

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*



Aishwarya Ganesan is a PhD student in Computer Sciences at the University of Wisconsin-Madison. Her advisors are Professors Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau. She is interested in distributed systems, file systems, and storage. ag@cs.wisc.edu



Ramnatthan Alagappan is a PhD student in computer sciences at the University of Wisconsin-Madison, advised by Professor Andrea Arpaci-Dusseau and Professor Remzi Arpaci-Dusseau. His research interests include file systems, storage, operating systems, and distributed systems. ra@cs.wisc.edu



Andrea Arpaci-Dusseau is a Professor of Computer Sciences at the University of Wisconsin-Madison. She is an expert in file and storage systems, having published more than 80 papers in this area, co-advised 20 PhD students, and received more than 10 Best Paper awards. dusseau@cs.wisc.edu



Remzi Arpaci-Dusseau is a Professor of Computer Sciences at the University of Wisconsin-Madison. His research focus is on file and storage systems, and his teaching interests lie in creating free online materials for all (e.g., <http://www.ostep.org>). remzi@cs.wisc.edu

* Joint senior authors

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 the behaviors of eight popular distributed storage systems, including Cassandra, Redis, and ZooKeeper. The major result of our study is that a single file-system fault introduced in one node of the cluster can induce catastrophic outcomes such as data loss, corruption, and unavailability. We find that most systems do not consistently use redundancy to recover from file-system faults. 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 storage systems.

Redundancy is a well-known technique for providing fault tolerance. Using redundancy, a system can tolerate failures of one or more of its components. For example, in a distributed storage system, data and functionality are replicated across many servers for fault tolerance. In most cases, replication can mask various failures such as system crashes, power failures, or nodes becoming inaccessible due to network failures. Modern distributed storage systems typically depend on local file systems to store and manage their data. Although replication can mask whole machine failures, local file systems exhibit a more complex failure model. For instance, certain blocks of data can become inaccessible due to an underlying latent sector error or, worse, the local file system may silently return corrupted data on reads if the underlying device block is corrupted. We call these failures file-system faults.

Several studies have shown the prevalence of errors and corruptions in disks and SSDs [1, 2, 5] that lead to these file-system faults. However, little is known about how modern distributed storage systems react to such file-system faults. Therefore, in this study, we answer the following questions: *How do distributed storage systems behave in the presence of local file-system faults? Do they use redundancy to recover from local file-system faults?*

To answer these questions, we systematically inject file-system faults into distributed storage systems and observe the effects of the injected fault. We picked a broad spectrum of distributed storage systems, implementing a variety of replication protocols such as replicated state machines, primary backup, and dynamo-style quorums.

Our fault model is very simple—we inject exactly one file-system fault into one file-system block in one node in the system at a time. We inject corruptions on reads, errors on reads, and errors on writes. Moreover, our fault model only includes data corruptions that are detectable by applications (e.g., using application-level checksums) and does not include undetectable memory corruptions.

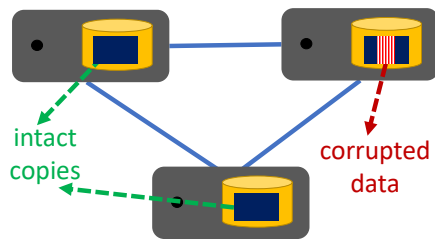


Figure 1: User expectations. The figure shows a data item replicated on three servers in a distributed storage system. When one copy is corrupted, users typically expect that redundant copies will help recover from the single corruption.

A common and widespread expectation is that redundancy in higher layers (i.e., across replicas) enables recovery from local file-system faults. For instance, consider a data item that is replicated across three machines in a system as shown in Figure 1. What would a user expect if one of the copies of the data item in the system gets corrupted? Similarly, what if one of the blocks in one of the copies becomes inaccessible? It is completely reasonable for a user to expect that the corrupted data will be recoverable from the intact copies on other replicas and that the user never sees the corrupted data.

Unfortunately, from our study, we find that redundancy does not provide fault tolerance in many distributed storage systems. We find several pieces of evidence where a single file-system fault in only one node leads to catastrophic outcomes such as data loss, silent user-visible corruption, unavailability, or sometimes even the spread of corrupted data to other intact replicas. Table 1 shows the prevalence of various undesirable behaviors across multiple systems. Note that since the system has redundant copies of data and we inject only one fault at a time, these behaviors are surprising and undesirable.

Why does redundancy not imply fault tolerance? One might wonder whether the discovered outcomes arise simply due to some implementation-level bugs that could be fixed by moderate developer effort. Unfortunately, from our study, we find that the above outcomes arise due to some alarming and fundamental root causes in file-system fault tolerance that are common to many distributed storage systems.

The first fundamental problem we observe is that *faults are often undetected locally* by the nodes in a distributed storage system, leading to harmful effects such as corrupted data being returned to the users. Second, even when systems reliably detect faults, in most cases, they *simply crash* instead of using redundancy to recover from the fault. Third, many systems *do not discern corruptions caused due to crashes from other corruptions*, resulting in many data loss cases. Finally, we find that local fault-handling

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

Table 1: Catastrophic outcomes: summary. The table shows the summary of catastrophic outcomes resulting from a single file-system fault. A shaded box for a system indicates that we discovered at least one instance of the outcome mentioned on the left.

behaviors and global distributed protocols *interact in an unsafe manner*, leading to propagation of corruption or data loss.

As distributed storage systems are emerging as the primary choice for storing critical user data, carefully building them to tolerate file-system faults is important. Our study is a step in this direction, and we hope that our results will lead to discussions and future research to improve the resiliency of next generation cloud storage systems. The full version of our work was published in FAST '17 [3]. Our testing framework is publicly available at <http://research.cs.wisc.edu/adsl/Software/cords>.

Methodology

In this section, we first discuss the fault model and then describe our methodology to study how distributed storage systems react to local file-system faults.

Fault Model

Our fault model is very simple—we inject a single fault into a single file-system block exactly one node at a time. We inject these faults into file-system user data and not the file-system metadata. The reason for this is simple: the file system is responsible for maintaining the integrity of its metadata, while applications should take care of their on-disk data.

Our fault model captures the behavior of different real file systems. Consider that the nodes of a distributed storage system run on an ext4 file system. If the underlying device block is corrupted, ext4 returns corrupted data as-is to applications since it does not have checksums for user data. On the other hand, consider a file system such as btrfs that maintains checksums for user data; such a file system transforms an underlying block corruption into a read error.

To capture these different file system behaviors, our fault model injects three types of faults: corruption on reads, error on reads, and error on writes. Our fault model assumes detectable corruptions (e.g., corruptions detectable using application-level check-

FILE SYSTEMS AND STORAGE

Redundancy Does Not Imply Fault Tolerance

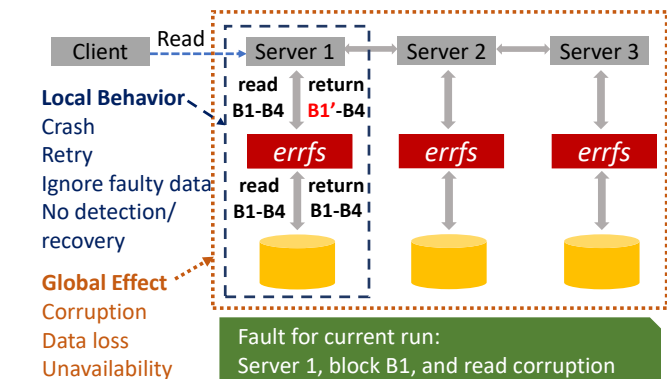


Figure 2: Fault injection methodology. *errfs* injects faults into one file-system block one node at a time. For each fault, we need to observe the local behavior and the global effect.

sums) and does not include arbitrary memory corruptions that are not detectable by applications (e.g., corruptions introduced before checksum computation or corruptions introduced after checksum verification).

Fault Injection

To study how distributed storage systems react to local file-system faults, we build a 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.

To understand how our fault-injection methodology works, consider a distributed storage system with three nodes, as shown in Figure 2. We configure the system to run atop *errfs* and run a system-specific workload multiple times, each time injecting a single fault for a single file-system block in a single node. Assume that for a particular run we would like to inject a read corruption for block *B1* on server 1. After reading the blocks from the disk, *errfs* corrupts *B1* before returning to the server. To emulate errors, *errfs* does not perform the operation but simply returns an appropriate error code.

Behavior Inference

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: first, the *local behavior* of the node where the fault is injected. Locally, the faulty node could *crash*, *retry* the operation, detect and *ignore the faulty data*, or perform *no detection or recovery*, etc.

Second, we need to observe the *global effect* of the injected fault. The global effect of a fault is the result that is externally visible. Ideally, we should not observe any harmful effect since the data

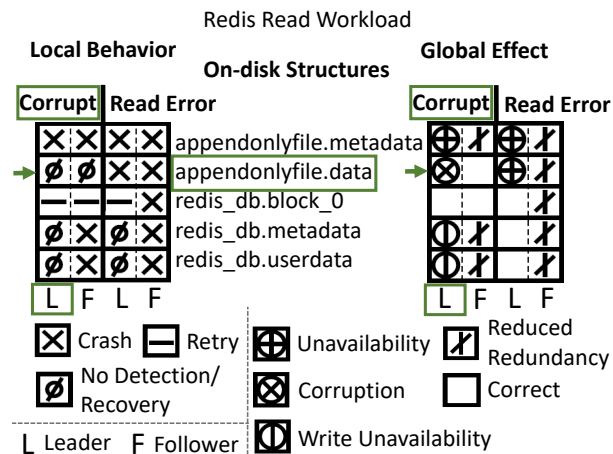


Figure 3: Behavior analysis of Redis read. The figure shows local behaviors and global effects when corruptions and read errors are injected in various on-disk logical structures during read workload in Redis. The grid on the left shows the local behavior of the node where the fault is injected, and the one on the right shows the cluster-wide global effect of the injected fault. The annotation on the top of a grid shows the type of fault: for example, “Corrupt” means that we inject data corruption using *errfs*. The annotation between the grids 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, F - follower).

is replicated and we inject only one fault at a time. Some adverse global effects that could occur include data loss, user-visible corruption, read-unavailability, write-unavailability, unavailability, or query failure. 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.

Behavior Analysis

We studied the following eight distributed storage systems using *CORDS*, our framework for injecting faults: 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).

An Example: Redis

To illustrate our behavior analysis, we use Redis as an example. Redis is a data structure store with a leader and set of followers. On a write request, data is appended to the *append-only file* and also replicated on to the followers. The append-only file is periodically snapshotted into the Redis *database_file*.

Figure 3 shows the behaviors of Redis when faults are injected during a read workload. We represent our results in grids like the ones shown in the figure. We inject different faults such as corruption and read or write errors into either a leader or a follower one at a time and for different on-disk structures. The on-disk structures take the form: *file_name.logical_entity*. We derive

Redundancy Does Not Imply Fault Tolerance

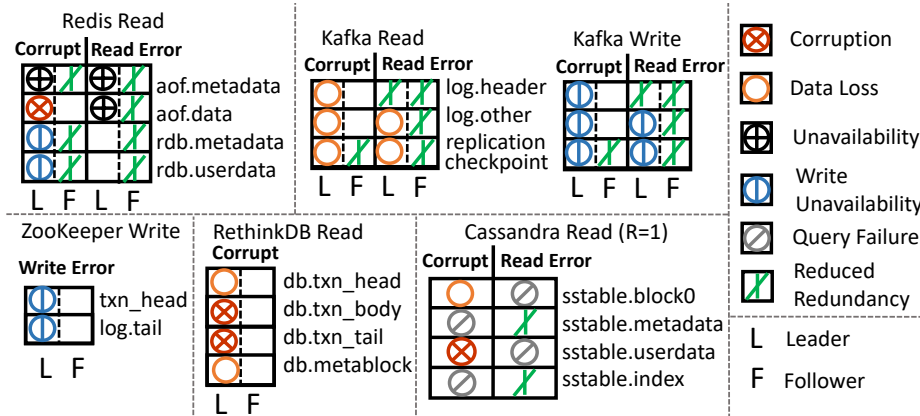


Figure 4: Redundancy does not provide fault tolerance. The figure shows a sample of catastrophic outcomes such as corruption, data loss, unavailability, query failures, and reduced redundancy that occur across many systems. These outcomes (global effects) occur when corruptions, read errors, and write errors are injected in various on-disk logical structures during read and write workloads in different distributed storage systems.

the logical entity name from our understanding of the on-disk format of the file. For each injected fault, we observe how the system behaves.

For example, when there are corruptions in the data in the append-only file on the leader (highlighted with outlining in the figure), the corruption is undetected (local behavior), and the corrupted data is silently returned (global effect). Redis does not use checksums for append-only file user data; thus, it does not detect corruptions. Moreover, the resynchronization protocol in Redis propagates corrupted user data from the leader to the followers leading to a global user-visible corruption. We repeat this analysis by running the read workload multiple times, each time injecting a different fault into a different on-disk structure.

We also repeat the analysis for other systems for read and write workloads. These results and analyses are presented in detail in our FAST '17 paper [3]. We will use the results from this behavior analysis of various systems to draw observations in the rest of this article.

Major Results

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. Figure 4 shows a sample of results that illustrate the prevalence of catastrophic problems across multiple systems.

In most cases, the problems shown in Figure 4 are not caused by simple implementation bugs. Rather, they are caused due to some

fundamental problems in file-system fault tolerance that are common to many distributed storage systems.

Fundamental Problems

We now discuss some of the fundamental root causes that are responsible for the catastrophic problems that we discover in all systems.

Faults Are Often Undetected Locally

The first fundamental problem we observe is that faults are often undetected locally. These locally undetected faults might lead to harmful global effects. For example, a locally undetected corruption could result in a global silent corruption.

Figure 5 shows how a locally undetected fault leads to harmful global effects in Cassandra. The figure shows the case where the user data in the *sstable* on one node is corrupted. Cassandra does not detect this corruption using checksums when compression is not enabled. Thus, any read request for this data item to the corrupted replica will silently receive corrupted data. Further, the

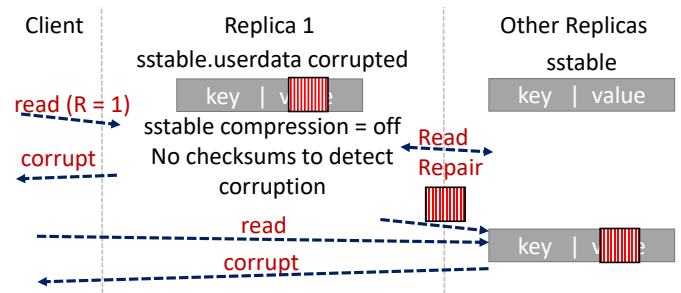


Figure 5: Faults are often undetected locally. The figure shows how a locally undetected fault can lead to harmful global effects in Cassandra.

FILE SYSTEMS AND STORAGE

Redundancy Does Not Imply Fault Tolerance

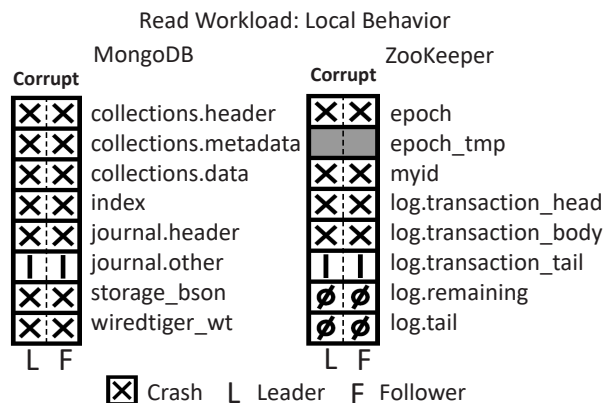


Figure 6: Crashing is the most common local reaction. The figure shows that crashing is the most common local reaction when corruptions are introduced into various on-disk structures during the read workload in MongoDB and ZooKeeper.

read repair protocol that fixes stale versions of data propagates the corruption to other replicas. Many other systems exhibit similar problems (e.g., RethinkDB and Redis); these systems completely trust and rely upon the lower layers in the storage stack to handle data integrity problems.

Crashing Is the Most Common Reaction

The next fundamental problem is that crashing is the most common local reaction. Many systems do reliably detect faults, but in most cases they simply crash on detecting a fault instead of using redundancy to recover from the fault. For example, MongoDB and ZooKeeper have checksums for most of their on-disk data structures to detect corruptions. Figure 6 shows the local behavior of these systems when corruptions are introduced into various on-disk structures during the read workload. As shown in the figure, nodes in MongoDB and ZooKeeper simply crash on detecting a corruption. We observe the same behavior in many other systems.

Although crashing does not result in a harmful effect immediately, it introduces the possibility of an imminent unavailability. Moreover, since storage faults could be persistent, simply restarting the faulty node does not help; the node would encounter the same fault and crash again. Solving such cases requires some manual intervention, which is often error-prone and cumbersome. Although crashing may seem like a good strategy to employ, in a distributed system there are opportunities to recover from local faults using copies on other intact replicas.

Crashing and Corruption Handling Are Entangled

The next observation we make is that crash and corruption handling are entangled. We illustrate this using Kafka. Kafka is a persistent distributed message queue in which the messages

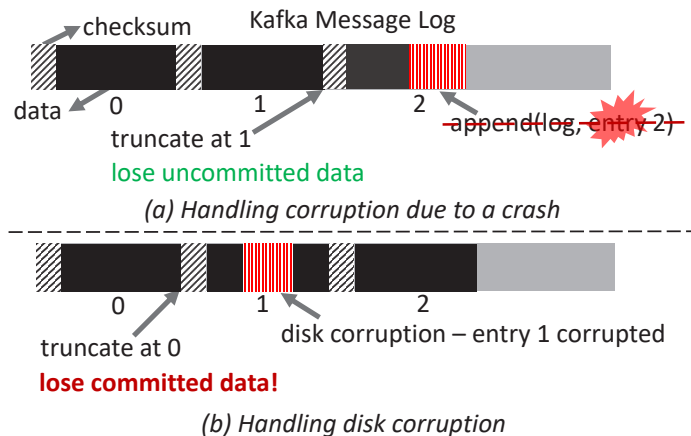


Figure 7: Crash and corruption handling are entangled. The figure shows how entanglement in crash and corruption handling could lead to a local data loss of committed data in Kafka.

are stored in a log. Incoming messages are appended to the log, and each message is checksummed. Consider that a Kafka node crashes during an append of message 2 as shown in Figure 7. When the node recovers from the crash, it detects a checksum mismatch because of the partially appended entry. As a recovery action, the node truncates the log at message 1. Note that message 2 is uncommitted as the node crashed while appending it. Hence, it is safe to truncate the uncommitted message in this case.

On the other hand, consider the case where all messages 0, 1, and 2 are persisted safely on disk, but the block holding message 1 is corrupted. Kafka detects this corruption using checksums, but it truncates the log at message 0 since it treats this disk corruption as a corruption that occurred due to a crash. Note that messages 1 and 2 were committed and it is not safe to lose them. Since Kafka conflates the handling of a disk corruption and a corruption due to a crash, it loses committed data.

Developers of RethinkDB and LogCabin agree that entanglement is a problem. Thus, there is a need to disentangle corruptions due to crashes from other types of corruptions.

Unsafe Interaction between Local and Global Protocols

Next, we observe that the local behavior of a faulty node and the global protocols interact in unsafe ways. We illustrate this again using Kafka. Recall that the Kafka node treats a disk corruption the same way it treats a corruption due to a crash, resulting in a data loss. However, this data loss is the local behavior of the corrupted node. Assume that this data loss occurred on node 1. Other nodes still have the data as shown in Figure 8.

Kafka maintains a piece of metadata that contains information about replicas that are in-sync; any node in this set has all the committed data and is eligible to become a leader. In this case,

Redundancy Does Not Imply Fault Tolerance

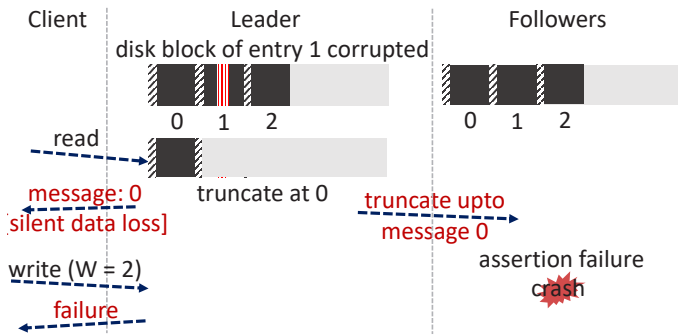


Figure 8: Unsafe interaction between local behaviors and global protocols. The figure shows how local fault-handling behaviors in Kafka interact with the global leader election protocol in an unsafe manner. Node 1, which lost committed data due to entanglement in crash and corruption handling, is elected as the leader, resulting in data loss and write unavailability.

node 1, which lost committed data, is not removed from the set of in-sync replicas and is elected as the leader. Thus, any further reads return only message 0, resulting in a silent data loss. Moreover, the leader also instructs the followers to truncate the log at message 0 which triggers an assertion at followers, resulting in their crash. Thus, all future writes become unavailable. 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. Thus, there is a need for synergy between local behaviors and global protocols to avoid such problems.

Fundamental Problems: Summary

Table 2 shows how the fundamental problems are common across many systems. We observe that all systems we studied simply crash on detecting a fault in many cases. In some cases, systems take incorrect recovery action on detecting a fault, leading to undesirable behaviors. We also observe that all systems miss opportunities to recover from local file-system faults using redundancy.

Conclusion

Most popular distributed systems we studied are not yet resilient to local file-system faults. Although a body of research work and enterprise storage systems provide software guidelines to tackle partial file-system 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.

Our study provides four important lessons for future distributed storage system design. First, in the world of layered storage stacks that run on commodity hardware, faults are common;

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

Table 2: Fundamental problems summary. The table shows the summary of the fundamental problems across all the systems we studied. A shaded box for a system indicates that we observed at least one instance of the problem mentioned on the left.

thus, distributed storage systems need to detect such faults carefully. Second, in a distributed system, several unavoidable cases such as power faults and network failures can cause nodes to be unavailable. In cases where automatic recovery is possible, simply crashing is not the optimal behavior. Next, by disentangling corruptions caused by a crash from other types of corruptions and by handling them differently, storage systems can avoid many problems. Finally, local fault-handling behavior has global implications for distributed systems. Distributed storage system developers need to fully understand this interaction in order to improve reliability.

We hope that our study and results will provide direction for the design of more robust distributed storage systems. Our fault-injection framework is available at <http://research.cs.wisc.edu/adsl/Software/cords>.

Acknowledgments

We thank the anonymous FAST reviewers, Hakim Weatherspoon (our shepherd), and Rik Farrow 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 CNS1218405, DOE grant DE-SC0014935, as well as donations from EMC, Facebook, Google, Huawei, Microsoft, NetApp, Samsung, Seagate, Veritas, and VMware. Finally, we thank CloudLab [4] 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] L. N. Bairavasundaram, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, G. R. Goodson, and B. Schroeder, “An Analysis of Data Corruption in the Storage Stack,” in *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST '08)*, February 2008, pp. 223–238: https://www.usenix.org/legacy/event/fast08/tech/full_papers/bairavasundaram/bairavasundaram.pdf.
- [2] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. 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)*, June 2007: <http://research.cs.wisc.edu/adsl/Publications/latent-sigmetrics07.pdf>.
- [3] A. Ganesan, R. Alagappan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions,” in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*, February 2017, pp. 149–166: <https://www.usenix.org/system/files/conference/fast17/fast17-ganesan.pdf>.
- [4] R. Ricci, E. Eide, and CloudLab Team, “Introducing CloudLab: Scientific Infrastructure for Advancing Cloud Architectures and Applications,” *login.*, vol. 39, no. 6 (December 2014): <https://www.usenix.org/publications/login/dec14>.
- [5] B. Schroeder, R. Lagisetty, and A. Merchant, “Flash Reliability in Production: The Expected and the Unexpected,” in *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, February 2016, pp. 67–80: <https://www.usenix.org/system/files/conference/fast16/fast16-papers-schroeder.pdf>.