Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions

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

https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

This paper is included in the Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17).

February 27–March 2, 2017 • Santa Clara, CA, USA

ISBN 978-1-931971-36-2

Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions

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

Abstract

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 bugs 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. Our results have implications for the design of next generation fault-tolerant distributed and cloud storage systems.

1 Introduction

Cloud-based applications such as Internet search, photo and video services [19, 65, 67], social networking [90, 93], transportation services [91, 92], and e-commerce [52] 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), document stores (e.g., MongoDB), column stores (e.g., Cassandra), messaging queues (e.g., Kafka), and databases (e.g., RethinkDB).

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, 30, 31, 40, 80]. 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, 48, 54, 79, 81] or worse, data can be silently corrupted [8, 60, 85]. These complex failures are known as partial storage faults [63].

Previous studies [10, 63, 98] 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, 35, 41, 81]. 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 is redundantly stored on many nodes. Given this expectation, in this paper, 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 which 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 which 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 [51], LogCabin [45], 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 untimely 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.

Second, our study has enabled us to make several observations across all systems concerning file-system fault handling. Specifically, we first have found that *systems employ diverse data-integrity strategies*; while some systems carefully use checksums, others completely trust lower layers in the stack to detect and handle corruption. Second, *faults are often undetected locally, and even if detected, crashing is the most common reaction*; undetected faults on one node can lead to harmful global effects such as user-visible data corruption. Third, as mentioned above, *a single fault can have disastrous cluster-wide effects*. Although distributed storage systems replicate data and functionality across many nodes, a single file-system fault on a single node can result in harmful cluster-wide effects; surprisingly, many distributed storage systems do not consistently use redundancy as a source of recovery. Fourth, *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. Finally, *nuances in commonly used distributed protocols can spread corruption or data loss*; for example, we find that subtleties in the implementation of distributed protocols such as leader election, read-repair, and re-synchronization can propagate corruption or data loss.

This paper contains three major contributions. First, we build a fault injection framework (CORDS) to carefully inject file-system faults into applications (§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(§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 observations across all systems showing some of the common data integrity and error handling problems (§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 paper 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 (§2). Then, we describe our fault model and how our framework injects faults and observes behaviors (§3). Next, we present our behavior analysis and observations across systems (§4). Finally, we discuss related work (§5) and conclude (§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, 48, 54, 63, 79, 81]. Sometimes, corruptions can arise due to software bugs in other parts of the operating system [13], device drivers [88], 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 upon detecting some problem with the block being accessed (such as in-disk ECC complaining that the block has a bit rot) [9, 79]. 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 [48, 54, 81].

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, 37, 75]. 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 [98]. 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, 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 [78]. 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 [29]. 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.

Type of Fault		**Op**	**Example Causes**
Corruption	zeros, junk	Read	misdirected and lost writes in *ext* and *XFS*
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

Table 1: **Possible Faults and Example Causes**. *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.*

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 storage 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.

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* 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 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.

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 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

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* which 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*.

errfs can inject two types of corruptions: corrupted with *zeros* or *junk*. For corruptions, *errfs* performs the read and changes the contents of the block that is marked for corruption, before returning to the application. *errfs* can inject three types of 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.

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*
- *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, 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.

In most cases, a node locally reacts to an injected fault. As shown in the legend of Figure 1, 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 as three.

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

4.1 System Behavior Analysis

Figure 1 shows the behaviors for all systems when faults are injected into different on-disk structures. The on-disk structure names shown on the right take the form: *file_name.logical_entity*. We derive the logical entity name from our understanding of the on-disk format of the file. If a file can be contained in a single file-system block, we do not show the logical entity name.

Interpreting Figure 1: We guide the reader to relevant portions of the figure for a few structures for one system (Redis). 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 Redis). If the leader crashes, then the cluster becomes unavailable and if the followers crash, the cluster runs with reduced redundancy (first row of global effect for both workloads). *Corruptions* in user data in the appendonly file are undetected (second row of local behavior for both workloads). If the leader is corrupted, it leads to a global user-visible corruption, and if the followers are corrupted, there is no harmful global effect (second row of global effect for read workload). In contrast, *errors* in appendonly file user data lead to crashes (second row of local behavior for both workloads); crashes of leader and followers lead to cluster unavailability and reduced redundancy, respectively (second row of global effect for both workloads).

We next qualitatively summarize the results in Figure 1 for each system.

4.1.1 Redis

Redis is a popular data structure store, used as database, cache, and message broker. Redis uses a simple appendonly file (*aof*) to log user data. 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. Redis does not elect a leader automatically when the current leader fails.

Summary and Bugs: Redis does not use checksums for *aof* user data; thus, it does not detect corruptions. Figure 2(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. If the followers are corrupted, the same protocol unintentionally fixes the corruption by fetching the data from the leader. Corruptions in metadata structures in *aof* and errors in *aof* in leader causes it to crash, making the cluster unavailable. Since the leader sends the *rdb* file during re-synchronization, corruption in the same causes both the followers to crash. These crashes ultimately make the cluster unavailable for writes.

4.1.2 ZooKeeper

ZooKeeper is a popular service for storing configuration information, naming, and distributed synchronization. It uses *log* files to append user data; the first block of the log contains a header, the second contains the transaction body, and the third contains the transaction tail along with ACLs and other information.

Summary and Bugs: ZooKeeper can detect corruptions in the log using checksums but reacts by simply crash-

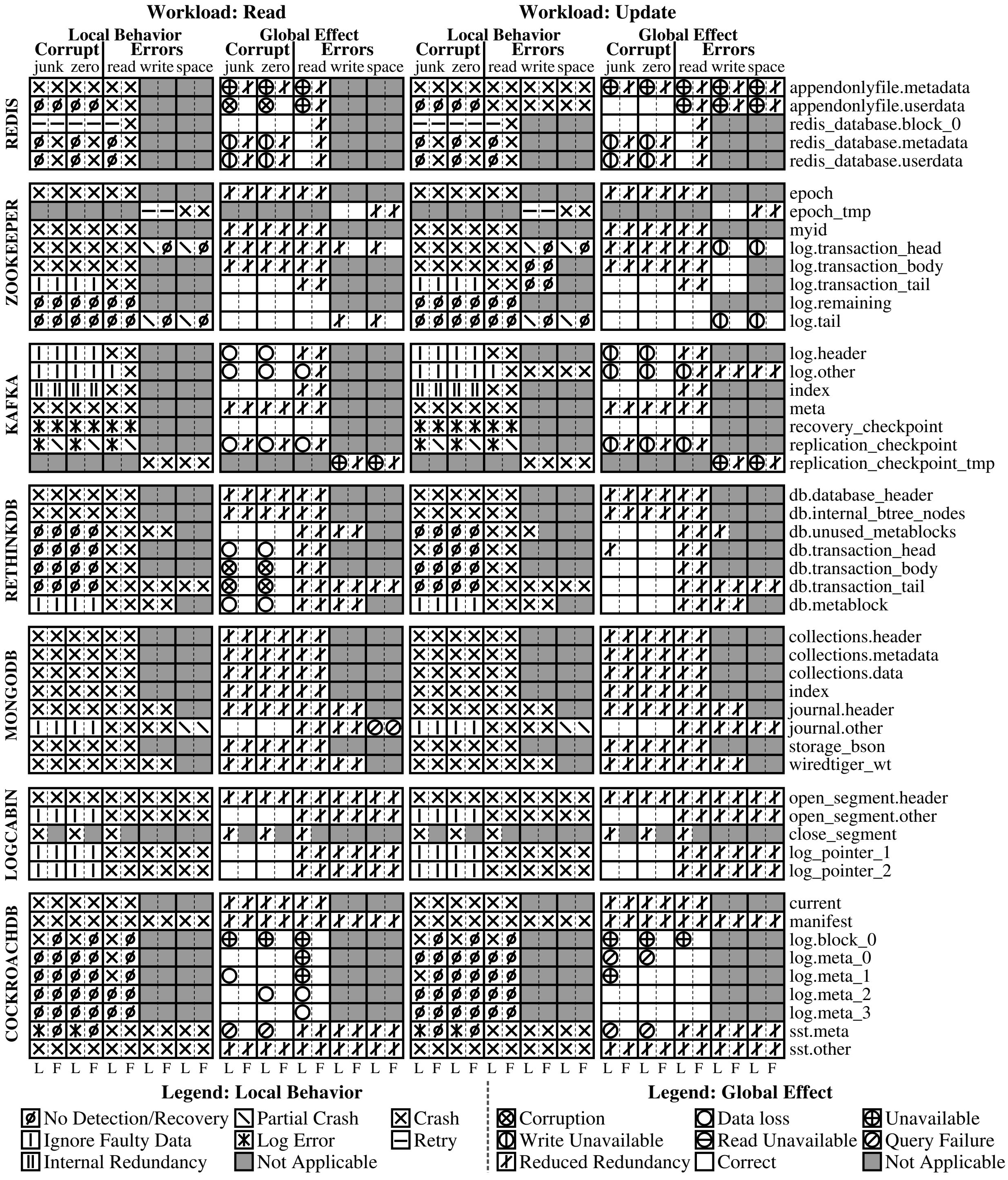
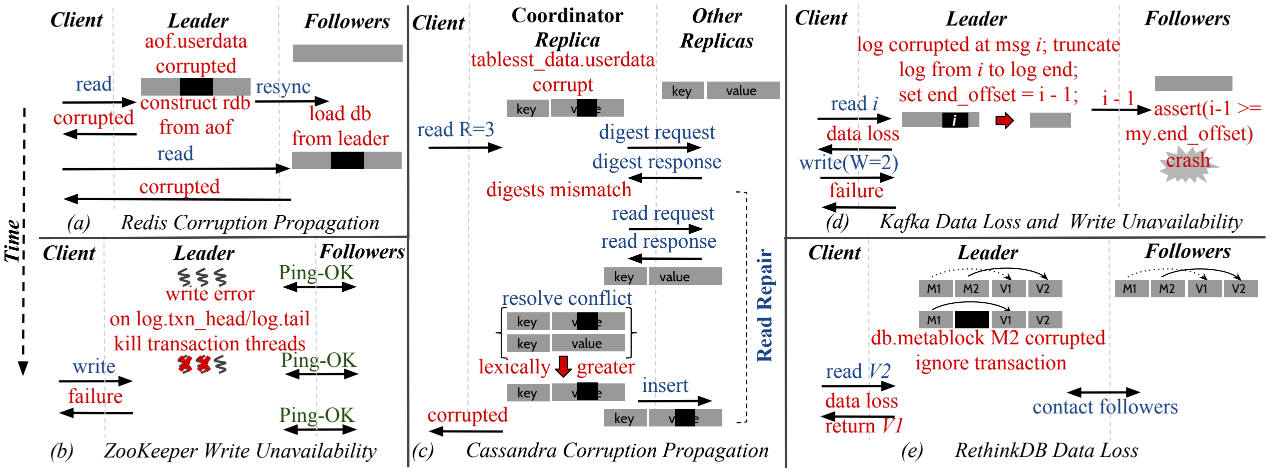


Figure 1: **System Behaviors.** *The figure shows system behaviors when corruptions (corrupted with either junk or zeros), read errors, write errors, and space errors are injected in various on-disk logical structures. The leftmost label shows the system name. Within each system workload (read and update), there are two boxes – first, local behavior of the node where the fault is injected and second, cluster-wide global effect of the injected fault. The rightmost annotation shows the on-disk logical structure in which the fault is injected. It takes the following form: file_name.logical_entity. If a file can be contained in a single file-system block, we do not show the logical entity name. 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 the epoch structure in ZooKeeper as it is not written and hence shown as a gray box.*



(a) Redis Corruption Propagation

(b) ZooKeeper Write Unavailability

(c) Cassandra Corruption Propagation

(d) Kafka Data Loss and Write Unavailability

(e) RethinkDB Data Loss

Figure 2: **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.*

ing. 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; the leader election protocol prevents that node from becoming the leader, avoiding undesirable behaviors. Eventually, the corrupted node repairs its log by contacting the leader, leading to correct behavior.

Unfortunately, ZooKeeper does not recover from write errors to the transaction head and log tail (Figure 1 – rows four and eight in ZooKeeper). Figure 2(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.

4.1.3 Cassandra

Cassandra is a Dynamo-like [23] NoSQL store. Both user data tables (tablesst) and system schema (schemasst) are stored using a variation of Log Structured Merge Trees [59]. Unlike other systems we study, Cassandra does not have a leader and followers; instead, the nodes form a ring. Hence, we show its behaviors separately in Figure 3.

Summary and Bugs: Cassandra enables checksum verification on user data only as a side effect of enabling compression. When compression is turned off, corruptions are not detected on user data (tablesst_data). On a read query, a coordinator node collects and compares digests (hash) of the data from R replicas [20]. If the digests mismatch, conflicts in the values are resolved using a *latest timestamp wins* policy. If there is a tie between timestamps, the lexically greatest value is chosen and installed on other replicas [38]. As shown in Figure 2(c), on $R = 3$, if the corrupted value is lexically greater than the original value, the corrupted value is returned to the user and the corruption is propagated to other intact replicas. On the other hand, if the corrupted value is lexically lesser, it fixes the corrupted node. Reads to a corrupted node with $R = 1$ always return corrupted data.

Faults in tablesst_index cause query failures. Faults in schema data and schema index cause the node to crash, making it unavailable for reads and writes with $R = 3$ and $W = 3$, respectively. Faults in other schema files result in query failure. In most cases, user-visible problems that are observed in $R = 1$ configuration are not fixed even when run with $R = 3$.

4.1.4 Kafka

Kafka is a distributed persistent message queue in which clients can publish and subscribe for messages. It uses a *log* to append new messages and each message is checksummed. It maintains an *index* file which indexes messages to byte offsets within the log. The *replication_checkpoint* and *recovery_checkpoint* indicate how many messages are replicated to followers so far and how many messages are flushed to disk so far, respectively.

Summary and Bugs: On read and write errors, Kafka mostly crashes. Figure 2(d) shows the scenario where Kafka can lose data and become unavailable for writes. When a log entry is corrupted on the leader (Figure 1 – rows one and two in Kafka), it locally ignores that entry and all subsequent entries in the log. The leader then instructs the followers to do the same. On receiving this instruction, the followers hit a fatal assertion and simply crash. Once the followers crash, the cluster becomes unavailable for writes and the data is also lost. Corruption in index is fixed using internal redundancy. Faults in the *replication_checkpoint* of the leader results in a data loss as the leader is unable to record the replication offsets

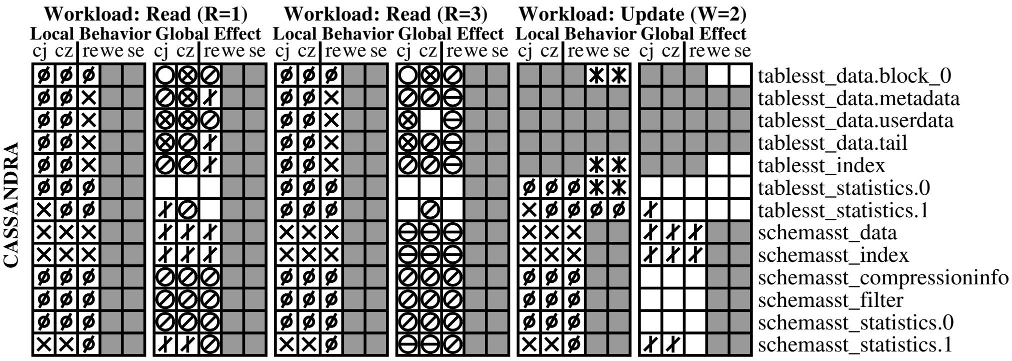


Figure 3: **System Behavior: Cassandra.** *The figure shows system behaviors when corruptions (corrupted with either junk (cj) or zeros(cz)), read errors (re), write errors (we), and space errors (se) are injected in various on-disk logical structures for Cassandra. The legend for local behaviors and global effects is the same as shown in Figure 1.*

of the followers. Kafka becomes unavailable when the leader cannot read or write *replication_checkpoint* and *replication_checkpoint_tmp*, respectively.

4.1.5 RethinkDB

RethinkDB is a distributed database suited for pushing query results to real-time web applications. It uses a persistent B-tree to store all data. *metablocks* in the B-tree point to the data blocks that constitute the current and the previous version of the database. During an update, new data blocks are carefully first written and then the *metablock* with checksums is updated to point to the new blocks, thus enabling atomic updates.

Summary and Bugs: On any fault in database header and internal B-tree nodes, RethinkDB simply crashes. If the leader crashes, 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 transaction body and tail (Figure 1 – rows five and six in RethinkDB). When these blocks of the leader are corrupted, RethinkDB silently returns corrupted data.

Figure 2(e) depicts how data is silently lost when the transaction head or the metablock pointing to the transaction is corrupted on the leader. Even though there are intact copies of the same data on the followers, the leader does not fix its corrupted or lost data, even when we perform the reads with *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 replicated document store that uses WiredTiger [53] underneath for storage. When an item is inserted or updated, it is added to the *journal* first; then, it is checkpointed to the *collections* file.

Summary and Bugs: MongoDB simply crashes on most errors, leading to reduced redundancy. A new leader 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. One exception to the above is when blocks other than journal header are corrupted (Figure 1 – the sixth row in MongoDB). In this case, MongoDB detects and ignores the corrupted blocks; 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.

4.1.7 LogCabin

LogCabin uses the Raft consensus protocol [56] to provide a replicated and consistent data store for other systems to store their core metadata. It implements a segmented-log [77] and each segment is a file on the file system. When the current *open* segment is fully utilized, it is *closed* and a new segment is opened. Two pointer files point to the latest two versions of the log. They are updated alternately; when a pointer file is partially updated, LogCabin uses the other pointer file that points to a slightly older but consistent version of the log.

Summary and Bugs: LogCabin crashes on all read, write, and space errors. Similarly, if an *open* segment file header or blocks in a closed segment 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 (Figure 1 – second row in LogCabin). If a log pointer file is corrupted, LogCabin ignores that pointer file and uses the other pointer file.

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.

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

Table 2: **Data Integrity Strategies.** *The table shows techniques employed by modern systems to ensure data integrity of user-level application data.*

4.1.8 CockroachDB

CockroachDB is a SQL database built to survive disk, machine, and data-center failures. It uses a tuned version of RocksDB underneath for storage; the storage engine is an LSM tree that appends incoming data to a persistent *log*; the in-memory data is then periodically compacted to create the *sst* files. The *manifest* and the *current* files point to the current version of the database.

Summary and Bugs: Most of the time, CockroachDB simply crashes on corruptions and errors on any data structure, resulting in reduced redundancy. Faults in the log file on the leader can sometimes lead to total cluster unavailability as some followers also crash following the crash of the leader. Corruptions and errors in a few other log metadata can cause a data loss where CockroachDB silently returns zero rows. Corruptions in sst files and few blocks of log metadata cause queries to fail 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 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.

#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 upon 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 [47]. 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 unusual 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: Local Behavior: Faults are often undetected; even if detected, crashing is the most common local reaction. We find that faults are often locally undetected. Sometimes, this leads to an immediate harmful global effect. For instance, in Redis, corruptions in the appendonly 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, causes 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.

We observe that *crashing is the most common local reaction to faults*. When systems detect corruption or encounter an error, they simply crash, as is evident from the abundance of crash symbols in local behaviors of Figure 1. 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

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

Table 3: **Scope Affected.** *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.*

also observe that nodes are more prone to crashes on errors than corruptions.

We 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.

#3: Redundancy is underutilized: A single fault can have disastrous cluster-wide 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 cluster-wide problems such as total unavailability, silent corruption, and loss or inaccessibility of inordinate amount of data. Almost all systems in many cases do not use redundancy as a source of recovery and miss opportunities of using 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 manifest.

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 segment 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 of user data can become inaccessible when metadata in the appendonly file is faulty or when there are read and write errors in appendonly 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.

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.

#4: Crash and corruption handling are entangled. We find that detection and recovery code of many systems often inadvertently try to detect and fix 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 corrupt 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. Similarly, when the log is corrupted in Kafka, the recovery code treats the corruption as a signal of a crash; hence, it truncates and loses all further data in the log instead of fixing only the corrupted entry. The underlying reason for this problem is the inability to differentiate corruptions due to crashes from real storage stack corruptions.

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 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.

#5: Nuances in commonly used distributed protocols can spread corruption or data loss. We find that subtleties in the implementation of commonly used distributed protocols such as leader election, read-repair [23], and re-synchronization can propagate corruption or data loss.

For instance, in Kafka, a local data loss in one node can lead to a global data loss due to the subtleties in its leader election protocol. Kafka maintains a set of *in-sync-replicas* (ISR) and any node in this set can become the 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. Logically, now this node should not be part of the ISR as it has lost some log entries. However, this node is not removed from the ISR and so eventually can still become the leader and silently lose data. This behavior is in contrast with leader election protocols of ZooKeeper, MongoDB, and LogCabin where a node that has ignored log entries do *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, 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, 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 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 of large distributed storage deployments and recommended by developers [50, 55, 64, 76], 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 a correct behavior; however, when the corruption is transformed into an error, these applications crash, resulting in reduced availability.

4.4 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 advocate 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, 79, 83, 84, 94] and enterprise storage systems [49, 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 practically tolerate faults other than crashes [17, 44, 97].

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 that 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 developers of the systems regarding the behaviors we found. RethinkDB and Redis rely on the underlying storage layers to ensure data integrity [68, 69]. RethinkDB intends to change the design to include application-level checksums in the future and updated the documentation to reflect the bugs we reported [71, 72] until this is fixed. They also confirmed the entanglement in corruption and crash handling [73].

The write unavailability bug in ZooKeeper discovered by CORDS was encountered by real-world users and has been fixed recently [99, 101]. ZooKeeper developers mentioned that crashing on detecting corruption was not a conscious design decision [100]. LogCabin developers also confirmed the entanglement in corruption and crash handling in open segments; they added that it is hard to distinguish a partial write from corruption in open segments [46]. Developers of CockroachDB and Kafka have also responded to our bug reports [15, 16, 39].

5 Related Work

Our work builds on four bodies of related work.

Corruptions and errors in storage stack: As discussed in §2, detailed studies on storage errors and corruptions [8, 9, 48, 54, 79, 81] motivated our work.

Fault injection: Our work is related to efforts that inject faults into systems and test their robustness [11, 32, 82, 89]. Several efforts have built generic fault injectors for distributed systems [21, 36, 86]. A few studies have shown how file systems [10, 63, 98] and applications running atop them [87, 97] react specifically to storage and memory faults. Our work draws from both bodies of work but is unique in its focus on testing behaviors of distributed systems to 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 distributed model checkers have succeeded in uncovering bugs in distributed systems [34, 43, 95]. 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. Similar to model checkers, tools such as Jepsen [42] 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.

Bug Studies: A few recent bug studies [33, 96] 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 to 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 testing 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 and Hakim Weatherspoon (our shepherd) 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. 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. Finally, we thank CloudLab [74] for providing a great environment for running our experiments. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and may not reflect the views of NSF, DOE, or other institutions.

References

[1] Cords Tool and Results. 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. Beyond Storage APIs: Provable Semantics for Storage Stacks. In *Proceedings of the 15th USENIX Conference on Hot Topics in Operating Systems (HOTOS'15)*, Kartause Ittingen, Switzerland, May 2015.

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

[4] Apache. Cassandra. http://cassandra.apache.org/.

[5] Apache. Kakfa. http://kafka.apache.org/.

[6] Apache. ZooKeeper. https://zookeeper.apache.org/.

[7] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[8] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[9] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 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)*, San Diego, CA, June 2007.

[10] Lakshmi N. Bairavasundaram, Meenali Rungta, Nitin Agrawal, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Michael M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[11] J.H. Barton, E.W. Czeck, Z.Z. Segall, and D.P. Siewiorek. Fault Injection Experiments Using FIAT. *IEEE Transactions on Computers*, 39(4), April 1990.

[12] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for Data Centers. Technical report, Google, 2016.

[13] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An Empirical Study of Operating System Errors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, October 2001.

[14] CockroachDB. CockroachDB. https://www.cockroachlabs.com/.

[15] CockroachDB. Disk corruptions and read-/write error handling in CockroachDB. https://forum.cockroachlabs.com/t/disk-corruptions-and-read-write-error-handling-in-cockroachdb/258.

[16] CockroachDB. Resiliency to disk corruption and storage errors. https://github.com/cockroachdb/cockroach/issues/7882.

[17] Miguel Correia, Daniel Gómez Ferro, Flavio P. Junqueira, and Marco Serafini. Practical Hardening of Crash-Tolerant Systems. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, Boston, MA, June 2012.

[18] Data Center Knowledge. Ma.gnolia data is gone for good. http://www.datacenterknowledge.com/archives/2009/02/19/magnolia-data-is-gone-for-good/.

[19] Datastax. Netflix Cassandra Use Case. http://www.datastax.com/resources/casestudies/netflix.

[20] DataStax. Read Repair: Repair during Read Path. http://docs.datastax.com/en/cassandra/3.0/cassandra/operations/opsRepairNodesReadRepair.html.

[21] S. Dawson, F. Jahanian, and T. Mitton. ORCHESTRA: A Probing and Fault Injection Environment for Testing Protocol Implementations. In *Proceedings of the 2nd International Computer Performance and Dependability Symposium (IPDS '96)*, 1996.

[22] Jeff Dean. Building Large-Scale Internet Services. 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. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP '07)*, Stevenson, WA, October 2007.

[24] Jon Elerath. Hard-disk Drives: The Good, the Bad, and the Ugly. *Commun. ACM*, 52(6), June 2009.

[25] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. 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)*, Salt Lake City, Utah, 2012.

[26] Daniel Fryer, Dai Qin, Jack Sun, Kah Wai Lee, Angela Demke Brown, and Ashvin Goel. Checking the Integrity of Transactional Mechanisms. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[27] Daniel Fryer, Kuei Sun, Rahat Mahmood, Ting-Hao Cheng, Shaun Benjamin, Ashvin Goel, and Angela Demke Brown. Recon: Verifying File System Consistency at Runtime. In *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST '12)*, San Jose, CA, February 2012.

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

[29] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[30] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, Toronto, Ontario, Canada, August 2011.

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

[32] Weining Gu, Z. Kalbarczyk, Ravishankar K. Iyer, and Zhenyu Yang. Characterization of Linux Kernel Behavior Under Errors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '03)*, San Francisco, CA, June 2003.

[33] 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. 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)*, Seattle, WA, November 2014.

[34] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical Software Model Checking via Dynamic Interface Reduction. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP '11)*, Cascais, Portugal, October 2011.

[35] James R Hamilton et al. On Designing and Deploying Internet-Scale Services. In *Proceedings of the 21st Annual Large Installation System Administration Conference (LISA '07)*, Dallas, Texas, November 2007.

[36] Seungjae Han, Kang G Shin, and Harold A Rosenberg. DOCTOR: An Integrated Software Fault Injection Environment for Distributed Real-time Systems. In *Proceedings of the International Computer Performance and Dependability Symposium (IPDS '95)*, 1995.

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

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

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

[40] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for

Disasters. In *Proceedings of the 3rd USENIX Symposium on File and Storage Technologies (FAST '04)*, San Francisco, CA, April 2004.

[41] 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. 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 2000)*, Cambridge, MA, November 2000.

[42] Kyle Kingsbury. Jepsen. `http://jepsen.io/`.

[43] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. 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)*, Broomfield, CO, October 2014.

[44] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. XFT: Practical Fault Tolerance Beyond Crashes. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI '16)*, Savannah, GA, November 2016.

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

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

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

[48] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 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)*, Portland, Oregon, June 2015.

[49] Ningfang Mi, A. Riska, E. Smirni, and E. Riedel. Enhancing Data Availability in Disk Drives through Background Activities. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '08)*, Anchorage, Alaska, June 2008.

[50] Michael Rubin. Google moves from ext2 to ext4. `http://lists.openwall.net/linux-ext4/2010/01/04/8`.

[51] MongoDB. MongoDB. `https://www.mongodb.org/`.

[52] MongoDB. MongoDB at eBay. `https://www.mongodb.com/presentations/mongodb-ebay`.

[53] MongoDB. MongoDB WiredTiger. `https://docs.mongodb.org/manual/core/wiredtiger/`.

[54] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*, Haifa, Israel, June 2016.

[55] Netflix. Cassandra at Netflix. `http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html`.

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

[57] Oracle. Fusion-IO Data Integrity. `https://blogs.oracle.com/linux/entry/fusion_io_showcases_data_integrity`.

[58] Oracle. Preventing Data Corruptions with HARD. `http://docs.oracle.com/cd/B12037_01/server.101/b10726/apphard.htm`.

[59] Patrick ONeil, Edward Cheng, Dieter Gawlick, and Elizabeth ONeil. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica*, 33(4), 1996.

[60] Bernd Panzer-Steindel. 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. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI '14)*, Broomfield, CO, October 2014.

[62] Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *The Proceedings of the International Conference on Dependable Systems and Networks (DSN-2005)*, Yokohama, Japan, June 2005.

[63] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, Brighton, UK, October 2005.

[64] Rahul Bhartia. MongoDB on AWS Guidelines and Best Practices. `http://media.amazonwebservices.com/AWS_NoSQL_MongoDB.pdf`.

[65] Redis. Instagram Architecture. `http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html`.

[66] Redis. Redis. `http://redis.io/`.

[67] Redis. Redis at Flickr. `http://code.flickr.net/2014/07/31/redis-sentinel-at-flickr/`.

[68] Redis. Silent data corruption in Redis. `https://github.com/antirez/redis/issues/3730`.

[69] RethinkDB. Integrity of read results. `https://github.com/rethinkdb/rethinkdb/issues/5925`.

[70] RethinkDB. RethinkDB. `https://www.rethinkdb.com/`.

[71] RethinkDB. RethinkDB Data Storage. `https://www.rethinkdb.com/docs/architecture/#data-storage`.

[72] RethinkDB. RethinkDB Doc Issues. `https://github.com/rethinkdb/docs/issues/1167`.

[73] RethinkDB. Silent data loss on metablock corruptions. `https://github.com/rethinkdb/rethinkdb/issues/6034`.

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

[75] Robert Harris. Data corruption is worse than you know. `http://www.zdnet.com/article/data-corruption-is-worse-than-you-know/`.

[76] Ron Kuris. Cassandra From tarball to production. `http://www.slideshare.net/planetcassandra/cassandra-from-tarball-to-production-2`.

[77] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

[78] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4), 1984.

[79] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Symposium on File and Storage Technologies (FAST '10)*, San Jose, CA, February 2010.

[80] Bianca Schroeder and Garth A. Gibson. 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)*, San Jose, CA, February 2007.

[81] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST 16)*, Santa Clara, CA, February 2016.

[82] D.P. Siewiorek, J.J. Hudak, B.H. Suh, and Z.Z. Segal. Development of a Benchmark to Measure System Robustness. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, Toulouse, France, June 1993.

[83] Gopalan Sivathanu, Charles P. Wright, and Erez Zadok. Ensuring Data Integrity in Storage: Techniques and Applications. In *The 1st International Workshop on Storage Security and Survivability (StorageSS '05)*, FairFax County, Virginia, November 2005.

[84] Mike J. Spreitzer, Marvin M. Theimer, Karin Petersen, Alan J. Demers, and Douglas B. Terry. Dealing with Server Corruption in Weakly Consistent Replicated Data Systems. *Wirel. Netw.*, 5(5), October 1999.

[85] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 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)*, Istanbul, Turkey, March 2015.

[86] David T. Stott, Benjamin Floering, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. 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)*, Chicago, IL, 2000.

[87] Sriram Subramanian, Yupu Zhang, Rajiv Vaidyanathan, Haryadi S Gunawi, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Jeffrey F Naughton. Impact of Disk Corruption on Open-Source DBMS. In *Proceedings of the 26th International Conference on Data Engineering (ICDE '10)*, Long Beach, CA, March 2010.

[88] Michael M. Swift, Brian N. Bershad, and Henry M. Levy. Improving the Reliability of Commodity Operating Systems. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, New York, October 2003.

[89] T. K. Tsai and R. K. Iyer. 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)*, London, UK, September 1995.

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

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

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

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

[94] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness in the Salus Scalable Block Store. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, Lombard, IL, April 2013.

[95] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: Transparent Model Checking of Unmodified Distributed Systems. In *Proceedings of the 6th Symposium on Networked Systems Design and Implementation (NSDI '09)*, Boston, MA, April 2009.

[96] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. 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)*, Broomfield, CO, October 2014.

[97] Yupu Zhang, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau. ViewBox: Integrating Local File Systems with Cloud Storage Services. In *Proceedings of the 12th USENIX Symposium on File and Storage Technologies (FAST '14)*, Santa Clara, CA, February 2014.

[98] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 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)*, San Jose, CA, February 2010.

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

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

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