Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to File-System Faults

AISHWARYA GANESAN, RAMNATTHAN ALAGAPPAN,
ANDREA C. ARPACI-DUSSEAU, and REMZI H. ARPACI-DUSSEAU,
University of Wisconsin—Madison

We analyze how modern distributed storage systems behave in the presence of file-system faults such as data corruption and read and write errors. We characterize eight popular distributed storage systems and uncover numerous problems related to file-system fault tolerance. We find that modern distributed systems do not consistently use redundancy to recover from file-system faults: a single file-system fault can cause catastrophic outcomes such as data loss, corruption, and unavailability. We also find that the above outcomes arise due to fundamental problems in file-system fault handling that are common across many systems. Our results have implications for the design of next-generation fault-tolerant distributed and cloud storage systems.

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: File-system faults, data corruption, fault tolerance

ACM Reference format:
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 Trans. Storage* 13, 3, Article 20 (September 2017), 33 pages.
https://doi.org/10.1145/3125497

1 INTRODUCTION

Cloud-based applications such as Internet search, photo and video services [19, 65, 67], social networking [91, 94], transportation services [92, 93], and e-commerce [53] depend on *modern distributed storage systems* to manage their data. This important class of systems includes key-value stores (e.g., Redis), configuration stores (e.g., ZooKeeper, LogCabin), document stores (e.g.,

This material was supported by funding from NSF grants CNS-1419199, CNS-1421033, CNS-1319405, and CNS-1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, 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 '17 article by Ganesan et al. [29]. The additional material here includes a behavior analysis of a few systems (Redis, Cassandra, and Kafka) in the presence of bit corruptions, a study of Cassandra in a different configuration, more graphics depicting key on-disk data structures of the various systems, more thorough description of the observations across systems with figures to aid in understanding, a summary of the results, figures to illustrate our fault injection methodology, and many other small edits and updates.
Authors' addresses: A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, 1210 W. Dayton St., Madison, WI 53706; emails: {ag, ra, dusseau, remzi}@cs.wisc.edu.

https://doi.org/10.1145/3125497

MongoDB), column stores (e.g., Cassandra), messaging queues (e.g., Kafka), and databases (e.g., RethinkDB, CockroachDB).

Modern distributed storage systems store data in a replicated fashion for improved reliability. Each replica works atop a commodity local file system on commodity hardware to store and manage critical user data. In most cases, replication can mask failures such as system crashes, power failures, and disk or network failures [22, 24, 31, 32, 41, 81]. Unfortunately, storage devices such as disks and flash drives exhibit a more complex failure model in which certain blocks of data can become inaccessible (read and write errors) [7, 9, 49, 55, 80, 82] or, worse, data can be silently corrupted [8, 60, 86]. These complex failures are known as partial storage faults [63].

Previous studies [10, 63, 99] have shown how partial storage faults are handled by file systems such as ext3, NTFS, and ZFS. File systems, in some cases, simply propagate the faults as-is to applications; for example, ext4 returns corrupted data as-is to applications if the underlying device block is corrupted. In other cases, file systems react to the fault and transform it into a different one before passing onto applications; for example, btrfs transforms an underlying block corruption into a read error. In either case, we refer to the faults thrown by the file system to its applications as *file-system faults*.

The behavior of modern distributed storage systems in response to file-system faults is critical and strongly affects cloud-based services. Despite this importance, little is known about how modern distributed storage systems react to file-system faults.

A common and widespread expectation is that redundancy in higher layers (i.e., across replicas) enables recovery from local file-system faults [12, 22, 36, 42, 82]. For example, an inaccessible block of data in one node of a distributed storage system would ideally *not* result in a user-visible data loss, because the same data are redundantly stored on many nodes. Given this expectation, in this article, we answer the following questions: *How do modern distributed storage systems behave in the presence of local file-system faults? Do they use redundancy to recover from a single file-system fault?*

To study how modern distributed storage systems react to local file-system faults, we build a fault injection framework called CORDS that includes the following key pieces: *errfs*, a user-level FUSE file system that systematically injects file-system faults, and *errbench*, a suite of system-specific workloads that drives systems to interact with their local storage. For each injected fault, CORDS automatically observes resultant system behavior. We studied eight widely used systems using CORDS: Redis [66], ZooKeeper [6], Cassandra [4], Kafka [5], RethinkDB [70], MongoDB [52], LogCabin [46], and CockroachDB [14].

The most important overarching lesson from our study is this: A single file-system fault can induce catastrophic outcomes in most modern distributed storage systems. Despite the presence of checksums, redundancy, and other resiliency methods prevalent in distributed storage, a single file-system fault can lead to data loss, corruption, unavailability, and, in some cases, the spread of corruption to other intact replicas.

The benefits of our systematic study are twofold. First, our study has helped us characterize file-system fault handling behaviors of eight systems and also uncover numerous bugs in these widely used systems. We find that these systems can silently return corrupted data to users, lose data, propagate corrupted data to intact replicas, become unavailable, or return an unexpected error on queries. For example, a single write error during log initialization can cause write unavailability in ZooKeeper. Similarly, corrupted data in one node in Redis and Cassandra can be propagated to other intact replicas. In Kafka and RethinkDB, corruption in one node can cause a user-visible data loss. Because distributed storage systems inherently store redundant copies of data and we inject only one fault at a time, these behaviors are surprising and undesirable.

Second, our study has enabled us to make several observations across all systems concerning file-system fault handling. We find that the above undesirable outcomes arise due to the following fundamental root causes in file-system fault tolerance that are common to many distributed storage systems.

Systems employ diverse data-integrity strategies. We find that systems employ diverse strategies to protect against file-system faults; while some systems carefully use checksums, others completely trust lower layers in the stack to detect and handle corruption.

Faults are often undetected locally. We find that faults are often locally undetected. Sometimes, such locally undetected faults lead to immediate harmful global effects.

Crashing is the most common reaction. Even when systems reliably detect faults, in most cases, they simply crash instead of using redundancy to recover from the fault.

Redundancy is underutilized. Although distributed storage systems replicate data and functionality across many nodes, a single file-system fault on a single node can result in harmful clusterwide effects; surprisingly, many distributed storage systems do not consistently use redundancy as a source of recovery.

Crash and corruption handling are entangled. Systems often conflate recovering from a crash with recovering from corruption, accidentally invoking the wrong recovery subsystem to handle the fault, and ultimately leading to poor outcomes such as data loss.

Local fault handling and global protocols interact in unsafe ways. Local fault-handling behaviors and global distributed protocols, such as read repair, leader election, and resynchronization, sometimes interact in an unsafe manner, leading to propagation of corruption to intact replicas or data loss.

This work makes three major contributions. First, we build a fault injection framework (CORDS) to carefully inject file-system faults into applications (Section 3). Second, we present a behavioral study of eight widely used modern distributed storage systems on how they react to file-system faults and also uncover numerous bugs in these storage systems (Section 4.1). We have contacted developers of seven systems and five of them have acknowledged the problems we found. While a few problems can be tolerated by implementation-level fixes, tolerating many others require fundamental design changes. Third, we derive a set of fundamental observations across all systems showing some of the common data integrity and file-system fault handling problems (Section 4.2). Our testing framework and bugs we reported are publicly available [1]. We hope that our results will lead to discussions and future research to improve the resiliency of next generation cloud storage systems.

The rest of the article is organized as follows. First, we provide a background on file-system faults and motivate why file-system faults are important in the context of modern distributed storage systems (Section 2). Then, we describe our fault model and how our framework injects faults and observes behaviors (Section 3). Next, we present our behavior analysis and observations across systems (Section 4). Finally, we discuss related work (Section 5) and conclude (Section 6).

2 BACKGROUND AND MOTIVATION

We first provide background on why applications running atop file systems can encounter faults during operations such as read and write. Next, we motivate why such file-system faults are important in the context of distributed storage systems and the necessity of end-to-end data integrity and error handling for these systems.

2.1 File-System Faults

The layers in a storage stack beneath the file system consist of many complex hardware and software components [2]. At the bottom of the stack is the media (a disk or a flash device). The firmware above the media controls functionalities of the media. Commands to the firmware are submitted by the device driver. File systems can encounter faults for a variety of underlying causes, including media errors, mechanical and electrical problems in the disk, bugs in firmware, and problems in the bus controller [8, 9, 49, 55, 63, 80, 82]. Sometimes, corruptions can arise due to software bugs in other parts of the operating system [13], device drivers [89], and sometimes even due to bugs in file systems themselves [26].

Due to these reasons, two problems arise for file systems: *block errors*, where certain blocks are inaccessible (also called latent sector errors), and *block corruptions*, where certain blocks do not contain the expected data.

File systems can observe block errors when the disk returns an explicit error on detecting some problem with the block being accessed (such as in-disk ECC complaining that the block has a bit rot) [9, 80]. A previous study [9] of over 1 million disk drives over a period of 32 months has shown that 8.5% of near-line disks and about 1.9% of enterprise class disks developed one or more latent sector errors. More recent results show similar errors arise in flash-based SSDs [49, 55, 82]. Similarly, a recent study on flash reliability [82] over a period of 6 years has shown that as high as 63% and 2.5% of millions of flash devices experience at least one read and write error, respectively.

File systems can receive corrupted data due to a misdirected or a lost write caused by bugs in drive firmware [8, 60] or if the in-disk ECC does not detect a bit rot. Block corruptions are insidious, because blocks become corrupt in a way not detectable by the disk itself. File systems, in many cases, obliviously access such corrupted blocks and silently return them to applications. Bairavasundaram et al., in a study of 1.53 million disk drives over 41 months, showed that more than 400,000 blocks had checksum mismatches [8]. Anecdotal evidence has shown the prevalence of storage errors and corruptions [18, 38, 76]. Given the frequency of storage corruptions and errors, there is a non-negligible probability for file systems to encounter such faults.

In many cases, when the file system encounters a fault from its underlying layers, it simply passes it as-is onto the applications [63]. For example, the default Linux file system, ext4, simply returns errors or corrupted data to applications when the underlying block is not accessible or is corrupted, respectively. In a few other cases, the file system may transform the underlying fault into a different one. For example, btrfs and ZFS transform an underlying corruption into an error—when an underlying corrupted disk block is accessed, the application will receive an error instead of corrupted data [99]. In either case, we refer to these faults thrown by the file system to its applications as *file-system faults*.

2.2 Why Distributed Storage Systems?

Given that local file systems can return corrupted data or errors, the responsibility of data integrity and proper error handling falls to applications, as they care about safely storing and managing critical user data. Most single-machine applications such as stand-alone databases and non-replicated key-value storage systems solely rely on local file systems to reliably store user data; they rarely have ways to recover from local file-system faults. For example, on a read, if the local file system returns an error or corrupted data, then applications have no way of recovering that piece of data. Their best possible course of action is to reliably detect such faults and deliver appropriate error messages to users.

Modern distributed storage systems, much like single-machine applications, also rely on the local file system to safely manage critical user data. However, unlike single-machine applications,

distributed storage systems inherently store data in a replicated fashion. A carefully designed distributed storage system can potentially use redundancy to recover from errors and corruptions, irrespective of the support provided by its local file system. Ideally, even if one replica is corrupted, the distributed storage system as whole should not be affected as other intact copies of the same data exist on other replicas. Similarly, errors in one node should not affect the global availability of the system given that the functionality (application code) is also replicated across many nodes.

The case for end-to-end data integrity and error handling can be found in the classical end-to-end arguments in system design [79]. Ghemawat et al. also describe the need for such end-to-end checksum-based detection and recovery in the Google File System as the underlying cheap IDE disks would often corrupt data in the chunk servers [30]. Similarly, lessons from Google [22] in building large-scale Internet services emphasize how higher layer software should provide reliability. Given the possibility of end-to-end data integrity and error handling for distributed systems, we examine if and how well modern distributed storage systems employ end-to-end techniques to recover from local file-system faults.

3 TESTING DISTRIBUTED SYSTEMS

As we discussed in the previous section, file systems can throw errors or return corrupted data to applications running atop them; robust applications need to be able to handle such file-system faults. In this section, we first discuss our file-system fault model. Then, we describe our methodology to inject faults defined by our model and observe the effects of the injected faults.

3.1 Fault Model

Our fault model defines what file-system fault conditions an application can encounter. The goal of our model is to inject faults that are representative of fault conditions in current and future file systems and to drive distributed systems into error cases that are rarely tested.

Our fault model has two important characteristics. First, our model considers injecting exactly a *single fault* to a *single file-system block* in a *single node* at a time. While correlated file-system faults [8, 9] are interesting, we focus on the most basic case of injecting a single fault in a single node, because our fault model intends to give maximum recovery leeway for applications. Correlated faults, on the other hand, might preclude such leeway. For example, if two or more file-system blocks containing important application-level data structures are corrupted (possible in a correlated fault model), then there might be less opportunity for the application to salvage its state.

Second, our model injects faults only into application-level on-disk structures and not file-system metadata. File systems may be able to guard their own (meta)data [27]; however, if user data becomes corrupt or inaccessible, the application will either receive a corrupted block or perhaps receive an error (if the file system has checksums for user data). Thus, it is essential for applications to handle such cases.

Table 1 shows faults that are possible in our model during read and write operations and some examples of root causes in most commonly used file systems that can cause a particular fault. For all further discussion, we use the term block to mean a file-system block.

It is possible for applications to read a block that is corrupted (with zeros or junk) if a previous write to that block was lost or some unrelated write was misdirected to that block. For example, in the ext family of file systems and XFS, there are no checksums for user data, and so it is possible for applications to read such corrupted data without any errors. Our model captures such cases by corrupting a block with zeros or junk on reads.

Even on file systems such as btrfs and ZFS where user data is checksummed, detection of corruption may be possible but not recovery (unless mounted with special options such as *copies=2*

Table 1. Possible Faults and Example Causes

Type of Fault		Op	Example Causes
Block Corruption	zeros	Read	lost and misdirected writes in *ext* and *XFS*
	junk	Read	lost and misdirected writes in *ext* and *XFS*
Block Error	I/O error (EIO)	Read	latent sector errors in all file systems, disk corruptions in *ZFS*, *btrfs*
		Write	file system mounted read-only, on-disk corruptions in *btrfs*
	Space error (ENOSPC, EDQUOT)	Write	disk full, quota exceeded in all file systems
Bit Corruption		Read	bit rots not detected by in-device ECC in *ext* and *XFS*

The table shows file-systems faults captured by our model and example root causes that lead to a particular fault during read and write operations.

in ZFS). Although user data checksums employed by btrfs and ZFS prevent applications from accessing corrupted data, they return errors when applications access corrupted blocks. Our model captures such cases by returning similar errors on reads. Also, applications can receive EIO on reads when there is an underlying latent sector error associated with the data being read. This condition is possible on all commonly used file systems including ext4, XFS, ZFS, and btrfs.

Applications can receive EIO on writes from the file system if the underlying disk sector is not writable and the disk does not remap sectors, if the file system is mounted in read-only mode, or if the file being written is already corrupted in btrfs. On writes that require additional space (for instance, append of new blocks to a file), if the underlying disk is full or if the user's block quota is exhausted, applications can receive ENOSPC and EDQUOT, respectively, on any file system.

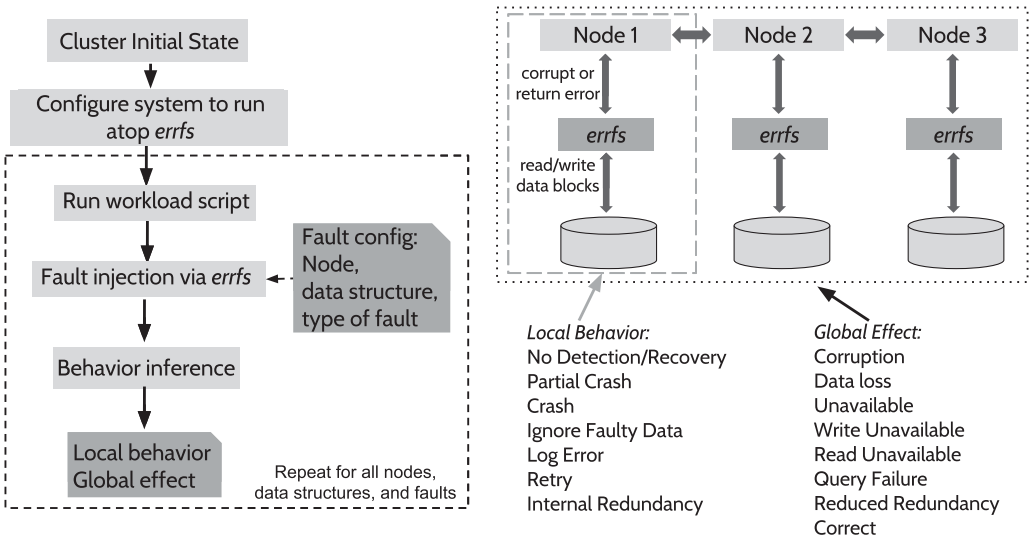
Our fault model also includes bit corruptions where applications read a similar-looking block with only a few bits flipped. This condition is possible when the in-disk ECC does not detect bit rots and the file system also does not detect such conditions (for example, XFS and ext) or when memory corruptions occur (e.g., corruptions introduced after checksum computation and before checksum verification [99]).

Our fault model injects faults in what we believe is a realistic manner. For example, if a block marked for corruption is written, subsequent reads of that block will see the last written data instead of corrupted data. Similarly, when a block is marked for read or write error and if the file is deleted and recreated (with a possible allocation of new data blocks), we do not return errors for subsequent reads or writes of that block. Similarly, when a space error is returned, all subsequent operations that require additional space will encounter the same space error. Notice that our model does not try to emulate any particular file system. Rather, it suggests an abstract set of faults possible on commonly used file systems that applications can encounter.

3.2 Methodology

We now describe our methodology to study how distributed systems react to local file-system faults. We built CORDS, a fault injection framework that consists of *errfs*, a FUSE [28] file system, and *errbench*, a set of workloads and a behavior-inference script for each system.

3.2.1 System Workloads. To study how a distributed storage system reacts to local file-system faults, we need to exercise its code paths that lead to interaction with its local file system. We



(a) Methodology Overview (b) errfs and behavior inference

Fig. 1. CORDS Methodology. (a) Overview of our methodology to study how distributed systems react to local file-system faults. (b) How errfs injects faults (corruptions and errors) into a block and how we observe the local behavior and the global effect of the injected fault.

crafted a workload suite, *errbench*, for this purpose; our suite consists of two workloads per system: read an existing data item and insert or update a data item.

3.2.2 Fault Injection. Figure 1(a) illustrates our methodology to analyze the behavior of distributed systems. We initialize the system under study to a known state by inserting a few data items and ensuring that they are safely replicated and persisted on disk. Our workloads either read or update the items inserted as part of the initialization. Next, we configure the application to run atop *errfs* by specifying its mount point as the `data-directory` of the application. Thus, all reads and writes performed by the application flow through *errfs* that can then inject faults. We run the application workload multiple times, each time injecting a single fault for a single file-system block through *errfs*. If the application-level data structure spans multiple file-system blocks, then we inject a fault only in a single file-system block constituting that data structure at a time. For bit corruptions, we flip a bit in a single field within a block at a time.

errfs can inject two types of block corruptions: corrupted with *zeros* or *junk*. For block corruptions, *errfs* performs the read and changes the contents of the block that is marked for corruption before returning to the application, as shown in Figure 1(b). *errfs* can inject three types of block errors: `EIO` on reads (*read errors*), `EIO` on writes (*write errors*), or `ENOSPC` and `EDQUOT` on writes that require additional space (*space errors*). To emulate errors, *errfs* does not perform the operation but simply returns an appropriate error code. For bit corruptions, *errfs* requires application-specific information consisting of various fields within a block along with their offsets and lengths. To inject a bit corruption, *errfs* flips a bit in the field that is marked for corruption before returning the data.

3.2.3 Behavior Inference. For each run of the workload where a single fault is injected, we observe how the system behaves. Our system-specific behavior-inference scripts glean system

behavior from the system's log files and client-visible outputs such as server status, return codes, errors (`stderr`), and output messages (`stdout`). Once the system behavior for an injected fault is known, we compare the observed behavior against expected behaviors. The following are the expected behaviors we test for:

- *Committed data should not be lost*
- *Queries should not silently return corrupted data*
- *The cluster should be available for reads and writes*
- *Queries should not fail after retries.*

We believe our expectations are reasonable, since a single fault in a single node of a distributed system should ideally not result in any undesirable behavior. If we find that an observed behavior does not match expectations, then we flag that particular run (a combination of the workload and the fault injected) as erroneous, analyze relevant application code, contact developers, and file bugs.

Local Behavior and Global Effect. In a distributed system, multiple nodes work with their local file system to store user data. When a fault is injected in a node, we need to observe two things: local behavior of the node where the fault is injected and global effect of the fault, as shown in Figure 1(b).

In most cases, a node locally reacts to an injected fault. As shown in Figure 1(b), a node can *crash* or *partially crash* (only a few threads of the process are killed) due to an injected fault. In some cases, the node can fix the problem by *retrying* any failed operation or by using *internally redundant* data (cases where the same data is redundant across files within a replica). Alternatively, the node can detect and *ignore* the corrupted data or just *log an error message*. Finally, the node may *not even detect* or take any measure against a fault.

The global effect of a fault is the result that is externally visible. The global effect is determined by how distributed protocols (such as leader election, consensus, recovery, repair) react in response to the local behavior of the faulty node. For example, even though a node can locally ignore corrupted data and lose it, the global recovery protocol can potentially fix the problem, leading to a *correct* externally observable behavior. Sometimes, because of how distributed protocols react, a global *corruption*, *data loss*, *read-unavailability*, *write-unavailability*, *unavailability*, or *query failure* might be possible. When a node simply crashes as a local reaction, the system runs with *reduced redundancy* until manual intervention.

These local behaviors and global effects for a given workload and a fault might vary depending on the role played (leader or follower) by the node where the fault is injected. For simplicity, we uniformly use the terms *leader* and *follower* instead of *master* and *slave*.

We note here that our workload suite and model are *not complete*. First, our suite consists only of simple read and write workloads while more complex workloads may yield additional insights. Second, our model does not inject all possible file-system faults; rather, it injects only a subset of faults such as corruptions, read, write, and space errors. However, even our simple workloads and fault model drive systems into corner cases, leading to interesting behaviors. Our framework can be extended to incorporate more complex faults and our workload suite can be augmented with more complex workloads; we leave this as an avenue for future work.

4 RESULTS AND OBSERVATIONS

We studied eight widely used distributed storage systems: Redis (v3.0.4), ZooKeeper (v3.4.8), Cassandra (v3.7), Kafka (v0.9), RethinkDB (v2.3.4), MongoDB (v3.2.0), LogCabin (v1.0), and CockroachDB (beta-20160714). We configured all systems to provide the highest safety guarantees

possible; we enabled checksums, synchronous replication, and synchronous disk writes. We configured all systems to form a cluster of three nodes and set the replication factor at three.

We present our results in four parts. First, we present our detailed behavioral analysis for each system (Section 4.1). Second, we derive and present a set of observations related to data integrity and error handling *across* all eight systems (Section 4.2) and summarize our results (Section 4.3). Next, we discuss features of current file systems that can impact the problems we found (Section 4.4). Finally, we discuss why modern distributed storage systems are not tolerant of single file-system faults and describe our experience interacting with developers (Section 4.5).

4.1 System Behavior Analysis

We now present our detailed behavioral analysis of all systems. For each system, we describe the behaviors when block corruptions and block errors are injected into different on-disk structures. For each system, we also show the format of the on-disk files and the logical data structures in the system. The on-disk structure names take the form *file_name.logical_entity*. We derive the logical entity name from our understanding of the on-disk format of the file. For a few systems (Redis, Cassandra, and Kafka), we also present the behaviors when bit corruptions are injected.

4.1.1 Redis. Redis is a popular data structure store, used as database, cache, and message broker. Redis uses asynchronous primary-backup replication; thus, there is a window for data loss. However, Redis can be configured to accept a write only if at least N followers with a lag of fewer than M seconds are currently connected to the leader. Redis does not elect a leader automatically when the current leader fails.

On-disk Structures. Figure 2(a) shows the on-disk structures of Redis. Redis uses a simple appendonly log file (*aof*) to store the sequence of commands or operations that modify the database state. The appendonly file is not checksummed. Before recording a sequence of operations, a database identifier is logged; this identifier specifies the database to which the operations are to be applied when the appendonly file is later replayed. Periodic snapshots are taken from the *aof* to create a redis database file (*rdb*). During startup, the followers re-synchronize the *rdb* file from the leader. The entire *rdb* file is protected by a single checksum.

Behavior Analysis. Figure 2(b) shows the behavior of Redis when block corruptions and block errors are introduced into different on-disk structures. When there are *corruptions* in metadata structures in the appendonly file or *errors* in accessing the same, the node simply crashes (first row of local behavior boxes for both workloads in Figure 2(b)). If the leader crashes, then the cluster becomes unavailable and if the followers crash, then the cluster runs with reduced redundancy (first row of global effect for both workloads). Redis does not use checksums for user data in the appendonly file; thus, it does not detect corruptions (second row of local behavior for both workloads). If the leader is corrupted, then it leads to a global user-visible corruption, and if the followers are corrupted, then there is no harmful global effect (second row of global effect for read workload). Figure 3(a) shows how the re-synchronization protocol propagates corrupted user data in *aof* from the leader to the followers leading to a global user-visible corruption. In contrast, *errors* in appendonly file user data lead to crashes (second row of local behavior for both workloads); crashes of the leader and followers lead to cluster unavailability and reduced redundancy, respectively (second row of global effect for both workloads).

Problems in the first block of redis_database are fixed by retrying and creating the redis_database file again from data in the appendonly file (third row in Figure 2(b)). When the redis_database file on a follower is corrupted, it crashes, leading to reduced redundancy. Since the leader sends the *rdb* file during re-synchronization, corruption in the same causes both the

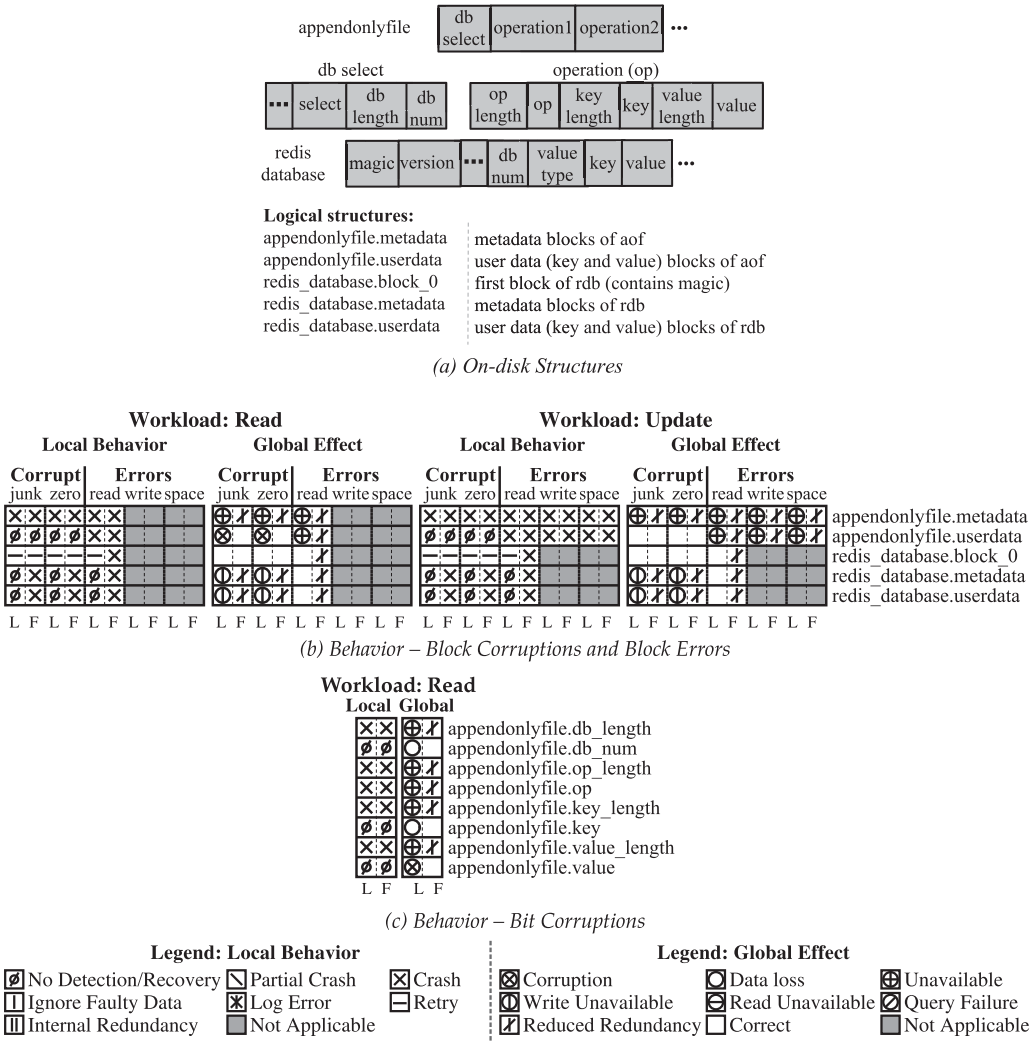


Fig. 2. Redis. (a) The on-disk format of the files and the logical data structures of Redis. The logical structures take the following form: *file_name.logical_entity*. If a file can be contained in a single file-system block, then we do not show the logical entity name. (b) System behavior when corruptions (corrupted with either junk or zeros), read errors, write errors, and space errors are injected in various on-disk logical structures in Redis. Within each system workload (read and update), there are two boxes—first, local behavior of the node where the fault is injected and, second, clusterwide global effect of the injected fault. The rightmost annotation shows the on-disk logical structure in which the fault is injected. Annotations on the bottom show where a particular fault is injected (L, leader/master; F, follower/slave). A gray box for a fault and a logical structure combination indicates that the fault is not applicable for that logical structure. For example, write errors are not applicable for any data structures in the read workload (since they are not written) and hence shown as gray boxes. (c) The behavior when bit corruptions are injected. For bit corruptions, we flip a *single bit* in a field within the on-disk structure. For example, appendonlyfile.db_num is part of appendonlyfile.metadata. The legend at the bottom is common to both (b) and (c).

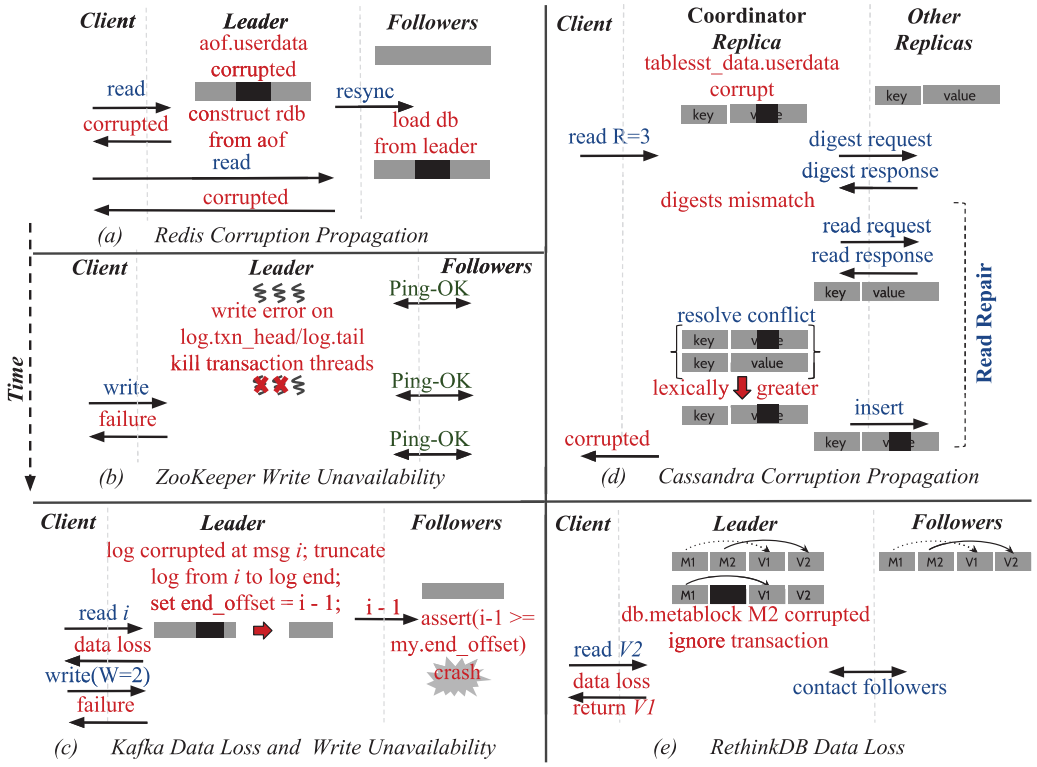


Fig. 3. Example Bugs. The figure depicts some of the bugs we discovered in Redis, ZooKeeper, Cassandra, Kafka, and RethinkDB. Time flows downwards as shown on the left. The black portions denote corruption.

followers to crash. These crashes ultimately make the cluster unavailable for writes (fourth and fifth rows in Figure 2(b)).

Bit Corruptions. Figure 2(c) shows the behavior of Redis when bit corruptions are injected. A single flipped bit in most of the appendonly file metadata structures results in a failed deserialization, ultimately leading to a crash. If the leader crashes, then the cluster becomes unavailable, and if the followers crash, then the cluster runs with reduced redundancy. On the leader, a bit flip in the key field results in a silent data loss while a bit flip in the value field results in a silent corruption.

Redis maintains a database identifier (db_num) for each database. When some data are inserted or updated, first the appropriate database (specifically, the database identifier) is recorded in the appendonly file followed by the actual update. If a bit in the recorded database identifier (P) flips and so changes to a new value (Q), then all succeeding operations in the appendonly file are redirected to database Q instead of P. This single bit flip in the database identifier results in a silent data loss when database P is queried while supplying spurious data when database Q is queried.

In our bit-corruption experiments, we reduce the granularity of our faults: We flip a *single bit* in a field within the on-disk structure. Our bit-corruption experiments help uncover interesting behaviors not discovered through our block-corruption experiments. For instance, consider the field *appendonlyfile.db_num* that is part of *appendonlyfile.metadata*. When we inject a coarse block corruption in *appendonlyfile.metadata* on the leader (first row in Figure 2(b)), the leader crashes,

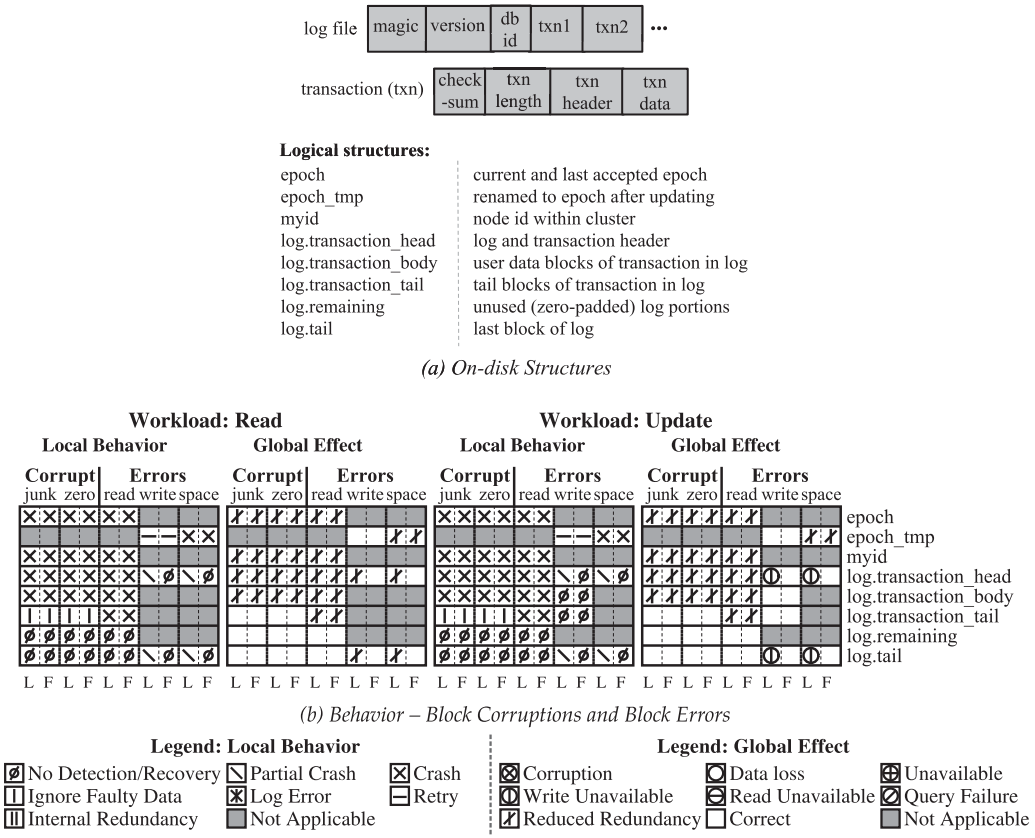


Fig. 4. ZooKeeper. (a) The on-disk format of the files and the logical data structures of ZooKeeper. (b) System behavior when faults are injected in various structures in ZooKeeper.

making the cluster unavailable. In contrast, when we inject a fine-grained bit flip in *appendonly-file.db_num* on the leader (second row in Figure 2(c)), it results in a data loss, as described above.

4.1.2 ZooKeeper. ZooKeeper is a popular service for storing configuration information, naming, and distributed synchronization. ZooKeeper provides a hierarchical name space (a *data tree*) and supports operations such as creation and deletion of nodes in the data tree. ZooKeeper implements state machine replication and uses an atomic broadcast protocol (ZAB) to maintain identical states in all the nodes in the system. The system remains available as long as a majority of the nodes are functional. It provides durability by persisting operations in a log and persisting periodic snapshots of the data tree.

On-disk Structures: Figure 4(a) shows the on-disk structures of ZooKeeper. ZooKeeper uses *log* files to append user data. The log contains a log header (magic, version, etc.) followed by a sequence of transactions. A transaction consists of a transaction header and is protected by a checksum. The transaction header contains epoch, session id, and so on. ZooKeeper maintains two important metadata structures: *epoch* (accepted and current epoch) and *myid* (node identifier). Epochs are updated by first writing to epoch_tmp and then renaming it to epoch.

Behavior Analysis. Figure 4(b) shows the behavior when block corruptions and block errors are introduced in ZooKeeper. ZooKeeper can detect corruptions in the transaction_head and transaction_body of the *log* using checksums but reacts by simply crashing (fourth and fifth rows of local behavior for both workloads in Figure 4(b)). When epoch and myid are corrupted or cannot be read, the node simply crashes (first and third rows for both workloads). Similarly, it crashes in most error cases, leading to reduced redundancy. In all crash scenarios, ZooKeeper can reliably elect a new leader, thus ensuring availability. ZooKeeper ignores a transaction locally when its tail is corrupted (sixth row of local behavior for both workloads); the leader election protocol prevents that node from becoming the leader. Eventually, the corrupted node repairs its log by contacting the leader, leading to correct behavior (sixth row of global effect for both workloads).

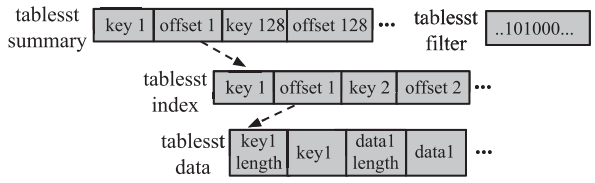
Unfortunately, ZooKeeper does not recover from write errors to the transaction head and log tail (fourth and eighth rows in Figure 4(b)). Figure 3(b) depicts this scenario. On write errors during log initialization, the error handling code tries to gracefully shutdown the node but kills only the transaction processing threads; the quorum thread remains alive (partial crash). Consequently, other nodes believe that the leader is healthy and do not elect a new leader. However, since the leader has partially crashed, it cannot propose any transactions, leading to an indefinite write unavailability. Notice that this scenario does not cause a harmful global effect for the read workload as reads can be locally served by any ZooKeeper node, without requiring the leader to propose new transactions.

4.1.3 Cassandra. Cassandra is a Dynamo-like [23] NoSQL store. Unlike other systems we study, Cassandra is a decentralized system; it does not have a leader and followers. The system divides all data evenly around a cluster of nodes, which form a ring. Cassandra replicates the data to a number of nodes specified by the replication factor. It also supports different read and write consistency levels. In Cassandra, rows are organized into tables and the rows are divided among nodes in the cluster based on a hash of the primary key. Cassandra also provides a SQL-like query language (CQL).

On-disk Structures: In Cassandra, the local storage engine is a variation of log-structured merge (LSM) trees [59] that stores data in sstables. A separate sstable is maintained for each key-space; we refer to the sstables of user-created key-space as *tablesst*. Figure 5(a) shows the on-disk files in an sstable. Each sstable consists of a Bloom filter (tablesst_filter); the filter provides a fast way to determine whether a given key is present or not. If a key is found in the filter, then the table summary (tablesst_summary) and table index (tablesst_index) are accessed. The tablesst_index contains the offset of a data item in the data file (tablesst_data). The tablesst_data contains all the rows in the table.

Behavior Analysis. Cassandra enables checksum verification on user data only as a side effect of enabling compression. Therefore, we conduct two experiments in Cassandra—one with compression disabled for user tables and the other with compression enabled.

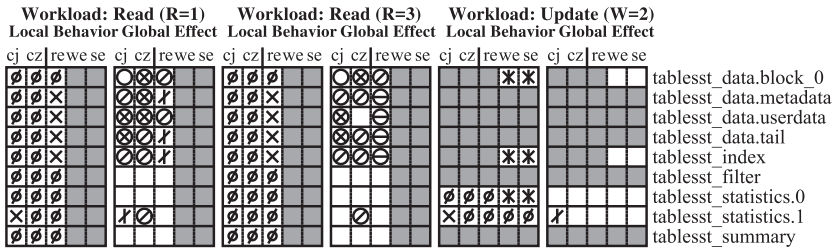
Figure 5(b) shows the results for block corruptions and block errors when compression is disabled for user sstables. When compression is turned *off*, corruptions are not detected on user data in tablesst_data (third row of local behavior for read workloads in Figure 5(b)). On a read query, a coordinator node collects and compares digests (hashes) of the data from R replicas [20]. If the digests mismatch, then conflicts in the values are resolved using a *latest timestamp wins* policy. If there is a tie between timestamps, then the lexically greatest value is chosen and installed on other replicas [39]. As shown in Figure 3(d), on $R = 3$, if the corrupted value is lexically greater than the original value, then the corrupted value is returned to the user and the corruption is propagated to other intact replicas (third row of global effect for $R = 3$ read workload when corrupted with



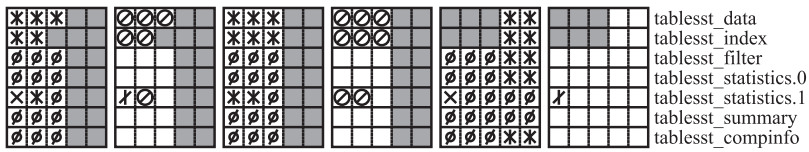
Logical structures:

tablesst_data.block_0	block 0 of data file in sstable for userdata
tablesst_data.metadata	meta data blocks of data file in sstable
tablesst_data.userdata	user data blocks of data file in sstable
tablesst_data.tail	metadata blocks of rdb
tablesst_index	index file in sstable
tablesst_filter	Bloom filter file in sstable
tablesst_statistics.0	block zero of statistics file in sstable
tablesst_statistics.1	block one of statistics file in sstable
tablesst_summary	summary file in sstable
tablesst_compinfo	contains compression information

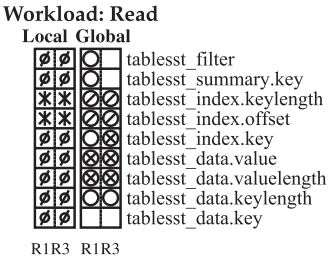
(a) On-disk Structures



(b) Behavior – Block Corruptions and Block Errors (sstable compression = off)



(c) Behavior – Block Corruptions and Block Errors (sstable compression = on)



(d) Behavior – Bit Corruptions (sstable compression = off)

Fig. 5. Cassandra. (a) The on-disk format of different structures in Cassandra. (b) and (c) System behavior in the presence of block corruptions (corrupted with either junk (cj) or zeros(cz)), read errors (re), write errors (we), and space errors (se) when sstable compression is turned off and turned on, respectively. (d) The behavior in the presence of bit corruptions when sstable compression is off; the annotations on the bottom indicate the read quorum (R1, quorum of 1; R3, quorum of 3).

junk). On the other hand, if the corrupted value is lexically lesser, it fixes the corrupted node (third row of global effect for $R = 3$ read workload when corrupted with zeros). Reads to a corrupted node with $R = 1$ always return corrupted data. Faults in tablesst_index cause query failures (fifth row of global effect for read workloads). In most cases, user-visible problems that are observed in the $R = 1$ configuration are not fixed even when run with $R = 3$.

Figure 5(c) shows the results when compression is enabled for user sstables. When compression is enabled, Cassandra maintains a checksum for every compressed block. When the value in the compressed data become corrupted, decompression fails due to a mismatch between the stored checksum and computed checksum of the decompressed data. Thus, corruptions to tablesst_data.userdata are detected and results in failures of table scans and point queries to the data within this block (first row for read workloads in Figure 5(c)); point queries to the data not in this corrupted block are not affected. The corruption is not fixed automatically even when queries are run with $R = 3$ and results in query failures. Since we do not alter the compression feature of system schema sstables, we do not repeat this experiment for these structures.

Bit Corruptions. Figure 5(d) shows the behavior of Cassandra when bit corruptions are injected and compression is enabled for user sstables. In Cassandra, the read path involves accessing several on-disk structures. For a point query of a key, first, the key is queried in the Bloom filter (tablesst_filter); if the filter indicates the key's presence, then the table summary (tablesst_summary) and table index (tablesst_index) are accessed to determine the offset of the entry in the data file (tablesst_data). Finally, the value is read from the data file. With $R = 1$, a single bit corruption in the filter causes a data loss (first row of global effect for $R = 1$ read workload in Figure 5(d)). Similarly, a single bit corruption in the key field in the table summary results in a data loss (second row of global effect for $R = 1$ read workload). With $R = 3$, the above two problems are masked (first and second rows for $R = 3$ read workload). A flipped bit in the keylength and offset of the table index results in query failures with both $R = 1$ and $R = 3$ (third and fourth rows of global effect). With $R = 1$, a corrupted key in the table index leads to a silent data loss (fifth row of global effect). When a key K in the index is corrupted to K', a table scan with $R = 3$ results in a surprising outcome: First, the scan result contains a spurious row with key K' with the same value as the one for K; furthermore, the spurious row is propagated to all other nodes (fifth row of global effect with $R = 3$). A single corrupted bit in valuelength or the value in the table data results in silent corruption and corruption propagation in $R = 1$ and $R = 3$, respectively.

Introducing bit corruptions into various fields within a block helps uncover interesting behaviors in Cassandra not discovered using block-corruption experiments. For example, consider the fields *keylength*, *offset*, and *key* that are a part of *tablesst_index*. When we inject block corruptions in *tablesst_index* (fifth row in Figure 5(b)), it results in query failures. In contrast, when we flip a bit in the *key* of *tablesst_index*, it results in the surprising outcome where a spurious row is silently propagated to all other nodes (fifth row in Figure 5(d)). Similarly, while a block corruption in *tablesst_summary* results in a correct behavior (last row in Figure 5(b)), a bit flip in *key* of *tablesst_summary* results in a data loss (second row for $R = 1$ in Figure 5(d)).

4.1.4 Kafka. Kafka is a distributed persistent message queue in which clients can publish and subscribe to messages. Kafka is run as a cluster consisting of a leader and a set of followers. The system stores streams of messages in categories called topics; a topic can have zero or more consumers that subscribe to it. Each message in a topic consists of a key, a value, and a timestamp.

On-disk Structures: The on-disk structures of Kafka are shown in Figure 6(a). Incoming messages are appended to a *log* file. Each message is checksummed and is associated with a message id and an optional key. Kafka maintains an *index* file that indexes messages to byte offsets within the log.

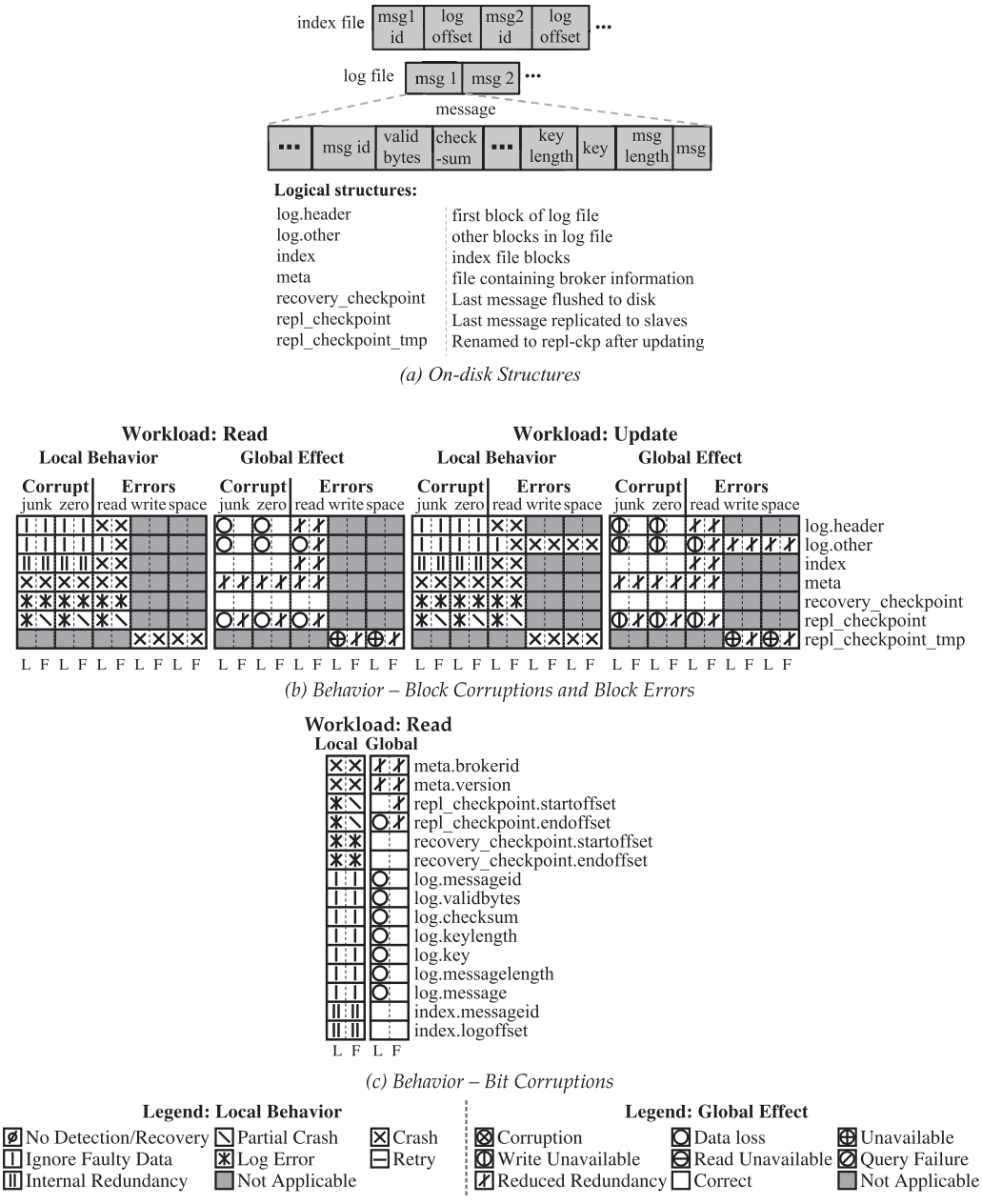


Fig. 6. Kafka. (a) The on-disk format of the files and the logical data structures of Kafka. (b) System behavior in the presence of block corruptions and block errors. (c) The behavior when bit corruptions are injected.

Important metadata structures (such as the node identifier) are maintained in a file called *meta*. The *replication_checkpoint* and *recovery_checkpoint* structures indicate how many messages are replicated to followers so far and how many messages are flushed to disk so far, respectively. The replication offsets are updated by first writing to a temporary file (*repl_checkpoint_tmp*) and then renaming it to the final file.

Behavior Analysis. Figure 6(b) shows the behavior of Kafka when block corruptions and block errors are introduced into different structures. On read and write errors, Kafka mostly crashes. Figure 3(c) shows the scenario where Kafka can lose data and become unavailable for writes. When a log entry is corrupted on the leader, it locally ignores that entry and all subsequent entries in the log (first and second rows of local behavior boxes for both workloads in Figure 6(b)), resulting in a data loss. The leader then instructs the followers to do the same. On receiving this instruction from the leader, the followers check whether the leader's offset is greater than their checkpointed offset. If this condition does not hold, then the followers hit a fatal assertion and simply crash. Once the followers crash, the cluster becomes unavailable for writes (first and second rows of global effect for write workload).

We conducted another experiment where the corruption on the leader occurs before the followers checkpoint the message offsets to their *recovery-offset-checkpoint* file. If the followers have not checkpointed the entries (that have been truncated on the leader), then they truncate the entries as instructed by the leader, leading to a silent permanent data loss. In this case, the followers continue to operate without crashing.

Corruption in index is fixed using internal redundancy (third row of local behavior for both workloads). Faults in the *replication_checkpoint* of the leader result in a data loss (sixth row of global effect for read workload) as the leader is unable to record the replication offsets of the followers. Kafka becomes unavailable when the leader cannot read or write *replication_checkpoint* and *replication_checkpoint_tmp*, respectively.

Bit Corruptions. Figure 6(c) shows the behavior of Kafka when bit corruptions are injected. On a bit flip in any field of the message log, the node truncates the corrupted message and all subsequent messages. If this corruption occurs on the follower, then the leader supplies the truncated messages to the followers. The same single bit flip on the leader leads to a silent data loss. A bit flip in the replication offsets sometimes causes a data loss.

4.1.5 RethinkDB. RethinkDB is a distributed database suited for pushing query results to real-time web applications [73]. RethinkDB uses the Raft consensus protocol to maintain cluster metadata. It relies on the underlying storage stack to handle data integrity and does not maintain checksums for user data.

On-disk Structures: RethinkDB uses a persistent B-tree to store all data. Transactions are stored at the leaf nodes of the tree; a transaction consists of three blocks: *db.transaction_head*, *db.transaction_body*, and *db.transaction_tail*. *metablocks* in the B-tree point to the data blocks that constitute the current and the previous version of the database. On an update, new data blocks are first carefully written and flushed to disk. Then, the *metablock* with checksums is updated to point to the new data blocks, thus enabling atomic updates. The B-tree data blocks and the metablocks are part of a single database file, as shown in Figure 7(a).

Behavior Analysis. Figure 7(b) shows the behavior when block corruptions and block errors are introduced in RethinkDB. On any fault in database header and internal B-tree nodes, RethinkDB simply crashes (first and second rows of local behavior for both workloads in Figure 7(b)). If the leader crashes, then a new leader is automatically elected. RethinkDB relies on the file system to ensure the integrity of data blocks; hence, it does not detect corruptions in the transaction body and tail (fifth and sixth rows of local behavior). When these blocks of the leader are corrupted, RethinkDB silently returns corrupted data (fifth and sixth rows of global effect for the leader).

Figure 3(e) depicts how data are silently lost when the transaction head or the metablock pointing to the transaction is corrupted on the leader (last row of global effect for the leader). Even though there are intact copies of the same data on the followers, the leader does not fix its

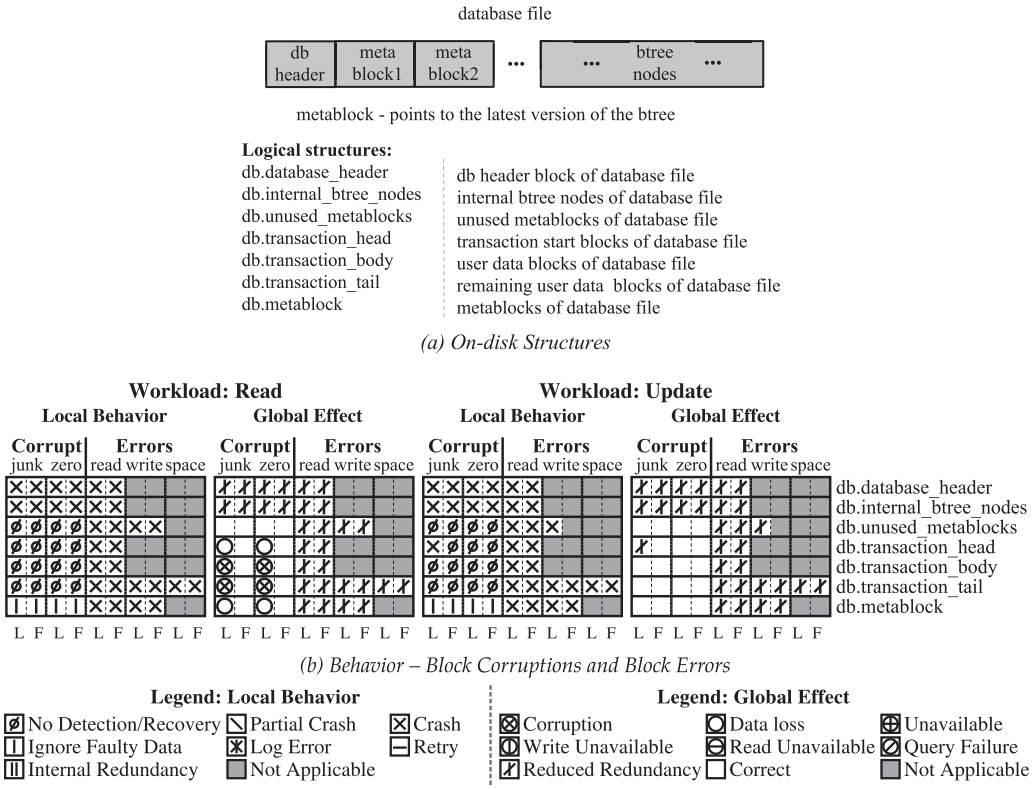


Fig. 7. RethinkDB. (a) The on-disk format of the files and the logical data structures of RethinkDB. (b) System behavior when faults are injected in various on-disk logical structures.

corrupted or lost data, even when we perform the reads with the *majority* option. When the followers are corrupted, they are not fixed by contacting the leader. Although this does not lead to an immediate user-visible corruption or loss (because the leader's data is the one finally returned), it does so when the corrupted follower becomes the leader in the future.

4.1.6 MongoDB. MongoDB is a popular document-oriented database that uses JSON-like documents [52]. MongoDB provides high availability using replica sets. It uses primary-backup replication with each replica set consisting of a primary and a set of secondaries. All writes and reads are done on the primary by default. When a primary fails, the replica set automatically elects its new primary. MongoDB supports multiple storage engines. For our experiments we use WiredTiger [54] as the storage engine.

On-disk Structures: Figure 8(a) shows the different on-disk files in MongoDB. When an item is inserted or updated, it is added to the *journal* and the in-memory database is updated. If the write operation specifies the option *j* as true, then WiredTiger forces an fsync of the journal. WiredTiger uses multi-version concurrency control, and periodically a consistent view of the in-memory data is checkpointed to the *collections* file and *index* file. A master *WiredTiger* file contains information on the latest checkpoint files. The storage engine information is stored in a *storage_bson* file.

Behavior Analysis. Figure 8(b) shows the results for block corruptions and block errors in MongoDB. MongoDB simply crashes on most errors, leading to reduced redundancy. A new leader

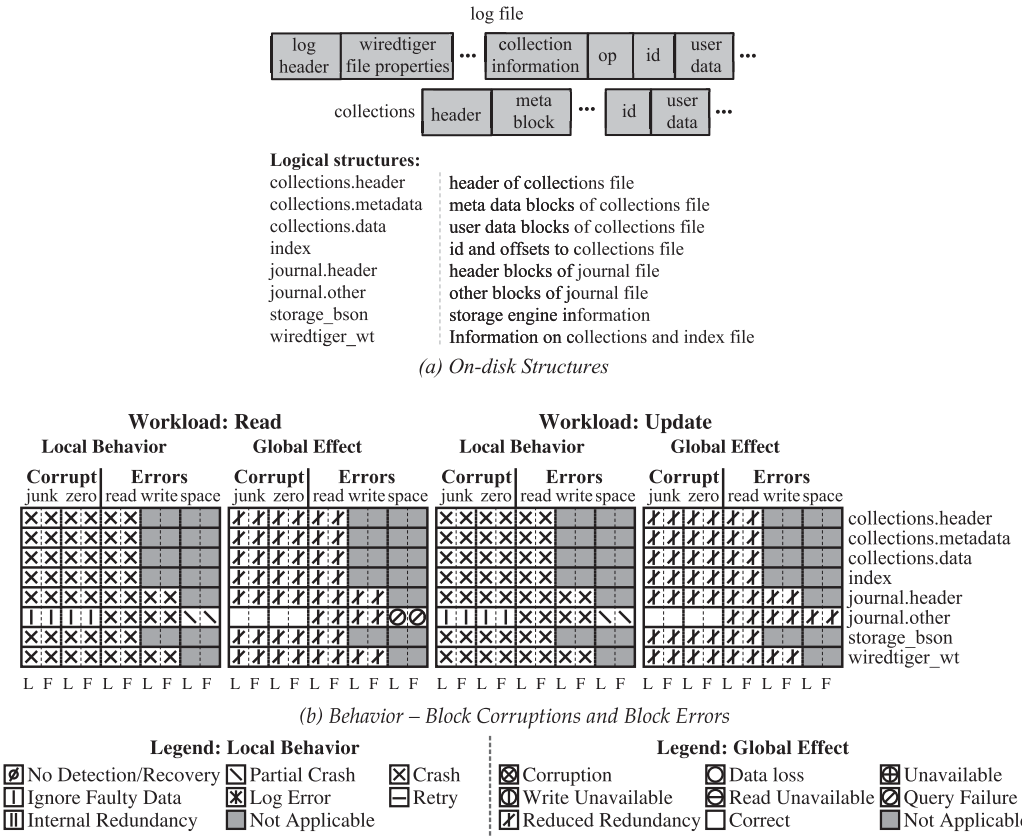


Fig. 8. MongoDB. (a) The on-disk format of the files and the logical data structures of MongoDB. (b) System behavior when faults are injected in various on-disk logical structures.

is automatically elected if the current leader crashes. MongoDB employs checksums for all files; corruption in any block of any file causes a checksum mismatch and an eventual crash, resulting in reduced redundancy.

One exception to the above is when blocks other than journal header are corrupted. In this case, MongoDB detects and ignores the corrupted blocks (sixth row of local behavior in Figure 8(b)); then, the corrupted node truncates its corrupted journal, descends to become a follower, and, finally, repairs its journal by contacting the leader. In a corner case where there are space errors while appending to the journal, queries fail (sixth row of global effect in Figure 8(b)).

4.1.7 LogCabin. LogCabin provides a replicated and consistent data store that serves as a place for other distributed systems to maintain their core metadata such as configuration settings. LogCabin implements state machine replication and uses the Raft consensus protocol [46]. In LogCabin, all reads and writes go through the leader by default.

On-disk Structures: LogCabin implements a segmented log [78] to store data; each segment is a file on the file system. The format of a segment file is shown in Figure 9(a). There are two types of segment files—the *open* segment and the *closed* segment. The *open* segment is the current file to which data is appended. When the *open* segment is fully utilized, it is *closed* and a new segment is opened. Two metadata files (*metadata1* and *metadata2*) maintain the Raft metadata and

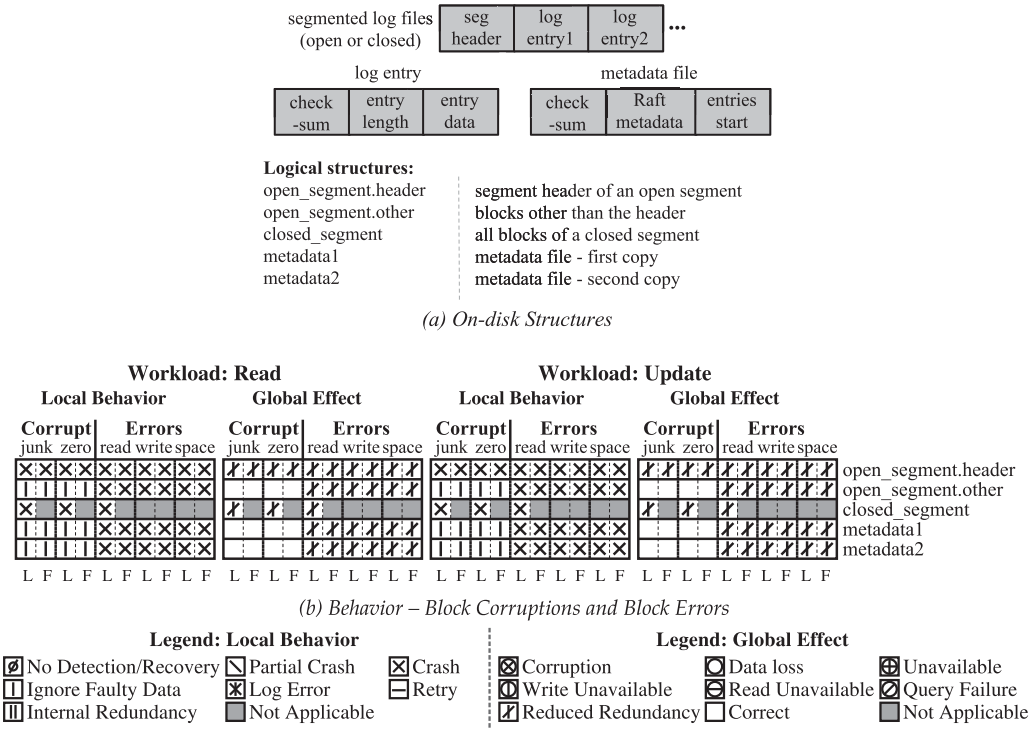


Fig. 9. LogCabin. (a) The on-disk format of the files and the logical data structures of LogCabin. (b) System behavior when faults are injected in various on-disk logical structures.

information about the log. The metadata files are updated alternately; when a metadata file is partially updated or corrupted, LogCabin uses the other metadata file that contains slightly older metadata.

Behavior Analysis. Figure 9(b) shows the behavior when block corruptions and block errors are introduced in LogCabin. LogCabin crashes on all read, write, and space errors. Similarly, if an *open* segment file header (first row in Figure 9(b)) or blocks in a *closed* segment (third row in the figure) are corrupted, LogCabin simply crashes. LogCabin recognizes corruption in any other blocks in an *open* segment using checksums and reacts by simply discarding and ignoring the corrupted entry and all subsequent entries in that segment (second row of local behavior). If a log pointer file is corrupted, then LogCabin ignores that pointer file and uses the other pointer file (fourth and fifth rows of local behavior).

In the above two scenarios, the leader election protocol ensures that the corrupted node does not become the leader; the corrupted node becomes a follower and fixes its log by contacting the new leader. This ensures that in any fault scenario, LogCabin would not globally corrupt or lose user data.

4.1.8 CockroachDB. CockroachDB is a distributed SQL database built atop a transactional and strongly consistent key-value store. It is built to survive disk, machine, rack, and data-center failures. CockroachDB uses Raft and so long as a majority of replicas remain available, the system can continue to make progress. It supports strongly consistent ACID transactions and also provides an SQL-like query language [14].

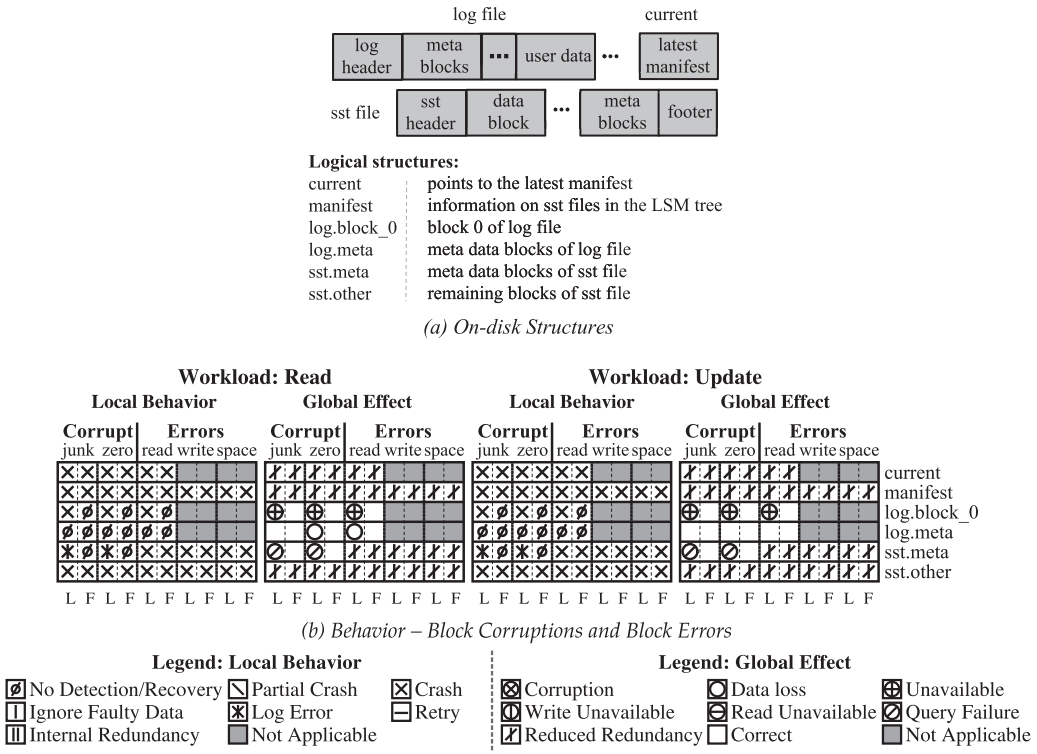


Fig. 10. CockroachDB. (a) The on-disk format of the files and the logical data structures of CockroachDB. (b) System behavior when faults are injected in various on-disk logical structures.

On-disk Structures: Figure 10(a) shows the different on-disk files in CockroachDB. CockroachDB uses a tuned version of RocksDB for its local storage; the storage engine is an LSM tree that appends incoming data to a persistent *log*; the in-memory data are then periodically compacted to create the *sst* files. The *manifest* file lists the set of sst files that make up a particular level in the LSM tree, and the *current* file points to the latest manifest.

Behavior Analysis. Figure 10(b) shows the results for block corruptions and block errors in CockroachDB. Most of the time, CockroachDB simply crashes on corruptions and errors on any data structure, resulting in reduced redundancy. Faults in the first block of the log file on the leader lead to total cluster unavailability as some followers also crash following the crash of the leader (third row of global effect). Corruptions and errors in a few other log metadata blocks can cause data loss where CockroachDB silently returns zero rows (fourth row of global effect). Corruptions in sst files cause queries to fail (fifth row of global effect) with error messages such as *table does not exist* or *db does not exist*. Overall, we found that CockroachDB has many problems in fault handling. However, the reliability may improve in the future, since CockroachDB is still under active development.

4.2 Observations Across Systems

We now present a set of observations with respect to data integrity and error handling *across* all eight systems.

Table 2. Data Integrity Strategies

Technique	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Metadata Checksums	*P*	√	√	√	*P*	√	√	√
Data Checksums	*P*	√[a]	√[$]	√		√	√	√
Background Scrubbing								√
External Repair Tools	√		√			√		√
Snapshot Redundancy	*P**	*P**				*P**		

P, applicable only for some on-disk structures; a, Adler32 checksum; ∗, only for certain amount of time; $, unused when compression is off.
The table shows techniques employed by modern systems to ensure data integrity of user-level application data.

#1: Systems employ diverse data integrity strategies. Table 2 shows different strategies employed by modern distributed storage systems to ensure data integrity. As shown, systems employ an array of techniques to detect and recover from corruption. The table also shows the diversity across systems. On one end of the spectrum, there are systems that try to protect against data corruption in the storage stack by using checksums (e.g., ZooKeeper, MongoDB, CockroachDB) while the other end of spectrum includes systems that completely trust and rely on the lower layers in the storage stack to handle data integrity problems (e.g., RethinkDB and Redis). Despite employing numerous data integrity strategies, all systems exhibit undesired behaviors.

Sometimes, *seemingly unrelated configuration settings affect data integrity*. For example, in Cassandra, checksums are verified only as a side effect of enabling compression. Due to this behavior, corruptions are not detected or fixed when compression is turned off, leading to user-visible silent corruption.

We also find that a few systems use *inappropriate checksum algorithms*. For example, ZooKeeper uses Adler32, which is suited only for error detection after decompression and can have collisions for very short strings [48]. In our experiments, we were able to inject corruptions that caused checksum collisions, driving ZooKeeper to serve corrupted data. We believe that it is not unreasonable to expect metadata stores like ZooKeeper to store small entities such as configuration settings reliably. In general, we believe that more care is needed to understand the robustness of possible checksum choices.

#2: Faults are often undetected. We find that faults are often locally undetected. Sometimes, this leads to an immediate harmful global effect. For instance, in Redis, corruptions in the append-only file of the leader are undetected, leading to global silent corruption. Also, corruptions in the rdb of the leader are also undetected and, when sent to followers, cause them to crash, leading to unavailability. Similarly, in Cassandra, corruption of tablesst_data is undetected, which leads to returning corrupted data to users and sometimes propagating it to intact replicas. Likewise, RethinkDB does not detect corruptions in the transaction head on the leader, which leads to a global user-visible data loss. Similarly, corruption in the transaction body is undetected leading to global silent corruption. The same faults are undetected also on the followers; a global data loss or corruption is possible if a corrupted follower becomes the leader in future.

While some systems detect and react to faults purposefully, some react to faults only as a side effect. For instance, ZooKeeper, MongoDB, and LogCabin carefully detect and react to corruptions. On the other hand, Redis, Kafka, and RethinkDB sometimes react to a corruption only as a side effect of a failed deserialization.

#3: Crashing is the most common reaction. We observe that *crashing is the most common local reaction to faults*. When systems detect corruption or encounter an error, they simply crash, resulting in reduced redundancy (as is evident from the abundance of crash symbols in local behaviors of the figures in behavior analysis). Although crashing of a single node does not immediately affect cluster availability, total unavailability becomes imminent as other nodes also can fail subsequently. Also, workloads that require writing to or reading from all replicas will not succeed even if one node crashes. After a crash, simply restarting does not help if the fault is sticky; the node would repeatedly crash until manual intervention fixes the underlying problem. We also observe that nodes are more prone to crashes on errors than corruptions.

We also observe that failed operations are rarely retried. While retries help in several cases where they are used, we observe that sometimes *indefinitely retrying operations may lead to more problems*. For instance, when ZooKeeper is unable to write new epoch information (to epoch_tmp) due to space errors, it deletes and creates a new file keeping the old file descriptor open. Since ZooKeeper blindly retries this sequence and given that space errors are sticky, the node soon runs out of descriptors and crashes, reducing availability.

#4: Redundancy is underutilized: A single fault can have disastrous clusterwide effects. Contrary to the widespread expectation that redundancy in distributed systems can help recover from single faults, we observe that even a single error or corruption can cause adverse clusterwide problems such as total unavailability, silent corruption, and loss or inaccessibility of inordinate amount of data. In many cases, almost all systems do not use redundancy as a source of recovery; they miss opportunities to use other intact replicas for recovering. Notice that all the bugs and undesirable behaviors that we discover in our study are due to injecting only a single fault in a single node at a time. Given that the data and functionality are replicated, ideally, none of the undesirable behaviors should arise.

A few systems (MongoDB and LogCabin) automatically recover from some (not all) data corruptions by utilizing other replicas. This recovery involves synergy between the local and the distributed recovery actions. Specifically, on encountering a corrupted entry, these systems locally ignore faulty data (local recovery policy). Then, the leader election algorithm ensures that the node where a data item has been corrupted and hence ignored does not become the leader (global recovery policy). As a result, the corrupted node eventually recovers the corrupted data by fetching it from the current leader. In many situations, even these systems do not automatically recover by utilizing redundancy. For instance, LogCabin and MongoDB simply crash when closed segments or collections are corrupted, respectively.

We also find that *an inordinate amount of data can be affected when only a small portion of data is faulty*. Table 3 shows different scopes that are affected when a small portion of the data is faulty. The affected portions can be silently lost or become inaccessible. For example, in Redis, all user data can become inaccessible when metadata in the append-only file is faulty or when there are read and write errors in append-only file data. Similarly, in Cassandra, an entire table can become inaccessible when small portions of data are faulty. Kafka can sometimes lose an entire log or all entries starting from the corrupted entry until the end of the log. RethinkDB loses all the data updated as part of a transaction when a small portion of it is corrupted or when the metablock pointing to that transaction is corrupted.

Table 3. Scope Affected

Structures	Fault Injected	Scope Affected
Redis:		
appendonlyfile.metadata	any	All[#]
appendonlyfile.userdata	read, write errors	All[#]
Cassandra:		
tablesst_data.block_0	corruptions (junk)	First Entry[$]
tablesst_index	corruptions	SSTable[#]
schemasst_compressioninfo	corruptions, read error	Table[#]
schemasst_filter	corruptions, read error	Table[#]
schemasst_statistics.0	corruptions, read error	Table[#]
Kafka:		
log.header	corruptions	Entire Log[$]
log.other	corruptions, read error	Entire Log[$*]
replication_checkpoint	corruptions, read error	All[$]
replication_checkpoint_tmp	write errors	All[#]
RethinkDB:		
db.transaction_head	corruptions	Transaction[$]
db.metablock	corruptions	Transaction[$]

[$], data loss; [#], inaccessible; [*], starting from corrupted entry.
The table shows the scope of data (third column) that becomes lost or inaccessible when only a small portion of data (first column) is faulty.

In summary, we find that redundancy is not effectively used as a source of recovery and the general expectation that redundancy can help availability of functionality and data is not a reality.

#5: Crash and corruption handling are entangled. We find that in many systems, the detection and recovery code does not try to distinguish two fundamentally distinct problems: *crashes* and *data corruption*.

Storage systems implement crash-consistent update protocols (i.e., even in the presence of crashes during an update, data should always be recoverable and should not be corrupted or lost) [7, 61, 62]. To do this, systems carefully order writes and use checksums to detect partially updated data or corruptions that can occur due to crashes. On detecting a checksum mismatch due to corruption, all systems invariably run the crash recovery code (even if the corruption was *not* actually due to crash but rather due to a real corruption in the storage stack), ultimately leading to undesirable effects such as data loss.

One typical example of this problem is RethinkDB. RethinkDB does not use application-level checksums to handle corruption. However, it does use checksums for its metablocks to recover from *crashes*. Whenever a metablock is corrupted, RethinkDB detects the mismatch in metablock checksum and invokes its crash recovery code. The crash recovery code believes that the system crashed when the last transaction was committing. Consequently, it rolls back the committed and already-acknowledged transaction, leading to a data loss.

We observe the same entanglement problem in other systems. For example, Figure 11 shows how crash and corruption handling are entangled in Kafka. As shown in the figure, all incoming messages are checksummed and appended to the message log. Figure 11(a) shows the case where a crash during the append of message 2 leaves that message partially updated, triggering a checksum mismatch during recovery. The recovery action that Kafka takes on a checksum mismatch

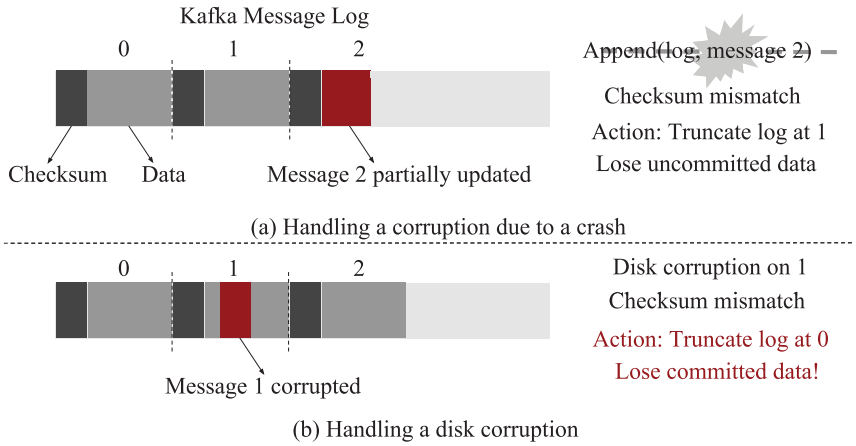


Fig. 11. Crash and corruption handling entanglement in Kafka. (a) How a crash during an update causes a checksum mismatch; in this case, the partially updated message is truncated. (b) The case where Kafka treats a disk corruption as a signal of a crash and truncates committed messages, leading to a data loss.

is to truncate the message whose checksum mismatches and *all subsequent messages*. In this case, the mismatch is caused due to a partial update; hence, it is safe to truncate the message because the client has not been acknowledged of the update. In contrast, consider the case shown in Figure 11(b); in this case, the second message has been successfully committed and the client has been acknowledged. Long after that, the disk block holding the first message gets corrupted, causing a checksum mismatch. However, the recovery code wrongly treats this corruption as a signal of a crash; hence, it truncates and loses committed messages 1 and 2.

Similarly, ZooKeeper, on detecting a data corruption in *log.transaction_tail*, concludes that the system crashed during the last transaction commit. On detecting this corruption, the node ignores the transaction and descends to become a follower. Eventually, the recovery protocol can recover the lost transaction from the leader. Even though this scenario does not lead to any global problems, it informs how crash and corruption handling are entangled.

LogCabin tries to distinguish crashes from corruption using the following logic: If a block in a closed segment (a segment that is full) is corrupted, it correctly flags that problem as a corruption and reacts by simply crashing. On the other hand, if a block in an open segment (still in use to persist transactions) is corrupted, it detects it as a crash and invokes its usual crash recovery procedure. MongoDB also differentiates corruptions in collections from journal corruptions in a similar fashion. Even systems that attempt to discern crashes from corruption do not always do so correctly.

There is an important consequence of entanglement of detection and recovery of crashes and corruptions. *During corruption (crash) recovery, some systems fetch an inordinate amount of data to fix the problem.* For instance, when a log entry is corrupted in LogCabin and MongoDB, they can fix the corrupted log by contacting other replicas. Unfortunately, they do so by ignoring the corrupted entry and all subsequent entries until the end of the log and subsequently fetching all the ignored data instead of simply fetching only the corrupted entry. Since a corruption is identified as a crash during the last committing transaction, these systems assume that the corrupted entry is the last entry in the log. Similarly, Kafka followers also fetch additional data from the leader instead of only the corrupted entry.

#6: Local fault handling and global protocols interact in unsafe ways. We find that local fault-handling behaviors and commonly used distributed protocols such as leader election, read repair [23], and re-synchronization interact in unsafe ways; such unsafe interaction leads to undesirable outcomes such as propagation of corruption or data loss.

For instance, in Kafka, the local fault-handling behavior on a corrupted node interacts unsafely with the leader election protocol, turning a local data loss on the node into a global data loss. Kafka maintains a piece of metadata called the *in-sync-replicas* (ISR); any node in this set contains all committed data and is eligible to become a leader. When a log entry is corrupted on a Kafka node, it ignores the current and all subsequent entries in the log and truncates the log until the last correct entry. Ideally, now this node should not be part of the ISR because it has lost some committed log entries. However, this node is not removed from the ISR and so can still become the leader. Thus, read requests issued to the leader result in a silent data loss. Furthermore, the leader also instructs the followers to truncate the log to match its log, which triggers an assertion on the followers, resulting in their crash. Thus, all future writes become unavailable (as shown in Figure 3(c)). The unsafe interaction between local behavior (i.e., to truncate the log) and the global protocol (leader election) in Kafka leads to a data loss and write unavailability. This behavior is in contrast with the leader election protocols of ZooKeeper, MongoDB, and LogCabin where a node that has truncated log entries does *not* become the leader.

Read-repair protocols are used in Dynamo-style quorum systems to fix any replica that has stale data. On a read request, the coordinator collects the digest of the data being read from a configured number of replicas. If all digests match, then the local data from the coordinator is simply returned. If the digests do not match, then an internal conflict-resolution policy is applied, and the resolved value is installed on replicas. In Cassandra, which implements read repair, the conflict resolution resolves to the lexically greater value; if the injected corrupted bytes are lexically greater than the original value, then the corrupted value is propagated to all other intact replicas.

Similarly, in Redis, when a data item is corrupted on the leader, it is not detected. Subsequently, the re-synchronization protocol propagates the corrupted data to the followers from the leader, overriding the correct version of data present on the followers.

4.3 Results Summary

We now summarize our behavior analysis results. Table 4(a) summarizes the catastrophic outcomes across all distributed storage systems that we studied. The table shows that redundancy does not provide fault tolerance in many systems: A single file-system fault on one node leads to catastrophic outcomes such as silent user-visible corruption, unavailability, data loss, query failures, or sometimes even the spread of corrupted data to other intact replicas. Ideally, none of these catastrophic outcomes should arise, since we inject only a single file-systems fault on a single node in the system at a time.

Table 4(b) shows the fundamental root causes in file-system fault handling that result in undesirable behaviors. As shown, these fundamental problems are common across all systems. First, in many systems, faults are often locally undetected. Even if faults are detected, the most common local reaction is to crash the node. All systems miss opportunities to use redundancy as a source of recovery from local file-system faults. We also find that crash and corruption handling are entangled in many systems. Finally, local fault-handling behaviors and global protocols interact in unsafe ways, leading to catastrophic outcomes.

4.4 File System Implications

All the bugs that we find can occur on XFS and all ext file systems including ext4, the default Linux file system. Given that these file systems are commonly used as local file systems in replicas

Table 4. Results Summary

Catastrophic Outcomes	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Silent Corruption	×		×		×			
Unavailability	×	×	×	×				×
Data Loss	×		×	×	×			×
Query Failures			×			×		×

(a) Outcomes Summary

Fundamental Problem	Redis	ZooKeeper	Cassandra	Kafka	RethinkDB	MongoDB	LogCabin	CockroachDB
Locally Undetected Faults	×	×	×		×			×
Crashing on Faults	×	×	×	×	×	×	×	×
Redundancy Underutilized	×	×	×	×	×	×	×	×
Crash Corruption Entangled		×		×	×	×	×	
Unsafe Protocol Interaction	×		×	×				

(b) Observations Summary

The table shows the summary of our results. (a) Shows the catastrophic outcomes caused by a single file-system fault across all systems we studied. A cross mark for a system denotes that we encountered at least one instance of the outcome specified on the left. (b) Shows the summary of fundamental problems observed across all systems. A cross mark for a system denotes that we observed at least one instance of the fundamental problem mentioned on the left.

of large distributed storage deployments and recommended by developers [51, 56, 64, 77], our findings have important implications for such real-world deployments.

File systems such as btrfs and ZFS employ checksums for user data; on detecting a corruption, they return an error instead of letting applications silently access corrupted data. Hence, bugs that occur due to an injected block corruption will not manifest on these file systems. We also find that applications that use end-to-end checksums when deployed on such file systems, surprisingly, lead to poor interactions. Specifically, applications crash more often due to errors than corruptions. In the case of corruption, a few applications (e.g., LogCabin, ZooKeeper) can use checksums and redundancy to recover, leading to correct behavior; however, when the corruption is transformed into an error, these applications crash, resulting in reduced availability.

4.5 Discussion

We now consider why distributed storage systems are not tolerant of single file-system faults. In a few systems (e.g., RethinkDB and Redis), we find that the primary reason is that they expect the underlying storage stack layers to reliably store data. As more deployments move to the cloud where reliable storage hardware, firmware, and software might not be the reality, storage systems need to start employing end-to-end integrity strategies.

Next, we believe that recovery code in distributed systems is not rigorously tested, contributing to undesirable behaviors. Although many systems employ checksums and other techniques, recovery code that exercises such machinery is not carefully tested. We suggest that future distributed systems need to rigorously test failure recovery code using fault injection frameworks such as ours.

Third, although a body of research work [25, 80, 84, 85, 95] and enterprise storage systems [50, 57, 58] provide software guidelines to tackle partial faults, such wisdom has not filtered down to commodity distributed storage systems. Our findings provide motivation for distributed systems to build on existing research work to tolerate practical faults other than crashes [17, 45, 98].

Finally, although redundancy is effectively used to provide improved availability, it remains underutilized as a source of recovery from file-system and other partial faults. To effectively use redundancy, first, the on-disk data structures have to be carefully designed so corrupted or inaccessible parts of data can be identified. Next, corruption recovery has to be decoupled from crash recovery to fix only the corrupted or inaccessible portions of data. Sometimes, recovering the corrupted data might be impossible if the intact replicas are not reachable. In such cases, the outcome should be defined by design rather than left as an implementation detail.

We contacted the developers of the systems regarding the behaviors we found. RethinkDB and Redis rely on the underlying storage layers to ensure data integrity [68, 69]. The RethinkDB developers intend to change the design to include application-level checksums in the future and have updated the documentation to reflect the bugs we reported [71, 72] until this is fixed. They also confirmed the entanglement between corruption and crash handling [74].

The write-unavailability bug in ZooKeeper discovered by CORDS was encountered by real-world users and has been fixed [100, 102]. The ZooKeeper developers mentioned that crashing on detecting corruption was not a conscious design decision [101]. The LogCabin developers also confirmed the entanglement between corruption and crash handling in open segments; they added that it is hard to distinguish a partial write from corruption in open segments [47]. The developers of CockroachDB and Kafka have also responded to our bug reports [15, 16, 40].

5 RELATED WORK

Our work draws inspiration from several bodies of prior research on file-system faults, fault injection, reliability testing, and bug studies. We now describe how our work relates to each of these avenues.

Corruption and errors in storage stack. Several past studies have analyzed storage errors and corruption in detail [8, 9, 49, 55, 80, 82]. Specifically, Bairavasundaram et al. and Schroeder et al. show the prevalence of latent sector errors in disks. Similarly, studies have also shown the prevalence of data corruption in the real world [8, 60]. Furthermore, studies have shown that cheap near-line disks are more prone to errors and corruption than enterprise-class devices [7]. Since large-scale deployments often tend to use cheap hardware (and build reliability into the software), problems such as disk errors and corruption are increasingly important. These prior studies motivated us to study the effects of such faults in distributed storage systems.

Generic fault injection. Our work is related to efforts that inject faults into systems and test their robustness [11, 33, 83, 90]. Several efforts have built generic fault injectors for distributed systems [21, 37, 87]. Most of these fault-injection frameworks aim to inject various types of faults and also emphasize the portability of the framework to several platforms and systems. For example, Han et al. described Doctor [37], a comprehensive framework that can inject processor, memory, and communication faults. CORDS differs from generic fault injectors through its specific focus on file-system faults.

File-system fault injection studies. A few studies have shown how file systems such as ext3, IBM JFS, ReiserFS, ZFS, and so on, react specifically to storage and memory faults [10, 63, 99]. These studies carefully inject disk faults just beneath the file system and observe how the file system reacts to the fault. The injection methodology used in these studies is type-aware: Faults are not injected at random; rather, they are injected into various specific on-disk structures of the file system. Type-aware fault injection helps in quickly exercising several file-system code paths compared to random fault injection techniques. The fault-injection methodology in CORDS is similar to such type-aware injectors. The results from these studies show that some file systems (such as ext3) simply propagate corruption to applications, since they do not employ checksums for user data. The results also show that some file systems (such as ZFS) use checksums for user data and hence transform an underlying corruption into read errors. These results imply that applications that desire to maintain data integrity have to handle such situations, motivating our study.

Application fault injection studies. A few studies [88, 98] have shown how applications running atop file systems react to file-system faults. For example, Subramanian et al. study how the MySql database engine reacts to disk corruption. Similarly, Zhang et al. study how file synchronization services (such as Dropbox) react in the presence of local file-system corruption. However, none of the studies examined modern distributed storage systems. Our work focuses on testing the behavior of distributed systems under storage faults. We believe our work is the first to comprehensively examine the effects of storage faults across many distributed storage systems.

Testing Distributed Systems. Several model checkers have succeeded in uncovering bugs in distributed systems [35, 44, 96]. CORDS exposes bugs that cannot be discovered by model checkers. Model checkers typically reorder network messages and inject crashes to find bugs; they do not inject storage-related faults. Similarly to model checkers, tools such as Jepsen [43] that test distributed systems under faulty networks are complementary to CORDS. Our previous work [3] studies how file-system crash behaviors affect distributed systems. However, these faults occur only on a crash, unlike block corruption and errors introduced by CORDS. Our targeted fault-injection framework can examine large storage systems faster than model checkers, which typically suffer from state-space explosion.

Bug Studies. A few recent bug studies [34, 97] have given insights into common problems found in distributed systems. Yuan et al. show that 34% of catastrophic failures in their study are due to unanticipated error conditions. Our results also show that systems do not handle read and write errors well; this poor error handling leads to harmful global effects in many cases. We believe that bug studies and fault injection studies are complementary to each other; while bug studies suggest constructing test cases by examining sequences of events that have led to bugs encountered in the wild, fault injection studies like ours concentrate on injecting one type of fault and uncovering new bugs and design flaws.

6 CONCLUSIONS

We show that tolerance of file-system faults is not ingrained in modern distributed storage systems. These systems are not equipped to effectively use redundancy across replicas to recover from local file-system faults; user-visible problems such as data loss, corruption, and unavailability can manifest due to a single local file-system fault. As distributed storage systems are emerging as the primary choice for storing critical user data, carefully designing them for all types of faults is important. Our study is a step in this direction and we hope our work will lead to more work on building next generation fault-resilient distributed systems.

ACKNOWLEDGMENTS

We thank the anonymous reviewers of FAST'17 for their insightful comments. We thank the members of the ADSL and the developers of CockroachDB, LogCabin, Redis, RethinkDB, and ZooKeeper for their valuable discussions. Finally, we thank CloudLab [75] for providing a great environment for running our experiments.

REFERENCES

[1] Cords Tool and Results. 2017. Retrieved from http://research.cs.wisc.edu/adsl/Software/cords/.
[2] Ramnatthan Alagappan, Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Aws Albarghouthi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2015. Beyond storage APIs: Provable semantics for storage stacks. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*.
[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. Cassandra. Retrieved from http://cassandra.apache.org/.
[5] Apache. Kakfa. Retrieved from http://kafka.apache.org/.
[6] Apache. ZooKeeper. Retrieved from https://zookeeper.apache.org/.
[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. 2015. *Operating Systems: Three Easy Pieces* (0.91 ed.). Arpaci-Dusseau Books.
[8] 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)*.
[9] 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)*.
[10] 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)*.
[11] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. 1990. Fault injection experiments using FIAT. *IEEE Trans. Comput.* 39, 4 (April 1990), 575–582.
[12] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. 2016. *Disks for Data Centers*. Technical Report. Google.
[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating system errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*.
[14] CockroachDB. CockroachDB. Retrieved from https://www.cockroachlabs.com/.
[15] CockroachDB. Disk corruptions and read/write error handling in CockroachDB. Retrieved from https://forum.cockroachlabs.com/t/disk-corruptions-and-read-write-error-handling-in-cockroachdb/258.
[16] CockroachDB. Resiliency to disk corruption and storage errors. Retrieved from https://github.com/cockroachdb/cockroach/issues/7882.
[17] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. 2012. Practical hardening of crash-tolerant systems. In *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*.
[18] Data Center Knowledge. Ma.gnolia data is gone for good. Retrieved from http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/.
[19] Datastax. Netflix Cassandra Use Case. Retrieved from http://www.datastax.com/resources/casestudies/netflix.
[20] DataStax. Read Repair: Repair during Read Path. Retrieved from http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html.
[21] S. Dawson, F. Jahanian, and T. Mitton. 1996. ORCHESTRA: A probing and fault injection environment for testing protocol implementations. In *Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS'96)*.
[22] Jeff Dean. Building Large-Scale Internet Services. Retrieved from http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf.
[23] 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)*. Stevenson, WA.
[24] Jon Elerath. 2009. Hard-disk drives: The good, the bad, and the ugly. *Commun. ACM* 52, 6 (June 2009), 38–45.

[25] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 2012. Detection and correction of silent data corruption for large-scale high-performance computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12)*.

[26] 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)*.

[27] 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)*.

[28] FUSE. Linux FUSE (Filesystem in Userspace) interface. Retrieved from https://github.com/libfuse/libfuse.

[29] 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 Symposium on File and Storage Technologies (FAST'17)*. 149–166.

[30] 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)*.

[31] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. 2011. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*.

[32] Jim Gray. 1985. *Why Do Computers Stop and What Can Be Done About It?* Technical Report PN87614. Tandem.

[33] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. 2003. Characterization of linux kernel behavior under errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*.

[34] Haryadi S. Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J. Eliazar, Agung Laksono, Jeffrey F. Lukman, Vincentius Martin, and Anang D. Satria. 2014. What bugs live in the cloud? A study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'14)*.

[35] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. 2011. Practical software model checking via dynamic interface reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*.

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

[37] Seungjae Han, Kang G. Shin, and Harold A. Rosenberg. 1995. DOCTOR: An integrated software fault injection environment for distributed real-time systems. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS'95)*.

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

[39] Jerome Verstrynge. Timestamps in Cassandra. Retrieved from http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm.

[40] Kafka. Data corruption or EIO leads to data loss. https://issues.apache.org/jira/browse/KAFKA-4009.

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

[42] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrisha Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. 2000. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'00)*.

[43] Kyle Kingsbury. Jepsen. Retrieved from http://jepsen.io/.

[44] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 2014. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[45] 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)*.

[46] LogCabin. LogCabin. Retrieved from https://github.com/logcabin/logcabin.

[47] LogCabin. Reaction to disk errors and corruptions. Retrieved from https://groups.google.com/forum/#!topic/logcabin-dev/wqNcdj0IHe4.

[48] Mark Adler. Adler32 Collisions. Retrieved from http://stackoverflow.com/questions/13455067/horrific-collisions-of-adler32-hash.

[49] 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)*.

[50] Ningfang Mi, A. Riska, E. Smirni, and E. Riedel. 2008. Enhancing data availability in disk drives through background activities. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'08)*, Anchorage, Alaska.
[51] Michael Rubin. Google moves from ext2 to ext4. Retrieved from http://lists.openwall.net/linux-ext4/2010/01/04/8.
[52] MongoDB. MongoDB. Retrieved from https://www.mongodb.org/.
[53] MongoDB. MongoDB at eBay. Retrieved from https://www.mongodb.com/presentations/mongodb-ebay.
[54] MongoDB. MongoDB WiredTiger. Retrieved from https://docs.mongodb.org/manual/core/wiredtiger/.
[55] 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)*.
[56] Netflix. Cassandra at Netflix. Retrieved from http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html.
[57] Oracle. Fusion-IO Data Integrity. Retrieved from https://blogs.oracle.com/linux/entry/fusion_io_showcases_data_integrity.
[58] Oracle. Preventing Data Corruptions with HARD. Retrieved from http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm.
[59] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. 1996. The log-structured merge-tree (LSM-tree). *Acta Inform.* 33, 4 (1996).
[60] Bernd Panzer-Steindel. 2007. Data integrity. *CERN/IT* (2007).
[61] 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)*.
[62] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. Model-based failure analysis of journaling file systems. In *The Proceedings of the International Conference on Dependable Systems and Networks (DSN'05)*.
[63] 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)*.
[64] Rahul Bhartia. MongoDB on AWS Guidelines and Best Practices. Retrieved from http://media.amazonwebservices.com/AWS_NoSQL_MongoDB.pdf.
[65] Redis. Instagram Architecture. Retrieved from http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html.
[66] Redis. Redis. Retrieved from http://redis.io/.
[67] Redis. Redis at Flickr. Retrieved from http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/.
[68] Redis. Silent data corruption in Redis. Retrieved from https://github.com/antirez/redis/issues/3730.
[69] RethinkDB. Integrity of read results. Retrieved from https://github.com/rethinkdb/rethinkdb/issues/5925.
[70] RethinkDB. RethinkDB. Retrieved from https://www.rethinkdb.com/.
[71] RethinkDB. RethinkDB Data Storage. Retrieved from https://www.rethinkdb.com/docs/architecture/#data-storage.
[72] RethinkDB. RethinkDB Doc Issues. Retrieved from https://github.com/rethinkdb/docs/issues/1167.
[73] RethinkDB. RethinkDB Faq. Retrieved from https://www.rethinkdb.com/faq/.
[74] RethinkDB. Silent data loss on metablock corruptions. Retrieved from https://github.com/rethinkdb/rethinkdb/issues/6034.
[75] Robert Ricci, Eric Eide, and CloudLab Team. 2014. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:* 39, 6 (2014).
[76] Robert Harris. Data corruption is worse than you know. Retrieved from http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/.
[77] Ron Kuris. Cassandra From tarball to production. Retrieved from http://www.slideshare.net/planetcassandra/cassandra-from-tarball-to-production-2.
[78] Mendel Rosenblum and John Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (February 1992), 26–52.
[79] J. H. Saltzer, D. P. Reed, and D. D. Clark. 1984. End-to-end arguments in system design. *ACM Trans. Comput. Syst.* 2, 4 (1984), 277–288.
[80] 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)*.
[81] Bianca Schroeder and Garth A. Gibson. 2007. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the 5th USENIX Symposium on File and Storage Technologies (FAST'07)*.

[82] 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)*.

[83] D. P. Siewiorek, J. J. Hudak, B. H. Suh, and Z. Z. Segal. 1993. Development of a benchmark to measure system robustness. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS'23)*.

[84] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. 2005. Ensuring data integrity in storage: Techniques and applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS'05)*.

[85] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. 1999. Dealing with server corruption in weakly consistent replicated data systems. *Wirel. Netw.* 5, 5 (October 1999), 357–371.

[86] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

[87] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 2000. A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the 4th International Computer Performance and Dependability Symposium (IPDS'00)*.

[88] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Jeffrey F. Naughton. 2010. Impact of disk corruption on open-source DBMS. In *Proceedings of the 26th International Conference on Data Engineering (ICDE'10)*.

[89] 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)*.

[90] T. K. Tsai and R. K. Iyer. 1995. Measuring fault tolerance with the FTAPE fault injection tool. In *Proceedings of the 8th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation: Quantitative Evaluation of Computing and Communication Systems (MMB'95)*.

[91] Twitter. Kafka at Twitter. Retrieved from https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time.

[92] Uber. The Uber Engineering Tech Stack, Part I: The Foundation. Retrieved from https://eng.uber.com/tech-stack-part-one/.

[93] Uber. The Uber Engineering Tech Stack, Part II: The Edge And Beyond. Retrieved from https://eng.uber.com/tech-stack-part-two/.

[94] Voldemort. Project Voldemort. http://www.project-voldemort.com/voldemort/.

[95] 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)*.

[96] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. 2009. MODIST: Transparent model checking of unmodified distributed systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI'09)*.

[97] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 2014. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI'14)*.

[98] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. 2014. ViewBox: Integrating local file systems with cloud storage services. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST'14)*.

[99] 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)*.

[100] ZooKeeper. Cluster unavailable on space and write errors. Retrieved from https://issues.apache.org/jira/browse/ZOOKEEPER-2495.

[101] ZooKeeper. Crash on detecting a corruption. Retrieved from http://mail-archives.apache.org/mod_mbox/zookeeper-dev/201701.mbox/browser.

[102] ZooKeeper. Zookeeper service becomes unavailable when leader fails to write transaction log. Retrieved from https://issues.apache.org/jira/browse/ZOOKEEPER-2247.

Received June 2017; accepted July 2017