers to recover its entry ⟨*epoch*:e, *index*:i⟩. Three types of responses are possible:

Response 1: **have** – a follower could respond that it has the entry ⟨*epoch*:e, *index*:i⟩ and is not faulty in its log.
Response 2: **dontHave** – a follower could respond that it does not have the entry ⟨*epoch*:e, *index*:i⟩ in its log.
Response 3: **haveFaulty** – a follower could respond that it has ⟨*epoch*:e, *index*:i⟩ but is faulty in its log too.

Once the leader collects these responses, it takes the following possible actions:

Case 1: if it gets a *have* response from at least one follower, it fixes the entry in its log.
Case 2: if it gets a *dontHave* response from a majority of followers, it confirms that the entry is uncommitted, discards that entry and all subsequent entries.
Case 3: if it gets a *haveFaulty* response from a follower, it waits for either Case 1 or Case 2 to happen.

Case 1 and Case 2 can happen in any order; both orderings are safe. Specifically, if the leader decides to discard the faulty entry (after collecting a majority *dontHave* responses), it is safe since the entry was uncommitted anyways. Conversely, there is no harm in accepting a correct entry (at least one *have* response) and replicating it. The first to happen out of these two cases will take precedence over the other.

The leader proceeds to normal operation only after its faulty data is discarded or recovered. However, Ctrl discards uncommitted data as early as possible and minimizes the recovery latency by recovering faulty data at a fine granularity (as we show in Section 5.2), ensuring that the leader proceeds to normal operation quickly.

The leader could crash or be partitioned while recovering its log. On a leader failure, the followers will elect a new leader and make progress. The partial repair done by the failed leader is harmless: it could have either fixed committed faulty entries or discarded uncommitted ones, both of which are safe.

3.5 CTRL Distributed Snapshot Recovery

Because the logs can grow indefinitely, periodically, the in-memory state machine is written to disk and the logs are garbage collected. Current systems including ZooKeeper and LogCabin do not handle faulty snapshots correctly (Section 2.3): they either crash or load corrupted snapshots obliviously. Ctrl aims to recover faulty snapshots from redundant copies. Snapshot recovery is different from log recovery in that all data in a snapshot is committed and already applied to the state machine; hence, faulty snapshots cannot be discarded in any case (unlike uncommitted log entries which can be discarded safely).

3.5.1 Leader-Initiated Identical Snapshots. Current systems [45] have two properties with respect to snapshots. First, they allow new commands to be applied to the state machine while a snapshot is in progress. Second, they take index-consistent snapshots: a snapshot S_i represents the state machine in which log entries exactly up to i have been applied. One of the mechanisms used in current systems to realize the above two properties is to take snapshots in a `fork`-ed child process; while the child can write an index-consistent image to the disk, the parent can keep applying new commands to its copy of the state machine. Ctrl should enable snapshot recovery while preserving the above two properties.

In current systems, every node runs the snapshot procedure independently, taking snapshots at different log indexes. Because the snapshots are taken at different indexes, snapshot recovery can be complex: a faulty snapshot on one node cannot be simply fetched from other nodes. Further, snapshots cannot be recovered at the granularity of chunks because they will be byte-wise non-identical; entire snapshots have to be transferred across nodes, slowing down recovery.

This complexity can be significantly alleviated if the nodes take the snapshot at the same index; identical snapshots also enable chunk-based recovery.

However, coordinating a snapshot operation across nodes can, in general, affect the common-case performance. For example, one naive way to realize identical snapshots is for the leader to produce the snapshot, insert it into the log as yet another entry, and replicate it. However, such an approach will affect update performance since the snapshot could be huge and all client commands must wait while the snapshot commits [52]. Moreover, transferring the snapshot to the followers wastes network bandwidth.

Ctrl takes a different approach to identical snapshots that preserves common-case performance. The leader initiates the snapshot procedure by first deciding the index at which a snapshot will be taken and informing the followers of the index. Once a majority agree on the index, all nodes independently take a snapshot at the index. When the leader learns that a majority (including itself) have taken a snapshot at an index i, it garbage collects its log up to i and instructs the followers to do the same.

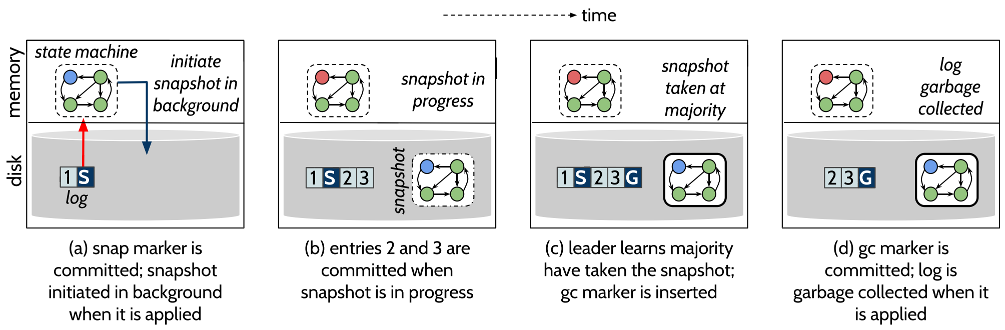

time
memory
disk
state machine
initiate snapshot in background
1 S
log
snapshot in progress
1 S 2 3
snapshot
snapshot taken at majority
1 S 2 3 G
log garbage collected
2 3 G
(a) snap marker is committed; snapshot initiated in background when it is applied
(b) entries 2 and 3 are committed when snapshot is in progress
(c) leader learns majority have taken the snapshot; gc marker is inserted
(d) gc marker is committed; log is garbage collected when it is applied

Fig. 5. **Leader-Initiated Identical Snapshots.** The figure shows how leader-initiated identical snapshots is implemented in Ctrl. The figure only shows the various states of the leader; the followers' states are not shown. Time flows from left to right. (a) At first, the leader decides to take a snapshot after entry 1; hence, it inserts the snap marker (denoted by S). When the snap marker commits, and consequently when the nodes apply the marker, they take a snapshot at that moment. As shown, the snapshot operation is initiated and performed in the background. (b) While the nodes take the snapshot in the background, the leader commits entries 2 and 3 and so the in-memory state machine moves to a different state. (c) When the leader learns that a majority of nodes have taken the snapshot, it inserts the gc marker (denoted by G). (d) Finally, when the gc marker is applied, the nodes garbage collect the log entries that are part of the persisted snapshot.

Ctrl implements the above procedure using the log. When the leader decides to take a snapshot, it inserts a special marker called snap into the log. When the snap marker commits, and thus when a node applies the marker to the state machine, it takes a snapshot (i.e., the snapshot corresponds to the state where commands exactly up to the marker have been applied). Within each node, we reuse the same mechanism used by the original system (e.g., a `fork`-ed child) to allow new commands to be applied while a snapshot is in progress. Notice that the snapshot operation happens independently on all nodes but the operation will produce identical snapshots because the marker will be seen at the same log index by all nodes when it is committed. When the leader learns that a majority of nodes (including itself) have taken a snapshot at an index i, it appends another marker called gc for i; when the gc marker is committed and applied, the nodes garbage collect their log entries up to i. Figure 5 illustrates how leader-initiated identical snapshots works.

3.5.2 Recovering Snapshot Chunks. With the identical-snapshot mechanism, snapshot recovery becomes easier. Once a faulty snapshot is detected, the local storage layer provides the distributed protocol the snapshot index and the chunk that is faulty. The distributed protocol recovers the faulty chunk from other nodes. First, the leader recovers its faulty chunks from the followers and then fixes the faulty snapshots on followers. Three cases arise during snapshot recovery.

First, the log entries for a faulty snapshot may not be garbage collected yet; in this case, the snapshot is recovered locally from the log (after fixing the log if needed).

Second, if the log is garbage collected, then a faulty snapshot has to be recovered from other nodes. However, if the log entries for a snapshot are garbage collected, then at least a majority of the nodes must have taken the same snapshot. This is true because the gc marker is inserted only after a majority of nodes have taken the snapshot. Thus, faulty garbage-collected snapshots are recovered from those redundant copies.

Third, sometimes, the leader may not know a snapshot that a follower is querying for (e.g., if a follower took a snapshot and went offline for a long time and the leader replaced that snapshot with an advanced one); in this case, the leader supplies the full advanced snapshot.

3.6 CTRL Summary

Ctrl's storage layer detects faulty data using checksums and handling errors. It also disentangles crashes and corruptions in the log. Finally, it identifies which portions of the data are faulty and passes on the identifiers to the distributed recovery layer.

The distributed protocol recovers the faulty data from the redundant copies. Figure 6 summarizes the distributed recovery protocol. Ctrl decouples the recovery of followers from that of the leader. In all cases, fixing the followers is straightforward: the leader supplies the correct data because the leader is guaranteed to have all the committed data. Ctrl couples the fixing of followers with common-case operations such as replication of entries. Actions taken by the leader and the followers to fix the followers' data are shown in boxes C and D of Figure 6. The leader can fix its faulty snapshots from its local log if the log is not garbage collected yet. If the log is garbage collected, the leader recovers the snapshot from the followers (a majority of nodes are guaranteed to have the snapshot). The leader fixes its log by determining commitment of the faulty entries. Actions taken by the leader and the followers during leader recovery are shown in boxes A and B of Figure 6.

Ctrl's storage and distributed recovery layers exploit RSM-specific knowledge to perform their functions. Table 3 shows a summary of techniques employed in both the layers.

4 IMPLEMENTATION

We implement Ctrl in two different RSM systems, LogCabin (v1.0) and ZooKeeper (v3.4.8); while LogCabin is based on Raft, ZooKeeper is based on ZAB. Implementing Ctrl's storage layer and distributed recovery took only a moderate developer effort; Ctrl adds about 1,500 lines of code to each of the base systems.

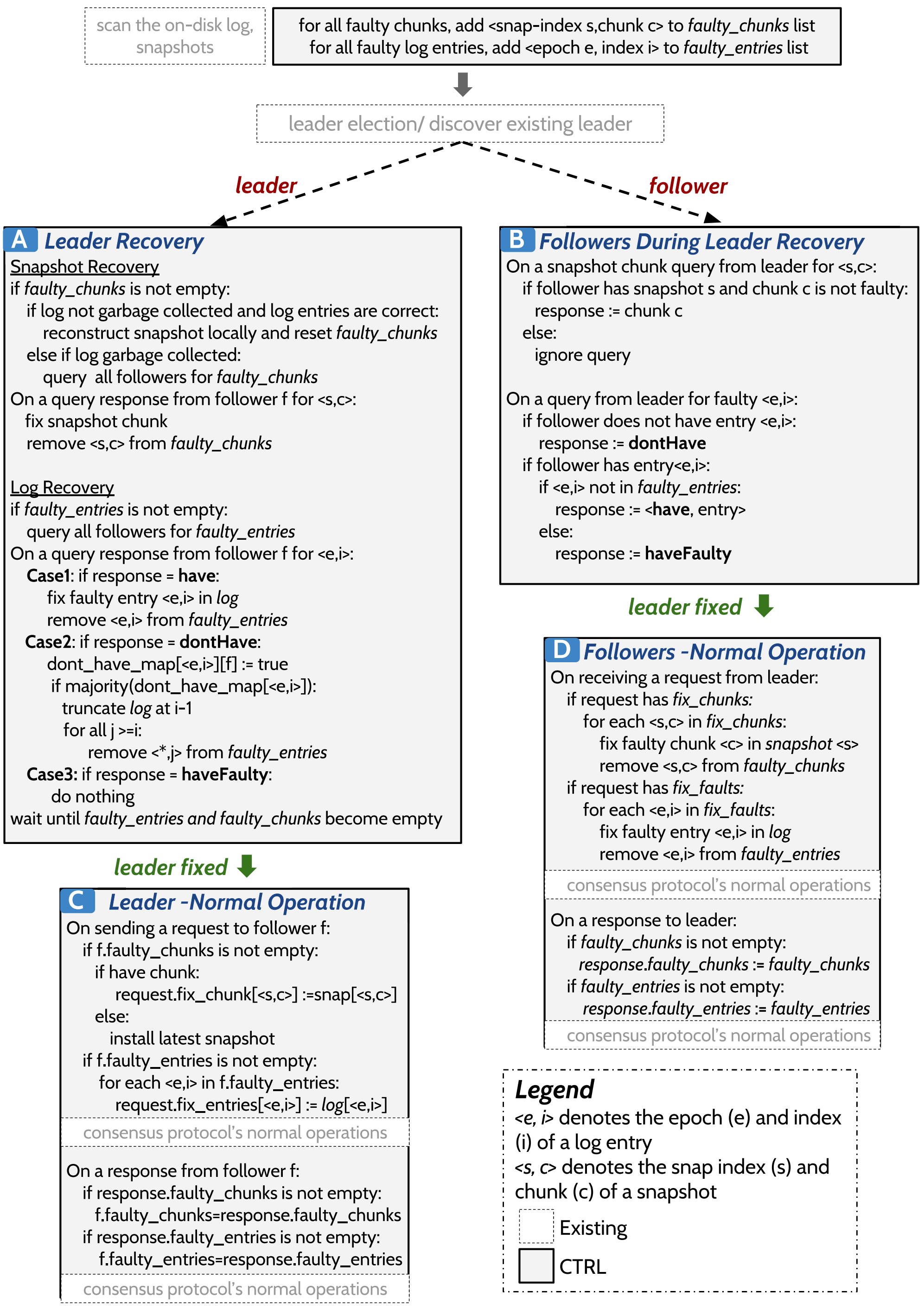
4.1 Local Storage Layer

We implemented Clstore by modifying the storage engines of LogCabin and ZooKeeper. In both systems, the log is a set of files, each of a fixed size and preallocated with zeros. The header of each file is reserved for the log-entry identifiers. The size of the reserved header is proportional to the file size. Clstore ensures that a log entry and its identifier are at least a few megabytes physically apart. Both systems batch many log entries to improve update performance. With batching, Clstore performs crash-corruption disentanglement as follows: the first faulty entry without an identifier and its subsequent entries are discarded; faulty entries preceding that point are marked as corrupted and passed on to the distributed layer.

In both systems, the state machine is a data tree. We modified both the systems to take index-consistent identical snapshots: when a `snap` marker is applied, the state machine (i.e., the tree) is serialized to the disk. The *snap-index* and snapshot size are stored separately. Clstore uses a chunk size of 4K, enabling fine-grained recovery.

In LogCabin, the metainfo contains the `currentTerm` and `votedFor` structures. Similarly, in ZooKeeper, structures such as `acceptedEpoch` and `currentEpoch` constitute the metainfo. Clstore stores redundant copies of metainfo and protects them using checksums.

Log entries, snapshot chunks, and metainfo are protected by a CRC32 checksum. Clstore detects inaccessible data items by catching errors (`EIO`); it then populates the item's in-memory buffer with zeros, causing a checksum mismatch. Thus, Clstore deals with both corruptions and errors as checksum mismatches. Lost log writes result in checksum mismatches because the log is preallocated with zeros. Misdirected writes can overwrite previously written log entries. Such misdirected writes typically occur at the block or sector granularity, causing a checksum mismatch for the log entries in most cases. In rare cases, the entries could be block aligned, and a misdirected write may



scan the on-disk log, snapshots
leader election/ discover existing leader
leader
follower
leader fixed
leader fixed
consensus protocol's normal operations
Legend
Existing
CTRL

```
for all faulty chunks, add <snap-index s,chunk c> to faulty_chunks list
for all faulty log entries, add <epoch e, index i> to faulty_entries list
```

A Leader Recovery

```
Snapshot Recovery
if faulty_chunks is not empty:
   if log not garbage collected and log entries are correct:
      reconstruct snapshot locally and reset faulty_chunks
   else if log garbage collected:
      query  all followers for faulty_chunks
On a query response from follower f for <s,c>:
   fix snapshot chunk
   remove <s,c> from faulty_chunks

Log Recovery
if faulty_entries is not empty:
   query all followers for faulty_entries
On a query response from follower f for <e,i>:
   Case1: if response = have:
      fix faulty entry <e,i> in log
      remove <e,i> from faulty_entries
   Case2: if response = dontHave:
      dont_have_map[<e,i>][f] := true
      if majority(dont_have_map[<e,i>]):
         truncate log at i-1
         for all j >=i:
            remove <*,j> from faulty_entries
   Case3: if response = haveFaulty:
      do nothing
wait until faulty_entries and faulty_chunks become empty
```

B Followers During Leader Recovery

```
On a snapshot chunk query from leader for <s,c>:
   if follower has snapshot s and chunk c is not faulty:
      response := chunk c
   else:
      ignore query

On a query from leader for faulty <e,i>:
   if follower does not have entry <e,i>:
      response := dontHave
   if follower has entry<e,i>:
      if <e,i> not in faulty_entries:
         response := <have, entry>
      else:
         response := haveFaulty
```

C Leader -Normal Operation

```
On sending a request to follower f:
   if f.faulty_chunks is not empty:
      if have chunk:
         request.fix_chunk[<s,c>] :=snap[<s,c>]
      else:
         install latest snapshot
   if f.faulty_entries is not empty:
      for each <e,i> in f.faulty_entries:
         request.fix_entries[<e,i>] := log[<e,i>]
consensus protocol's normal operations
On a response from follower f:
   if response.faulty_chunks is not empty:
      f.faulty_chunks=response.faulty_chunks
   if response.faulty_entries is not empty:
      f.faulty_entries=response.faulty_entries
consensus protocol's normal operations
```

D Followers -Normal Operation

```
On receiving a request from leader:
   if request has fix_chunks:
      for each <s,c> in fix_chunks:
         fix faulty chunk <c> in snapshot <s>
         remove <s,c> from faulty_chunks
   if request has fix_faults:
      for each <e,i> in fix_faults:
         fix faulty entry <e,i> in log
         remove <e,i> from faulty_entries
consensus protocol's normal operations
On a response to leader:
   if faulty_chunks is not empty:
      response.faulty_chunks := faulty_chunks
   if faulty_entries is not empty:
      response.faulty_entries := faulty_entries
consensus protocol's normal operations
```

Legend
<e, i> denotes the epoch (e) and index (i) of a log entry
<s, c> denotes the snap index (s) and chunk (c) of a snapshot

Fig. 6. **Ctrl Recovery Protocol Summary.** The figure shows the summary of the protocol. Ctrl's recovery code is shown in thick boxes and original consensus operations are shown in dashed boxes.

Table 3. Techniques Summary

	Local Storage	**Distributed Recovery**
Log	granularity: entry identifier:⟨*epoch*, *index*⟩ crash-corruption disentanglement	global-commitment determination to fix leader, leader fixes followers
Snapshot	granularity: chunk identifier:⟨*snap-index*, *chunk#*⟩ no entanglement	leader-initiated identical snapshots, chunk-based recovery
Metainfo	granularity: file identifier: n/a no entanglement	none (only internal redundancy)

The table shows a summary of techniques employed by Ctrl's storage layer and distributed recovery.

not cause a checksum mismatch. However, the storage layer catches such cases through a sanity check that verifies that the index of the log entries are in order and are monotonically increasing.

4.2 Distributed Recovery

LogCabin. In Raft, *terms* are equivalent to epochs. Thus, a log entry is uniquely identified by its ⟨*term*, *index*⟩ pair. To fix the followers, we modified the AppendEntries RPC used by the leader to replicate entries [53]. The followers inform the leader of their faulty log entries and snapshot chunks in the responses of this RPC; the leader sends the correct entries and chunks in a subsequent RPC. A follower starts applying commands to its state machine once its faulty data is fixed. To fix the leader, we added a new RPC which the leader issues to the followers. The leader does not proceed to normal operation until its faulty data is fixed. After a configurable recovery timeout, the leader steps down if it is unable to recover its faulty data (e.g., due to a partition), allowing other nodes to become the leader. Several entries and chunks are batched in a single request/response, avoiding multiple round trips.

ZooKeeper. In ZAB, the epoch and index are packed into the *zxid* which uniquely identifies a log entry [6]. Followers discover and connect to the leader in Phase 1. We modified Phase 1 to send information about the followers' faulty data. The followers are synchronized with the leader in Phase 2. We modified Phase 2 so that the leader sends the correct data to the followers. The leader waits to hear from a majority during Phase 1 after which it sends a `newEpoch` message; we modified this message to send information about the leader's faulty data. The leader does not proceed to Phase 2 until its data is fixed.

5 EVALUATION

We evaluate the correctness and performance of Ctrl versions of LogCabin and ZooKeeper. We conducted our performance experiments on a three-node cluster on a 1-Gb network; each node is a 40-core Intel Xeon CPU E5-2660 machine with 128GB memory running Linux 3.13, with a 500-GB SSD and a 1-TB HDD managed by ext4.

5.1 Correctness

To verify Ctrl's safety and availability guarantees, we built a fault-injection framework that can inject storage faults (targeted corruptions and random block corruptions and errors). The framework can also inject crashes. By injecting crashes at different points in time, the framework simulates lagging nodes. After injecting faults, we issue reads from clients to determine whether the target system remains available and preserves safety.

Table 4. Targeted Corruptions

System	Recovery Scenario	Total Test Cases	Original Approach	Original Outcomes: Unavailable	Original Outcomes: Unsafe	Original Outcomes: Correct	CTRL Outcomes: Unavailable	CTRL Outcomes: Unsafe	CTRL Outcomes: Correct
LogCabin	Possible	2401	truncate	0	2,355	46	0	0	2,401
			crash	2,355	0	46	0	0	2,401
	Not possible	1695	truncate	0	1,695	0	1,695	0	0
			crash	1,695	0	0	1,695	0	0
ZooKeeper	Possible	2401	truncate	0	2,355	46	0	0	2,401
			crash	2,355	0	46	0	0	2,401
	Not possible	1695	truncate	0	1,695	0	1,695	0	0
			crash	1,695	0	0	1,695	0	0

The table shows results for targeted corruptions in log; we trigger two policies (truncate and crash) in the original systems. Recovery is possible when at least one intact copy exists; recovery is not possible when no intact copy exists.

We first exercise different log-recovery scenarios. Then, we test snapshot recovery, and finally file-system metadata fault recovery.

5.1.1 Log Recovery. We perform three different experiments to test log-recovery: targeted log-entry corruptions, random block corruptions and errors, and faults with crashed and lagging nodes.

Targeted Corruptions. We initialize the cluster by inserting four log entries and ensuring that the entries are replicated to all three nodes in the cluster. We exercise all combinations of entry corruptions across the three nodes ($(2^4)^3 = 4096$ combinations). Out of the 4,096 cases, a correct recovery is possible in 2,401 cases (at least one non-faulty copy of each entry exists). In the remaining 1,695 cases, recovery is not possible because one or more entries are corrupted on *all* the nodes. We inject targeted corruptions into two different sets of on-disk structures. In the first set, on a corruption, the original systems invoke the *truncate* action (i.e., they truncate faulty data and continue). In the second set, the original systems invoke the *crash* action (i.e., node crashes on detection). For example, while ZooKeeper *truncates* when the tail of a transaction is corrupted, it *crashes* the node if the transaction header is corrupted. Ctrl always recovers the corrupted data from other replicas.

Table 4 shows the results. When recovery is possible, the original systems recover only in 46/2,401 cases. In those 46 cases, no node or only one node is corrupted. In the remaining 2,355 cases, the original systems are either unsafe (for *truncate*) or unavailable (for *crash*). In contrast, Ctrl correctly recovers in all 2,401 cases. When a recovery is not possible (all copies corrupted), the original systems are either unsafe or unavailable in all cases. Ctrl, by design, correctly remains unavailable since continuing would violate safety.

Table 5. Log Recovery

System	Experiment	Total Test Cases	Outcomes					
			Original			CTRL		
			Unavailable	Unsafe	Correct	Unavailable	Unsafe	Correct
LogCabin	Corruptions	5,000	738	793	3,469	0	0	5,000
	Errors	5,000	2,497	0	2,503	0	0	5,000
ZooKeeper	Corruptions	5,000	807	656	3,537	0	0	5,000
	Errors	5,000	2,469	0	2,531	0	0	5,000

(a) Random Block Corruptions and Errors

System	Total Test Cases	Outcomes					
		Original			CTRL		
		Unavailable	Unsafe	Correct	Unavailable	Unsafe	Correct
LogCabin	5,000	4,194	141	665	0	0	5,000
ZooKeeper	5,000	1,306	1,806	1,888	0	0	5,000

(b) Corruptions with Lagging Nodes

(a) shows results for random block corruptions and errors in the log. (b) shows results for random corruptions in the log with crashed and lagging nodes.

Random Block Corruptions and Errors. We initialize the cluster by replicating a few entries to all nodes. We first choose a random set of nodes. In each such node, we then corrupt a randomly selected file-system block (from the files implementing the log). We repeat this process, producing 5,000 test cases. We similarly inject block errors. Since we inject a fault into a block, several entries and their checksums within the block will be faulty.

Table 5(a) shows the results. For *block corruptions*, original LogCabin is unsafe or unavailable in about 30% ((738 + 793)/5,000) of cases. Similarly, original ZooKeeper is incorrect in about 30% of cases. On a *block error*, original LogCabin and ZooKeeper simply crash the node, leading to unavailability in about 50% of cases. In contrast, Ctrl correctly recovers in all cases.

Faults with Crashed and Lagging Nodes. In the previous experiments, all entries were committed and present on all nodes. In this experiment, we inject crashes at different points on a random set of nodes while inserting entries. Thus, in the resultant log states, nodes could be lagging, entries could be uncommitted, and have different epochs on different nodes for the same log index.

$\langle S_1 : [a^1, \_, \_], S_2 : [b^2, c^3, \_], S_3 : [b^2, \_, \_]\rangle$ is an example state where S_1 appends a at index 1 in epoch 1 (shown in superscript) and crashes, S_2 appends b at index 1 in epoch 2, replicates to S_3, then S_2, S_3 crash and recover, S_2 appends c in epoch 3 and crashes. From each such state, we corrupt different entries, generating 5,000 test cases. For example, from the above state, we corrupt a on $S1$ and b, c on S_2. If S_2 is elected the leader, S_2 needs to fix b from S_3 (since b is committed), discard

Table 6. Snapshot and FS Metadata Faults

System	Total Test Cases	Outcomes					
		Original			CTRL		
		Unavailable	Unsafe	Correct	Unavailable	Unsafe	Correct
LogCabin	1,000	297	257	446	0	0	1,000
ZooKeeper	1,000	417	200	383	0	0	1,000

(a) Snapshot Recovery

System	Total Test Cases	Outcomes					
		Original			CTRL		
		Unavailable	Unsafe	Correct	Unavailable	Unsafe	Correct
LogCabin	1,000	405	36	559	434	0	566
ZooKeeper	1,000	329	192	479	502	0	498

(b) FS Metadata Faults

(a) and (b) show How Ctrl recovers from snapshot and FS metadata faults, respectively.

c (*c* is uncommitted and cannot be recovered), and also instruct $S1$ to discard *a* (*a* is uncommitted) and replicate correct entry *b*. As shown in Table 5(b), Ctrl correctly recovers from all such cases, while the original versions are unsafe or unavailable in many cases.

Model Checking. We also model checked Ctrl's log recovery since it involves many corner cases, using a Python-based model that we developed. We explored over 2.5M log states all of in which Ctrl correctly recovered. Also, when key decisions are tweaked, the checker finds a violation immediately: for example, the leader concludes that a faulty entry is uncommitted only after gathering $\lfloor N/2 \rfloor + 1$ *dontHave* responses; if this number is reduced, then the checker finds a safety violation. We have also added the specification of Ctrl's log recovery to the TLA+ specification of Raft [25] and confirmed that it correctly recovers from corruptions, while the original specification violates safety.

5.1.2 *Snapshot Recovery.* In this experiment, we insert a few entries and trigger the nodes to take a snapshot. We crash the nodes at different points, producing three possible states for each node: l, t, and g, where l is a state where the node has only the log (it has not taken a snapshot), t is a snapshot for which garbage collection has not been performed yet, and g is a snapshot which has been garbage collected. We produce all possible combinations of states across three nodes. On each such state, we randomly pick a set of nodes to inject faults, and corrupt a random combination of snapshots and log entries, generating 1,000 test cases. For example, $\langle S_1 : t, S_2 : g, S_3 : l \rangle$ is a base state on which we corrupt snapshot t and a few preceding log entries on S_1 and g on S_2. In such a state, if S_1 becomes the leader, it has to fix its log from S_3, then has to locally recover its t snapshot, after which it has to fix g on S_2. S_1 also needs to install the snapshot on S_3.

As shown in Table 6(a), Ctrl correctly recovers from all such cases. Original LogCabin is incorrect in about half of the cases because it obliviously loads faulty snapshots sometimes and crashes sometimes. Original ZooKeeper crashes the node if it is unable to locally construct the data from the snapshot and the log, leading to unavailability; unsafety results because a faulty log is truncated in some cases.

5.1.3 *File-System Metadata Faults.* To test how Ctrl recovers from file-system metadata faults, we corrupt file-system metadata structures (such as inodes and directory blocks) resulting in unopenable files, missing files, and files with fewer or more bytes. We inject such faults in a randomly chosen file on one or two nodes at a time, creating 1,000 test cases. Table 6(b) shows the results. In some cases, the faulty nodes in original versions crash because of a failed deserialization or assertion. However, sometimes original LogCabin and ZooKeeper do not detect the fault and continue

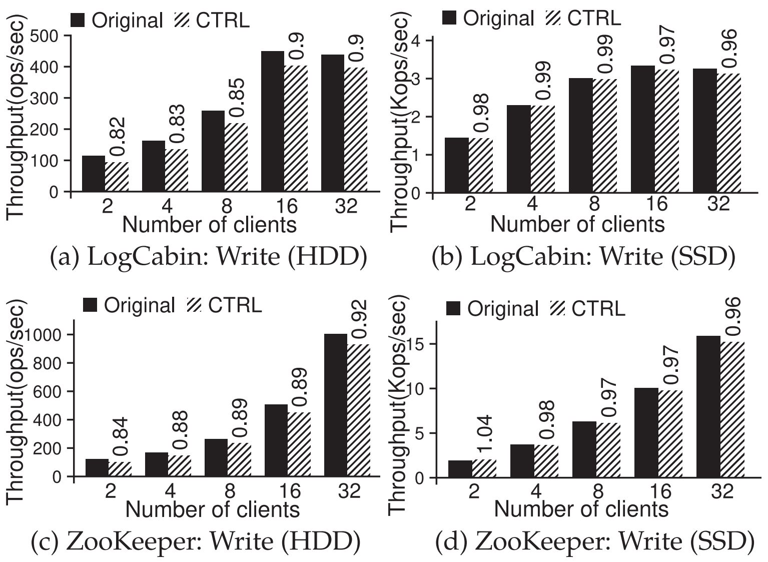


Fig. 7. **Write Performance.** (a) and (c) show the write throughput in original and Ctrl versions of LogCabin and ZooKeeper on an HDD. (b) and (d) show the same for SSD. The number on top of each bar shows the performance of Ctrl normalized to that of original.

operating, violating safety in 36 and 192 cases, respectively. In contrast, Ctrl reliably crashes the node on a file-system metadata fault, preserving safety always.

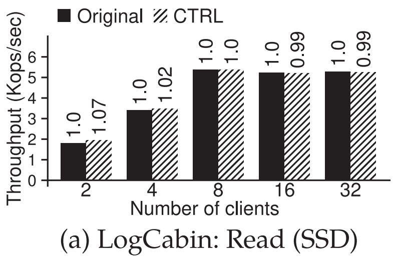
5.2 Performance

We now compare the common-case performance of the Ctrl versions against the original versions. During writes, the entries are first written to the on-disk log; snapshots are taken periodically in the background. Both LogCabin and ZooKeeper batch several log entries to improve write throughput. In addition to the above steps, Ctrl writes an identifier for each log entry at the head of the log. First, we run a write-only workload that exposes the worst-case overheads (caused by the additional writes) introduced by Ctrl. The workload runs for 300 seconds, inserting entries each of size 1K. Numbers reported are the average over five runs.

Figure 7(a) and (c) show the throughput on an HDD for varying number of clients in LogCabin and ZooKeeper, respectively. Ctrl writes the identifiers in a physically separate location compared to that of the entries; this separation induces a seek on disks in the update path. However, the seek cost is amortized when more requests are batched; Ctrl has an overhead of 8%–10% for 32 clients on disks. Figure 7(b) and (d) show throughput on an SSD; Ctrl adds very minimal overhead on SSDs (4% in the worst case). Note that this workload performs only writes and therefore shows Ctrl's overheads in the worst case.

In both LogCabin and ZooKeeper, reads are served from memory; thus, the read paths should ideally not be affected by Ctrl. To confirm this, we run a read-only workload. Figure 8(a) and (b) show the throughput on an SSD for varying number of clients in LogCabin and ZooKeeper, respectively. As shown in the figure, Ctrl does not introduce any overheads and maintains the same performance as the original systems. We see similar results on HDDs (not shown in the figure).

Fast Log Recovery. We now show how Ctrl can recover a faulty log quickly. To show the potential reduction in log-recovery time, we insert 30K log entries (each of size 1K) and corrupt the



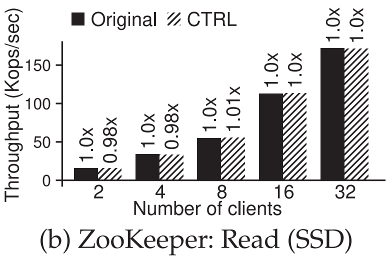


Fig. 8. **Read Performance.** (a) and (b) show the read throughput in original and Ctrl versions of LogCabin and ZooKeeper on a SSD. The number on top of each bar shows the performance of Ctrl normalized to that of original.

first entry on one node. In original LogCabin, the faulty node detects the corruption but truncates *all* entries; hence, the leader transfers all entries to bring the node up-to-date. Ctrl fixes only the faulty entry, reducing recovery time. The faulty node is fixed in 1.24 seconds (32MB transferred) in the original system, while Ctrl takes only 1.2 ms (7KB transferred). We see a similar reduction in log-recovery time in ZooKeeper.

6 PROTOCOL-AWARE RECOVERY FOR OTHER CLASSES OF SYSTEMS

So far, we have described Ctrl, a protocol-aware recovery mechanism specific to RSM systems. We now discuss how a protocol-aware recovery approach can be adapted to other classes of systems; we consider two classes in our discussion: primary backup and Dynamo-style quorums. Our previous work [31, 32] shows that, in the presence of storage faults, many systems in these two classes can lose data, spread corruption, or become unavailable. A Par-based approach to handling storage faults can improve the reliability of these systems.

Primary-Backup. The primary-backup class includes systems such as Redis [58] and Kafka [4]. The individual systems vary in the degree of how well they handle storage faults. For example, Redis entirely relies on the local file system to protect the data against storage faults; consequently, it does not employ checksums or handle storage errors. Fixing such a system would require several changes to its storage layer. In contrast, Kafka already uses checksums and handles errors carefully; however, it does not use protocol-specific knowledge to recover from storage faults correctly. We thus use Kafka as an example to describe how Par can be applied to a primary backup system.

Kafka implements a distributed persistent *message* queue in which clients can publish and subscribe to different *message topics*. The nodes persistently store streams of messages in a log. Kafka and most primary-backup systems are similar to RSM systems in many aspects. For instance, all message updates flow through a primary (similar to an RSM leader), which then replicates the messages to a set of backup servers (similar to RSM followers). During a message append, the Kafka leader replicates the message to a number of nodes specified in the *min.insync.replicas* option. A message is considered committed if the message is persisted successfully on at least a *min.insync.replicas* number of replicas. In its default (and strongly consistent) configuration, in which *dirty election* is disabled, Kafka ensures the leader completeness property by only allowing a node which is part of the *in-sync* replicas (i.e., nodes that contain all committed data) to become the leader.

Given that the messages are grouped under different topics, each message is uniquely identified by combination of its *topic_id* and *message_id*; the local storage layer can redundantly store the ⟨*topic_id*, *message_id*⟩ pair and use it as the message identifier. During distributed recovery, the leader queries the followers (using the identifiers) to fix its faulty messages. The leader can determine the commitment of a message as follows. If ($N - min.insync.replicas$), where N is the replication factor, nodes reply that they do not have the message, the leader can conclude that the message is uncommitted and discard such a message. Committed messages can be fixed from any one correct response from the followers. Similar to RSM systems, follower recovery is simple; the followers follow the leader's log and simply fix their faulty messages by querying the leader.

Dynamo-Style Quorums. Next, we consider Dynamo-style quorum-based systems such as Cassandra [24]. Cassandra and other dynamo-style quorum systems are significantly different from RSM and primary-backup systems: they do not have a dedicated leader or a set of followers. Instead, the nodes form a ring in a circular keyspace. In Cassandra, data is stored in column families; a column family contains multiple rows, and each row is uniquely identified by a primary key K. A write request for a key K specifies the write quorum value, W: the update needs to be persisted on W nodes before acknowledging the client. Similarly, a read request for K specifies the read quorum value, R: the read will be served after contacting R nodes. Each request (based on the key it accesses) is assigned to a coordinator; the coordinator is responsible for assigning a timestamp for the request and replicating the update to W nodes specified in the request. If a coordinator has failed, then the next node in the ring acts as the coordinator for the request. The system supports different read and write consistency levels (i.e., values of W and R) for each request.

The primary key K and the timestamp associated with the write request that modified K acts as the identifier, which the local storage layer can redundantly store. Because there is no dedicated leader, each faulty node has to fix its data by querying the other nodes in the system. During distributed recovery, a node with a corrupted data item associated with key K queries the other nodes responsible for K. A faulty node has to wait for at least $N - W + 1$ other nodes to determine commitment, where N is the replication factor, and W is the write quorum with which K was updated. If a node responds with the correct data for K and the timestamp, the faulty node can fix its data.

7 RELATED WORK

Our work builds upon prior work from two bodies of research: fault-injection studies and tolerating practical faults in distributed systems. Our analysis of how RSM-based systems react to storage faults (Section 2.3) draws inspiration from many past fault-injection studies. Our design of Ctrl (Section 3) builds upon several recent efforts on tolerating practical faults other than crashes in distributed systems.

Storage Faults. Several studies have shown the prevalence of storage faults in hard disk drives [63] and SSDs [36, 48, 51, 64]. Further, studies have shown that cheap and near-line storage devices are more likely to be affected by storage faults [10, 11]. Given that many distributed deployments tend to use cheap storage hardware [23, 33], analyzing the effects of storage faults is important. These prior studies motivated us to analyze how RSM systems behave in the presence of storage faults.

Our previous work [31, 32] studied how many popular distributed systems behave in the presence of storage faults. This work discovered fundamental reasons why distributed systems are not resilient to storage faults. At a high level, similar to our analysis in Section 2.3, the prior work also found that distributed systems do not effectively use redundancy to recover from storage faults. However, the study did not uncover any safety or availability violations reported in Section 2.3; this is because the fault model in our previous study considers injecting only storage faults (precisely, a single storage fault on a single node at a time). In contrast, our fault model in this work considers crashes and network failures in addition to storage faults, exposing previously unknown safety and availability violations in RSM systems.

Targeted Approaches. Prior research describes two approaches [16, 18] to tackle storage faults in RSM systems. However, these approaches suffer from unavailability. Furthermore, the

MarkNonVoting approach [18] can violate safety because important metainfo such as promises can be lost on a storage fault [74]. CTRL avoids such safety violations by storing two copies of metainfo on each node. Approaches that improve the reliability of other specific systems have also been proposed [72, 75].

Generic Approaches. Many generic approaches to handling practical faults other than crashes have been proposed. PASC [22] hardens systems to tolerate corruptions by maintaining two copies of the entire state on each node and assumes that both the copies will not be faulty at the same time. This approach does not work well for storage faults; having two copies of on-disk state incurs 2× space overhead. Furthermore, in most cases, PASC crashes the node on a fault, causing unavailability. XFT [44] is designed to tolerate non-crash faults. However, it can tolerate only a total of $\lfloor (N-1)/2 \rfloor$ crash and non-crash faults. Similarly, UpRight [21] has an upper bound on the total faults to remain safe and available.

CTRL differs from the generic approaches through its special focus on storage faults. This focus brings two main advantages. First, CTRL attributes faults at a fine granularity: while the generic approaches consider a node as faulty if any of its data is corrupted, CTRL considers faults at the granularity of individual data items. Second, because of such fine-granular fault treatment, CTRL can be available as long as a majority of nodes are up and at least one non-faulty copy of a data item exists even though portions of data on *all* nodes could be corrupted. CTRL cannot tolerate arbitrary non-crash faults [42] (e.g., memory errors). However, CTRL can augment the generic approaches: for example, a system can be hardened against memory faults using PASC while making it robust to storage faults using CTRL.

8 CONCLUSIONS

Recovering from storage faults in distributed systems is surprisingly hard. We introduce *protocol-aware recovery* (PAR), a new approach that exploits protocol-specific knowledge of the underlying distributed system to correctly recover from storage faults. We design CTRL, a protocol-aware recovery approach for RSM systems. We experimentally show that CTRL correctly recovers from a range of storage faults with little performance overhead.

Our work is only a first step in hardening distributed systems to storage faults: while we have successfully applied the PAR approach to RSM systems, other classes of systems (e.g., primary-backup, Dynamo-style quorums) still remain vulnerable. We believe the PAR approach can be applied to such classes as well. We hope our work will lead to more work on building reliable distributed storage systems that are robust to storage faults.

ACKNOWLEDGMENTS

We thank Mahesh Balakrishnan (our FAST'18 shepherd), the anonymous FAST'18 reviewers, and the members of ADSL for their excellent feedback and valuable suggestions. We also thank CloudLab [60] for providing a great environment to run our experiments.

APPENDIX A IMPOSSIBILITY OF LAST-ENTRY DISENTANGLEMENT

In any log-based storage system, if the last entry in the log is corrupted, it is impossible to determine whether the corruption was due to a crash or a disk corruption (as we discussed in Section 3.3). We now present a proof of this claim; we first define the various elements necessary for the proof and then present the proof.

Log. We model the log L as two disjoint lists: one list L_e that stores entries and one list L_{id} that stores identifiers.

Identifiers. The identifier of a log entry contains vital information about that entry; this information helps CTRL's distributed protocol to recover corrupted entries from copies on other nodes.

Operations. Two kinds of operations update the log:

— $write(v)$, which updates L_e or L_{id} (depending on if v is an entry or identifier).
— $fsync()$ flushes all previous writes to disk.

Sequences. A *disentangled sequence* of transactions $\sigma = t_1, \ldots, t_n$, where $n > 1$ is one where each t_i is a subsequence of three operations: a_i^1, a_i^2, a_i^3, where

— a_i^1 is of the form $write(e_i)$,
— a_i^2 is of the form $write(id_i)$,
— a_i^3 is of the form $fsync()$,

where e_i is the entry to be written and id_i is its respective identifier. For simplicity, we assume a single log.

Log Appends. Suppose we are given a disentangled sequence $\sigma = t_1, \ldots, t_n$. We use L^I to denote the initial state of the log. We use σL^I to denote the state of the log after executing the sequence σ beginning from state L^I.

Corruption and Crash. We distinguish two *bad* events: corruptions co and crashes cr.

— A corruption co_i changes element e_i in L_e to some new e_i' where $e_i' \neq e_i$.
— We assume identifiers (id_i) cannot be affected by a corruption.
— We assume the identifiers can be atomically written to the disk because an identifier is much smaller than a single sector (i.e., $write(id_i)$ is atomic).
— We assume a crash cr_i can only happen between a_i^2 and a_i^3, that is, right before the $fsync$, for a sequence $t_1, \ldots, t_n$, as defined above.

Given sequence σ, we use σ_{cr_i} to denote σ with a crash in t_i. If the system crashes during t_i, then no entries t_j would appear in the log for any $j > i$. Given σ, we use σ_{co_i} to denote σ with a corruption event co_i appended at the end.

THEOREM A.1 (DISENTANGLEMENT). *Suppose we are given the disentangled sequence σ and log L.*

— **Case 1:** *Let $L^1 = \sigma_{\mathrm{cr}_n} L^I$, and let $L^2 = \sigma_{\mathrm{co}_n} L^I$. Suppose we are provided L^I, σ, and one of the logs L^1 and L^2. We cannot detect whether σ_{cr_n} or σ_{co_n} is the one that executed resulting in L^1 or L^2.*
— **Case 2:** *Let $L^{\mathrm{co}_i} = \sigma_{\mathrm{co}_i} L^I$, where $i \in [1, n)$. Provided L^I, σ, and L^{co_i}, we can conclude that σ_{cr_j} did not execute, where $j \in [1, n]$.*

PROOF. First, we note that by being *able to detect* whether a crash or corruption happened, we mean that there exists a deterministic algorithm that will return whether a crash or corruption happened.

Case 1: We prove the first case with a simple construction. Let $\sigma = t_1$, where

$$t_1 = write(e_1), write(id_1), fsync().$$

Let L^I be the empty log. Let $L^1 = \sigma_{cr_1} L^I$ and $L^2 = \sigma_{co_1} L^I$.

Assume that when the crash cr_1 happened, only a strict subset of e_1 was written in addition to id_1. Let the strict subset of e_1 that was written be e_1'. The above condition can arise because $write(e_1)$ need not be atomic and writes can be reordered by the underlying file system on a crash. Now, assume that the corruption co_1 turns e_1 to e_1'.

We can now prove the first case by contradiction: Suppose there is an algorithm M that can take (i) the initial state of the log, (ii) the current state of the log, and (iii) the sequence of transactions σ that lead to the current state (minus co and cr events), and deterministically returns whether a crash or corruption happened. In the above example, $L^1 = L^2$ by construction. So, $M(L^I, L^1, \sigma) = M(L^I, L^2, \sigma)$. Therefore, no such M exists.

Case 2: Fix i, j as in theorem statement. Let $L^{cr_j} = \sigma_{cr_j} L^I$. Assume $L^{cr_j} = L^{co_i}$. If $j \neq i$, then entry e_i cannot be affected by the crash, and therefore the $L^{cr_j} \neq L^{co_i}$. If $j = i$, since $i < n$, then e_i is fixed by recovery. Therefore, $L^{cr_j} \neq L^{co_i}$. □

REFERENCES

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. 2006. Byzantine disk paxos: Optimal resilience with byzantine shared memory. *Distributed Computing* 18, 5 (2006), 387–408.

[2] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2018. Protocol-aware recovery for consensus-based storage. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST'18)*.

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

[4] Apache. 2017. Kakfa. Retrieved April 21, 2017 from http://kafka.apache.org/.

[5] Apache. 2008. ZooKeeper. Retrieved April 21, 2017 from https://zookeeper.apache.org/.

[6] Apache. 2008. ZooKeeper Guarantees, Properties, and Definitions. Retrieved April 21, 2017 from https://zookeeper.apache.org/doc/r3.2.2/zookeeperInternals.html#sc_guaranteesPropertiesDefinitions.

[7] Apache Cassandra. 2017. Cassandra Replication. Retrieved April 21, 2017 from http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html.

[8] Apache ZooKeeper. 2014. Applications and Organizations using ZooKeeper. Retrieved April 21, 2017 from https://cwiki.apache.org/confluence/display/ZOOKEEPER/PoweredBy.

[9] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2015. *Operating Systems: Three Easy Pieces* (0.91 ed.). Arpaci-Dusseau Books.

[10] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. 2008. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST'08)*.

[11] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. An analysis of latent sector errors in disk drives. In *Proceedings of the 2007 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*.

[12] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. 2008. Analyzing the effects of disk-pointer corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*.

[13] Lakshmi Narayanan Bairavasundaram. 2008. *Characteristics, Impact, and Tolerance of Partial Disk Failures*. Ph.D. dissertation. University of Wisconsin, Madison.

[14] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobber, Michael Wei, and John D. Davis. 2012. CORFU: A shared log design for flash clusters. In *Proceedings of the 9th Symposium on Networked Systems Design and Implementation (NSDI'12)*.

[15] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. 1982. Grapevine: An exercise in distributed computing. *Communications of the ACM* 25, 4 (April 1982), 260–274. http://dx.doi.org/10.1145/358468.358487

[16] William J. Bolosky, Dexter Bradshaw, Randolph B. Haagens, Norbert P. Kusters, and Peng Li. 2011. Paxos replicated state machines as the basis of a high-performance data store. In *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI'11)*.

[17] Mike Burrows. 2006. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06)*.

[18] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. 2007. Paxos made live: An engineering perspective. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing*.

[19] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. Optimistic crash consistency. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.

[20] Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2012. Consistency without ordering. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*.

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

[22] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. 2012. Practical hardening of crash-tolerant systems. In *2012 USENIX Annual Technical Conference (USENIX ATC'12)*.

[23] Jeff Dean. 2010. Building Large-Scale Internet Services. Retrieved April 21, 2017 from http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf.

[24] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*.

[25] Diego Ongaro. 2014. Raft TLA+ Specification. Retrieved April 21, 2017 from https://github.com/ongardie/raft.tla.

[26] epaxos. 2012. epaxos Source Code. Retrieved April 21, 2017 from https://github.com/efficient/epaxos.

[27] etcd. 2014. etcd. Retrieved April 21, 2017 from https://coreos.com/etcd.

[28] etcd. 2014. etcd: Production Users. Retrieved April 21, 2017 from https://coreos.com/etcd/docs/latest/production-users.html.

[29] Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. 2014. Checking the integrity of transactional mechanisms. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST'14)*.

[30] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. 2012. Recon: Verifying file system consistency at runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST'12)*.

[31] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy does not imply fault tolerance: Analysis of distributed storage reactions to file-system faults. *ACM Transactions on Storage* 13, 3 (Sept. 2017), 20:1–20:33.

[32] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. 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)*.

[33] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*.

[34] Matthias Grawinkel, Thorsten Schafer, Andre Brinkmann, Jens Hagemeyer, and Mario Porrmann. 2011. Evaluation of applied intra-disk redundancy schemes to improve single disk reliability. In *Proceedings of the 19th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.

[35] Kevin M. Greenan, Darrell D. E. Long, Ethan L. Miller, Thomas Schwarz, and Avani Wildani. 2009. Building flexible, fault-tolerant flash-based storage systems. In *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep'09)*.

[36] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. 2009. Characterizing flash memory: Anomalies, observations, and applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'09)*.

[37] James Hamilton. 2007. On designing and deploying internet-scale services. In *Proceedings of the 21st Annual Large Installation System Administration Conference (LISA'07)*.

[38] James Myers. 2014. Data Integrity in Solid State Drives. Retrieved April 21, 2017 from http://intel.ly/2cF0dTT.

[39] John Goerzen. 2017. Silent Data Corruption Is Real. Retrieved April 21, 2017 from http://changelog.complete.org/archives/9769-silent-data-corruption-is-real.

[40] Jonathan Corbet. 2008. Responding to ext4 Journal Corruption. Retrieved April 21, 2017 from https://lwn.net/Articles/284037/.

[41] Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini. 2011. Zab: High-performance broadcast for primary-backup systems. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'11)*.

[42] Dmitrii Kuvaiskii, Rasha Faqeh, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2016. HAFT: Hardware-assisted fault tolerance. In *Proceedings of the EuroSys Conference (EuroSys'16)*.

[43] Leslie Lamport. 2001. Paxos made simple. *ACM SIGACT News* 32, 4 (2001), 18–25.

[44] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical fault tolerance beyond crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[45] LogCabin. 2014. LogCabin. Retrieved April 21, 2017 from https://github.com/logcabin/logcabin.

[46] Jacob R. Lorch, Atul Adya, William J. Bolosky, Ronnie Chaiken, John R. Douceur, and Jon Howell. 2006. The SMART way to migrate replicated stateful services. In *Proceedings of the EuroSys Conference (EuroSys'06)*.

[47] Parisa Jalili Marandi, Christos Gkantsidis, Flavio Junqueira, and Dushyanth Narayanan. 2016. Filo: Consolidated consensus as a cloud service. In *2016 USENIX Annual Technical Conference (USENIX ATC'16)*.
[48] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. A large-scale study of flash memory failures in the field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'15)*.
[49] MongoDB. 2017. MongoDB Replication. Retrieved April 21, 2017 from https://docs.mongodb.org/manual/replication/.
[50] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP'13)*.
[51] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD failures in datacenters: What? When? and Why?. In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR'16)*.
[52] Diego Ongaro. 2014. *Consensus: Bridging Theory and Practice*. Ph.D. dissertation. Stanford University.
[53] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*.
[54] Bernd Panzer-Steindel. 2007. Data integrity. *CERN/IT*.
[55] Thanumalayan Sankaranarayana Pillai, Ramnatthan Alagappan, Lanyue Lu, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Application crash consistency and performance with CCFS. In *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST'17)*.
[56] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. 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)*.
[57] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON file systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*.
[58] Redis. 2015. Redis. Retrieved April 21, 2017 from http://redis.io/.
[59] Redis. 2015. Redis Replication. Retrieved April 21, 2017 from http://redis.io/topics/replication.
[60] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:* 39, 6 (2014).
[61] Robert Harris. 2007. Data Corruption Is Worse than You Know. Retrieved April 21, 2017 from http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/.
[62] Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys* 22, 4 (Dec. 1990), 299–319. http://dx.doi.org/10.1145/98163.98167
[63] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. 2010. Understanding latent sector errors and how to protect against them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST'10)*.
[64] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash reliability in production: The expected and the unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST'16)*.
[65] Michael D. Schroeder, Andrew D. Birrell, and Roger M. Needham. 1984. Experience with grapevine: The growth of a distributed system. *ACM Transactions on Computer Systems* 2, 1 (Feb. 1984), 3–23. http://dx.doi.org/10.1145/2080.2081
[66] Thomas Schwarz, Ahmed Amer, Thomas Kroeger, Ethan L. Miller, Darrell D. E. Long, and Jehan-François Pâris. 2016. RESAR: Reliable storage at exabyte scale. In *Proceedings of the 24th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*.
[67] Romain Slootmaekers and Nicolas Trangez. 2012. Arakoon: A distributed consistent key-value store. In *SIGPLAN OCaml Users and Developers Workshop*, Vol. 62.
[68] Stackoverflow. 2015. Can ext4 Detect Corrupted File Contents? Retrieved April 21, 2017 from http://stackoverflow.com/questions/31345097/can-ext4-detect-corrupted-file-contents.
[69] Stackoverflow. 2013. ZooKeeper Clear State. Retrieved April 21, 2017 from http://stackoverflow.com/questions/17038957/org-apache-hadoop-hbase-pleaseholdexception-master-is-initializing.
[70] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. 2003. Improving the reliability of commodity operating systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*.
[71] D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP'95)*.
[72] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. 2013. HARDFS: Hardening HDFS with selective and lightweight versioning. In *Proceedings of the 11th Conference on File and Storage Technologies (FAST'13)*.

[73] Theodore Ts'o. 2008. What to Do when the Journal Checksum is Incorrect. Retrieved April 21, 2017 from https://lwn.net/Articles/284038/.

[74] Robbert Van Renesse, Nicolas Schiper, and Fred B. Schneider. 2015. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing* 12, 4 (2015), 472–484.

[75] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the Salus scalable block store. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI'13)*.

[76] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. End-to-end data integrity for file systems: A ZFS case study. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST'10)*.

[77] ZooKeeper Jira Issues. 2012. Unable to Load Database on Disk when Restarting after Node Freeze. Retrieved April 21, 2017 from https://issues.apache.org/jira/browse/ZOOKEEPER-1546.

Received May 2018; accepted July 2018