Correlated Crash Vulnerabilities

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

https://www.usenix.org/conference/osdi16/technical-sessions/presentation/alagappan

This paper is included in the Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16).

November 2–4, 2016 • Savannah, GA, USA

ISBN 978-1-931971-33-1

Correlated Crash Vulnerabilities

Ramnatthan Alagappan, Aishwarya Ganesan, Yuvraj Patel,
Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin – Madison

Abstract

Modern distributed storage systems employ complex protocols to update replicated data. In this paper, we study whether such update protocols work correctly in the presence of *correlated crashes*. We find that the correctness of such protocols hinges on how local file-system state is updated by each replica in the system. We build PACE, a framework that systematically generates and explores persistent states that can occur in a distributed execution. PACE uses a set of generic rules to effectively prune the state space, reducing checking time from days to hours in some cases. We apply PACE to eight widely used distributed storage systems to find *correlated crash vulnerabilities*, i.e., problems in the update protocol that lead to user-level guarantee violations. PACE finds a total of 26 vulnerabilities across eight systems, many of which lead to severe consequences such as data loss, corrupted data, or unavailable clusters.

1 Introduction

Modern distributed storage systems are central to large scale internet services [19,22,28,66]. Important services such as photo stores [74,77,79], e-commerce [61], video stores [27], text messaging [82], social networking [93], and source repositories [78] are built on top of modern distributed storage systems. By providing replication, fault tolerance, high availability, and reliability, distributed storage systems ease the development of complex software services [17,31,59,75].

Reliability of user data is one of the most important tenets of any storage system [3, 41, 49, 69]. Distributed storage systems improve reliability by replicating data over a collection of servers [7,13,63,80,84,89,91].

To safely replicate and persist application data, modern storage systems implement complex data update protocols. For example, ZooKeeper [5] implements an atomic broadcast protocol and several systems including LogCabin [57] and etcd [23] implement the Raft consensus protocol to ensure agreement on application data between replicas. Although the base protocols (such as atomic broadcast [12], Raft [68], or Paxos [51]) are provably correct, implementing such a protocol without bugs is still demanding [16, 36, 42, 44, 94], especially when machines can crash at any instant [53].

Many distributed storage systems can recover from single node or partial cluster failures. In this study, we consider a more insidious crash scenario in which all replicas of a particular data shard crash at the same time and recover at a later point. We refer to such crash scenarios as *correlated failures*. Correlated failures are common and several instances of such failures have been reported in the recent past [11,20,21,25,26,35,43,49,65, 92]; these failures occur due to root causes such as data-center-wide power outages [38], operator errors [97], planned machine reboots, or kernel crashes [35].

When nodes recover from a correlated failure, the common expectation is that the data stored by the distributed system would be recoverable. Unfortunately, local file systems (which store the underlying data and metadata of many distributed storage systems) complicate this situation. Recent research has shown that file systems vary widely with respect to how individual operations are persisted to the storage medium [70]. For example, testing has revealed that in ext4, f2fs [8], and u2fs [58], one cannot expect the following guarantee: a file always contains a prefix of the data appended to it (i.e., no unexpected data or garbage can be found in the appended portion) after recovering from a crash. The same test also shows that this property *may* be held by btrfs and xfs. Since most practical distributed systems run atop local file systems [47,62,81], it is important for them to be aware of such behaviors. These file-system nuances can result in unanticipated persistent states in one or more nodes when a distributed storage system recovers from a correlated crash.

Recent studies [14, 70] have demonstrated that these widely varying file-system behaviors increase the complexity of building a crash-consistent update protocol, even for single machine applications such as SQLite. We refer to this form of crash consistency as *single-machine (application-level) crash consistency*.

Distributed storage systems have to deal with the complexity of building a crash-consistent storage update protocol in addition to correctly implementing a distributed agreement and recovery protocol. We refer to this form of crash consistency in a distributed setting as *correlated crash consistency*. Although the challenges of building a crash-consistent distributed update protocol have the same flavor as building a crash-consistent single-machine protocol, correlated crash consistency is a fundamentally different problem for three reasons.

First, distributed systems can fail in more ways than a single machine system. Since a distributed system constitutes many components, a group of components may fail together at the same or different points in the protocol. Second, unique opportunities and problems exist in distributed crash recovery; after a failure, it is possible for one node in an inconsistent state to repair its state by contacting other nodes or to incorrectly propagate the corruption to other nodes. In contrast, single-machine applications rarely have external help. Third, crash recovery in a distributed setting has many more possible paths than single-machine crash recovery as distributed recovery depends on states of many nodes in the system.

We say a distributed system has a *correlated crash vulnerability* if a correlated crash during the execution of the system's update protocol (and subsequent recovery) exposes a user-level guarantee violation. In this paper, we examine whether distributed storage systems are vulnerable to correlated crashes. To do this, we introduce PACE, a novel framework that systematically explores correlated crash states that occur in a distributed execution.

PACE considers consistent cuts in the distributed execution and generates persistent states corresponding to those cuts. PACE models local file systems at individual replicas using an *abstract persistence model* (APM) [70] which captures the subtle crash behaviors of a particular file system. PACE uses *protocol-specific knowledge* to reduce the exploration state space by systematically choosing a subset of nodes to introduce file-system nuances modeled by the APM. In the worst case, if no attributes of a distributed protocol are known, PACE can operate in a slower brute-force mode to still find vulnerabilities.

We applied PACE to eight widely used distributed storage systems spanning important domains including database caches (Redis [76]), configuration stores (ZooKeeper [5], LogCabin [57], etcd [23]), real-time databases (RethinkDB [83]), document stores (MongoDB [60]), key-value stores (iNexus [46]), and distributed message queues (Kafka [6]).

We find that many of these systems are vulnerable to correlated crashes. Modern distributed storage systems expect certain guarantees from file systems such as ordered directory operations and atomic appends for their local update protocols to work correctly. We also find that in many cases global recovery protocols do not use intact replicas to fix the problematic nodes. PACE found a total of 26 vulnerabilities that have severe consequences such as data loss, silent corruption, and unavailability. We also find that many vulnerabilities can be exposed on commonly used file systems such as ext3, ext4, and btrfs. We reported 18 of the discovered vulnerabilities to application developers. Twelve of them have been already fixed or acknowledged by developers. While some vulnerabilities can be fixed by straightforward code changes, some are fundamentally hard to fix.

Our study also demonstrates that PACE is *general*: it can be applied to any distributed system; PACE is *systematic*: it explores different systems using general rules that we develop; PACE is *effective*: it found 26 *unique* vulnerabilities across eight widely used distributed systems. PACE's source code and details of the discovered vulnerabilities are publicly available [2].

The rest of the paper is organized as follows. We first describe correlated crash consistency in detail (§2). Next, we explain how PACE works and how it uses protocol-awareness to systematically reduce exploration state space (§3). Then, we present our study of correlated crash vulnerabilities in distributed storage systems (§4). Finally, we discuss related work (§5) and conclude (§6).

2 Correlated Crash Consistency

Building a crash-consistent distributed update protocol is complex for two reasons: machines can crash at any time in a correlated fashion and updates to the local file system have to be performed carefully to recover from such crashes. Given this complexity, we answer the following question in this paper: *Do modern distributed storage systems implement update and recovery protocols that function correctly when nodes crash and recover in a correlated fashion, or do they have vulnerabilities*? To answer this question, we first describe the failure model that we consider and build arguments for why the considered failure model is important. Next, we explain the system states we explore to find if a distributed update protocol has correlated crash vulnerabilities.

2.1 Failure Model

Components in a distributed system can fail in various ways [39]. Most practical systems do not deal with Byzantine failures [52] where individual components may give conflicting information to different parts of the system. However, they handle fail-recover failures [39] where components can crash at any point in time and recover at any later point in time, after the cause of the failure has been repaired. When a node crashes and recovers, all its in-memory state is lost; the node is left only with its persistent state. Our study considers only such fail-recover failures.

A common expectation is that practical distributed systems would not violate user-level guarantees in the face of such fail-recover failures. However, nuances in local file system can cause unanticipated persistent states to arise after a crash and a subsequent reboot, complicating proper recovery. We explain such nuances in local file-system behaviors later (§2.2.2). Notice that this is how one individual node crashes and recovers; other nodes may continue to function and make progress.

Our failure model concentrates on a more devastating scenario where *all* or a *group* of nodes crash and recover together. We refer to this type of failure as a correlated crash. Specifically, we focus on two kinds of correlated crash scenarios: first, data-center-wide power outages where *all* machines in the cluster crash and recover together; second, correlated failures where only a *group of machines* containing all replicas for a shard of data crash and recover together and other machines (that are not part of the shard) in the cluster do not react to the failure.

Large-scale correlated failures such as cluster-wide power outages are common and occur in the real world [11,20,21,25,26,35,43,49,65,92]. For example, a recent study from Google [35] showed that node failures in Google data centers are often correlated and the causes of node failures fall into three categories: application restarts, planned machine reboots, and unplanned machine reboots. From data over a period of three months, the study showed that as many as 15 unplanned reboots and 30 planned reboots per 1000 machines can happen in a single day. Also, a *failure burst* in one instance of a distributed file system [37] can take down as many as 50 machines at almost the same time. This kind of failure typically can be seen during a power outage in a data center. Rolling kernel upgrades also cause failure bursts that can take down around 20 machines within a short window of time.

Although the system cannot progress when all replicas crash, the common expectation is that the data stored by the storage system will be recoverable after the replicas come alive (for example, after power has been restored).

Our failure model is not intended to reason about scenarios where only a subset of replicas of a particular data shard crash and recover by themselves. Also, the vulnerabilities we find with our correlated failure model do not apply to a geo-replicated setting; in such a setting, conscious decisions place replicas such that one power failure cannot affect all replicas at the same time. While correlated failures are less problematic in such settings, the storage systems we examine in this study are heavily tested and the common expectation is that these systems should be reliable irrespective of how they are deployed and the probability of failures. Further, many deployments are not geo-replicated and thus may expect strong guarantees even in the presence of correlated crashes. Overall, crash-correctness should be deeply ingrained in these systems regardless of deployment decisions.

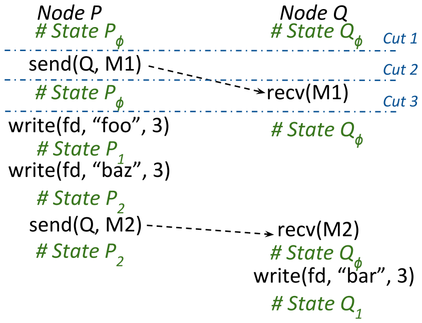


Figure 1: **A simple distributed protocol.** *The figure shows a simple distributed protocol. Annotations show the persistent state after performing each operation. Dash dot lines show different cuts.*

2.2 Distributed Crash States

Now we explain the global system states that result due to correlated crashes. As we explained, after a crash and subsequent reboot, a node is left only with its persistent data. The focus of our study is in checking only the resulting *persistent states* when failures happen. The global states that we capture are similar to distributed snapshots [18] described by Chandy and Lamport. The main difference between a generic distributed snapshot and a global persistent state is that the latter consists only of the on-disk state and not the in-memory state of the machines. Moreover, since network channels do not affect persistent on-disk state, our global persistent states do not keep track of them.

To understand the persistent states that we capture, consider a cluster of three machines named A, B, and C. Assume that the initial persistent states of these machines are A_ϕ, B_ϕ, and C_ϕ, respectively. Assume that a workload W run on this cluster transitions the persistent states to A_f, B_f, and C_f, respectively. For instance, W could be a simple workload that inserts a new key-value pair into a replicated key-value store running on A, B, and C. Notice that the persistent state of all nodes goes through a transition before arriving at the final states A_f, B_f, and C_f. A correlated crash may happen at any time while W runs, and after a reboot, the persistent state of a node X may be any intermediate state between X_ϕ and X_f where X can be A or B or C. For simplicity, we refer to this collection of persistent states across all nodes as global persistent state or simply global state. If a particular global state G can occur in an execution, we call G a *reachable* global state.

2.2.1 Reachable Global States

The reachability of a global state depends on two factors: the order in which messages are exchanged between nodes and the local file systems of the nodes. To illustrate the first factor, consider a distributed protocol shown in Figure 1. In this protocol, node P starts by sending message `M1`, then writes `foo` and `baz` to a file, and then sends another message `M2` to node Q. Node Q receives `M1` and `M2` and then writes `bar` to a file. For now, assume that the toy application is single threaded and all events happen one after the other. Also assume that the file system at P and Q is synchronous (i.e., operations are persisted *in order* to the disk). We will soon remove the second assumption and subsequently the first (§3.2).

Assume that the initial persistent state of P was P_ϕ and Q was Q_ϕ. After performing the first and second write, P moves to P_1 and P_2, respectively. Similarly, Q moves to Q_1 after performing the write. Notice that $<P_\phi, Q_\phi>$ is a reachable global persistent state as P could have crashed before writing to the file and Q could have crashed before or after receiving the first message. Similarly, $<P_2, Q_1>$ and $<P_2, Q_\phi>$ are globally reachable persistent states.

In contrast, $<P_\phi, Q_1>$ and $<P_1, Q_1>$ are unreachable persistent states as it is not possible for Q to have written the file without P sending the message to it. Intuitively, global states that are not reachable in an execution are logically equivalent to inconsistent cuts in a distributed system [10]. For example, $<P_\phi, Q_1>$ and $<P_1, Q_1>$ are inconsistent cuts since the `recv` of `M2` is included in the cut but the corresponding `send` is excluded from the cut. Also, network operations such as `send` and `recv` do not affect the persistent state. For example, the three different cuts shown in Figure 1 map onto the same persistent state $<P_\phi, Q_\phi>$.

Next, we consider the fact that the local file systems at P and Q also influence the global states. Assume that the application is still single threaded but writes issued by an application can be buffered in memory as with modern file systems. Depending on which exact file system and mount options are in use, modern file systems may reorder some (or many) updates [9, 70, 73]. With this asynchrony and reordering introduced by the file system, it is possible for the second write `baz` to reach the disk before the first write `foo`. Also, it is possible for P to crash after `baz` is persisted and the message is sent to Q, but before `foo` reaches the disk. In such a state of P, it is possible for Q to have either reached its final state Q_1 or crash before persisting `bar` and so remain in Q_ϕ. All these states are globally reachable.

2.2.2 File-system Behavior

The reordering of writes by the file system is well understood by experienced developers. To avoid such reordering, developers force writes to disk by carefully issuing `fsync` on a file as part of the update protocol. Although some common behaviors such as reordering of writes are well understood, there are subtle behaviors that application developers find hard to reason about. For example, the following subtle behavior is not well documented: if a crash happens when appending a single block of data to a file in ext4 writeback mode, the file may contain garbage on reboot. These behaviors are neither bugs nor intended features, but rather implications of unrelated performance improvements. To worsen the problem, these subtle behaviors vary across file systems.

Recent research [4,14,70–72] classifies file-system behaviors into two classes of properties: atomicity and ordering. The atomicity class of properties say whether a particular file system must persist a particular operation in an atomic fashion in the presence of crashes. For instance, must ext2 perform a `rename` in an atomic way or can it leave the system in any intermediate state? The ordering class of properties say whether a particular file system must persist an operation A before another operation B. For instance, must ext4 (in its default mode) order a `link` and a `write` operation? While ext4 orders directory operations and file write operations, the same does not hold true with btrfs which can reorder directory operations and write operations.

Given these variations across file systems and sometimes even across different configurations of the same file system, it is onerous to implement a crash-consistent protocol that works correctly on all file systems. Recent research has discovered that single-machine applications have many vulnerabilities in their update protocols which can cause them to corrupt or lose user data [4, 70, 98].

Distributed storage systems also face the same challenge as each replica uses its local file system to store user data and untimely crashes may leave the application in an inconsistent state. However, distributed systems have more opportunities for recovery as redundant copies of data exist on other nodes.

3 Protocol-Aware Crash Explorer

To examine if distributed storage systems violate user-level guarantees in correlated crash scenarios, we build a generic correlated crash exploration framework, PACE, which systematically generates persistent states that can occur in a distributed execution in the presence of correlated crashes. We note here that PACE is not intended to catch bugs in distributed consensus protocols. Specifically, it does not exercise reordering of network messages to explore corner cases in consensus protocols; as explained later (§5), distributed model checkers attack this problem. PACE's intention is to examine the interaction of global crash recovery protocols and the nuances in local storage protocols (introduced by each replica's local file system), in the presence of correlated crashes.

Some vulnerabilities that we discover are exposed only if a particular file-system operation is reordered on all replicas while some vulnerabilities are exposed even when the reordering happens on a single replica. Using observations from how vulnerabilities are exposed and a little knowledge about the distributed protocol, we make our exploration *protocol-aware*. Using this awareness, PACE can prune the search space while finding as many vulnerabilities as a brute-force search. To explain how protocol-aware exploration works, we first describe the design of our crash exploration framework.

3.1 Design and Implementation Overview

PACE is easy to use and can be readily applied to any distributed storage system. PACE needs a workload script and a checker script as inputs. For many modern distributed systems, a group of processes listening on different ports can act as a cluster of machines. For systems that do not allow this convenience, we use a group of Docker [30] containers on the same machine to serve as the cluster. In either case, PACE can test the entire system on a single machine. PACE is implemented in around 5500 lines of code in Python.

To begin, PACE starts the cluster with system call tracing, runs the workload, and then stops the cluster after the workload is completed. PACE parses the traces obtained and identifies cross node dependencies such as a `send` on one node and the corresponding `recv` on some other node. After the traces are parsed and cross node dependencies established, PACE replays the trace to generate different persistent crash states that can occur in the traced execution. A system-specific checker script is run on top of each crash state; the checker script asserts whether user-level guarantees (e.g., committed data should not be corrupted or lost) hold. Any violations in such assertions are reported as vulnerabilities. We next discuss what correlated crash states can occur in a distributed execution and how we generate them.

3.2 Crash States

We use a running example of a ZooKeeper cluster executing an update workload for further discussion. PACE produces a diagrammatic representation of the update protocol as shown in Figure 2.

First, the client contacts the leader in the ZooKeeper cluster. The leader receives the request and orchestrates the atomic broadcast protocol among its followers as shown by `send` and `recv` operations and careful updates to the file system (`write` and `fdatasync` on a log file that holds user data). Finally, after ensuring that the updated data is carefully replicated and persisted, the client is acknowledged. At this point, it is guaranteed that the data will be consistent and durable.

Note that each node runs multiple threads and the figure shows the observed order of events when the traces were collected. If arbitrary delays were introduced, the order may or may not change, but this observed order is one schedule among all such possible schedules.

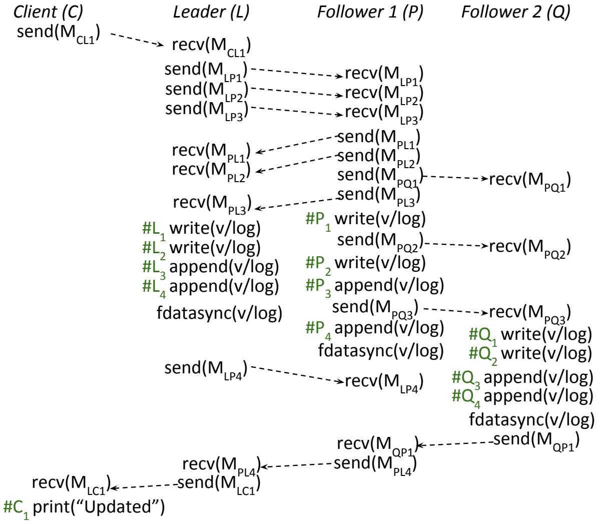


Figure 2: **ZooKeeper protocol for an update workload.** *The figure shows the sequence of steps when the client interacts with the ZooKeeper cluster. The workload updates a value. The client prints to stdout once the update request is acknowledged.*

We reiterate here that PACE captures crash states that occur due to a correlated failure where all replicas fail together. PACE is not intended to reason about partial crashes where only a subset of replicas crash.

3.2.1 Globally Reachable Prefixes

Assume that all nodes shown in Figure 2 start with persistent state X_ϕ where X is the node identifier with L for leader, C for client, and so forth. M_{XYi} is the i^{th} message sent by X to Y. All operations that affect persistent state are annotated with the persistent state to which the node transitions by performing that operation. For example, the leader transitions to state L_1 after the first `write` to a file. The total set of global persistent states is the cross product of all local persistent states. Precisely, the total set is the cross product of the sets $\{C_\phi, C_1\}$, $\{L_\phi, L_1, L_2, L_3, L_4\}$, $\{P_\phi, P_1, P_2, P_3, P_4\}$ and $\{Q_\phi, Q_1, Q_2, Q_3, Q_4\}$. However, some of the global states in this resultant set cannot occur in the distributed execution. For example, $<C_\phi, L_2,\ P_2,\ Q_1>$ is an inconsistent cut and cannot occur as a global state since it is not possible for Q to receive M_{PQ3} before P reaches P_3 and then sends M_{PQ3}.

We refer to a global state that is reachable in this trace as a globally reachable persistent prefix or simply *globally reachable prefix*. We call this a prefix as it is a prefix of the file-system operations within each node.

Previous work [70] has developed tools to uncover single-machine crash vulnerabilities. Such tools trace only file-system related system calls and do not trace network operations. Hence, they cannot capture dependencies across different nodes in a distributed system. Such tools cannot be directly applied to distributed systems; if applied, they may generate states that may not actually occur in a distributed execution and thus can report spurious vulnerabilities. On the other hand, PACE captures all cross node dependencies and so generates only states that can occur in a distributed execution.

3.2.2 File-system Persistence Models

Generating globally reachable prefixes does not require any knowledge about how a particular file system persists operations. As we discussed, file systems exhibit important behaviors with respect to how operations are persisted. We borrow the idea of *abstract persistence model (APM)* from our previous work [70] to model the file system used by each node.

An APM specifies all constraints on the atomicity and ordering of file-system operations for a given file system, thus defining which crash states are possible. For example, in an APM that specifies the ext2 file system, appends to a file can be reordered and the `rename` operation can be split into smaller operations such as deleting the source directory entry and creating the target directory entry. In contrast, in the ext3 (data-journaling) APM, appends to a file cannot be reordered and the `rename` operation cannot be split into smaller operations. An APM for a new file system can be easily derived using the *block order breaker* (BOB) tool [70].

PACE considers all consistent cuts in the execution to find globally reachable prefixes. On each such globally reachable prefix, PACE applies the APM (that specifies what file-system specific crash states are possible) to produce more states. The default APM used by PACE has few restrictions on the possible crash states. Intuitively, our default APM models a file system that provides the least guarantees when crashes occur but is still `POSIX` compliant. For simplicity, we refer to file-system related system calls issued by the application as *logical operations* and the smaller operations into which each logical operation is broken down as *micro operations*. We now describe our default APM.

Atomicity of operations. Applications may require a single logical operation such as *append* or *overwrite* to be atomically persisted for correctness. In the default APM used by PACE, all logical operations are broken into the following micro operations: *write_block*, *change_size*, *create_dir_entry*, and *delete_dir_entry*. For example, a logical truncate of a file will be broken into *change_size* followed by *write_block(random)* followed by *write_block(zeroes)*. Similarly, a rename will be broken into *delete_dir_entry(dest)* + *truncate if last link* followed by *create_dir_entry(dest)* followed by *delete_dir_entry(src)*. Overwrites, truncates, and appends are split into micro operations aligned at the block boundary or simply into three micro operations. PACE can generate crash states corresponding to different intermediate states of the logical operation.

Ordering between operations. Applications may require that a logical operation A be persisted before another logical operation B for correctness. To reorder operations, PACE considers each pair of operations (A, B) and applies all operations from the beginning of the trace until B except for A. This reordering produces a state corresponding to the situation where the node crashes after all operations up to B have been persisted but A is still not persisted. The ordering constraint for our default APM is as follows: all operations followed by an *fsync* on a file or directory F are ordered after the operations on F that precede the *fsync*.

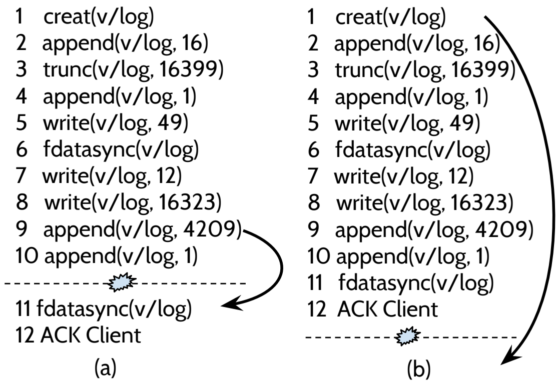
We now describe how applying an APM produces more states on a *single* machine. Consider the ZooKeeper protocol in which $<C_\phi, L_1, P_2, Q_\phi>$ is a globally reachable prefix. P has moved to P_2 by applying two `write` operations starting from its initial state P_ϕ. On applying the default APM onto the above prefix, PACE recognizes that on node P it is possible for the second write to reach the disk before the first one (by considering different ordering between two operations). Hence, it can *reorder* the first write after the second write on P. This resultant state is different from the prefix. In this resultant state, after recovery, P will see a file-system state where the second write to the log is persisted but effects of the first write are missing. If there were an `fsync` or `fdatasync` after the first write, then the default APM cannot and will not reorder the two write operations. This reordering is within a *single* node; similar reorderings can be exercised on all nodes.

Depending on the APM specification, logical operations can be partially persisted or reordered or both at each node in the system. Intuitively, applying an APM on a global prefix *relaxes* its constraints. This relaxation allows the APM to partially persist logical operations (atomicity) or reorder logical operations with one another (ordering). We refer to the relaxations allowed by an APM as *APM-allowed relaxations* or simply *APM relaxations*. For simplicity, we refer to this process of relaxing the constraints (by reordering and partially persisting operations) as applying that particular relaxation.

PACE can be configured with any APM. We find the most vulnerabilities with our default and ext2 APMs. We also report the vulnerabilities when PACE is configured with APMs of other commonly used file systems.

3.3 Protocol-Aware Exploration

While applying relaxations on a single node results in many persistent states for that node, PACE needs to consider applying different relaxations across every combination of nodes to find vulnerabilities. As a consequence, there are several choices for how PACE can apply relaxations. Consider a five node cluster and assume that n relaxations are possible in one node. Then, as-



```
1  creat(v/log)
2  append(v/log, 16)
3  trunc(v/log, 16399)
4  append(v/log, 1)
5  write(v/log, 49)
6  fdatasync(v/log)
7  write(v/log, 12)
8  write(v/log, 16323)
9  append(v/log, 4209)
10 append(v/log, 1)
-----------------------
11 fdatasync(v/log)
12 ACK Client
(a)
```

```
1  creat(v/log)
2  append(v/log, 16)
3  trunc(v/log, 16399)
4  append(v/log, 1)
5  write(v/log, 49)
6  fdatasync(v/log)
7  write(v/log, 12)
8  write(v/log, 16323)
9  append(v/log, 4209)
10 append(v/log, 1)
11 fdatasync(v/log)
12 ACK Client
-----------------------
(b)
```

Figure 3: **Local file-system update protocol on a single ZooKeeper node.** *The figure shows the sequence of file-system operations on a single ZooKeeper node. Operations 1 through 6 happen on node initialization and operations 7 through 12 when the client starts interacting. Several operations that happen on initialization are not shown for clarity. (a) and (b) show two different crash scenarios.*

suming there are no cross node dependencies, there are $\binom{5}{1} * n + \binom{5}{2} * n^2 + \binom{5}{3} * n^3 + \binom{5}{4} * n^4 + \binom{5}{5} * n^5$ ways of combining the relaxations across nodes. Even for a moderate n such as 20, there are close to 4 million states. A brute-force approach would explore all such states. We now explain how Pace prunes this space by using knowledge about the distributed protocols (such as agreement and leader election) employed by a system.

3.3.1 Replicated State Machine Approaches

We use the same ZooKeeper traces shown in Figure 2 for this discussion. For simplicity, we assume that there are odd number of nodes in the system.

ZooKeeper implements an atomic broadcast protocol which is required to run a replicated state machine (RSM) [45, 68, 90]. There are various paradigms to implement an RSM some of which include Paxos [51], Raft [68], and atomic broadcast [24]. Google's Chubby [15] implements a Paxos-like algorithm and LogCabin [57] implements Raft. An RSM system as a whole should continue to make progress as long as a *majority* of the nodes are operational and can communicate with each other and the clients [68].

Figure 3(a) shows the file-system operations on a single ZooKeeper node; network operations are not shown for clarity. The tenth operation appends one byte to the log to denote the commit of a transaction after which the file is forced to disk by the `fdatasync` call. It is possible for the tenth operation to reach the disk *before* the ninth operation and a crash can happen at this exact point before the `fdatasync` call. After this crash and subsequent restart, ZooKeeper would fail to start as it detects a checksum mismatch for the data written, and the node becomes unusable. The same reordering can happen on all nodes, rendering the entire cluster unusable.

In the simple case where this reordering happens on only one node, even though that single node would fail to start, the other two nodes still constitute a *majority* and so can elect a leader and make progress. Pace uses this knowledge about the protocol to eliminate testing cases where a reordering happens on only one node. Also, it is unnecessary to apply the relaxation on all three nodes as the cluster can become unavailable even when the relaxation is applied on just a majority (any two) of the nodes.

As another example, consider the same protocol but with a different crash that happens after the client is acknowledged, as shown in Figure 3(b). Once acknowledged, ZooKeeper guarantees that the data is replicated and persisted to disk on a majority of nodes. The directory entry for the log file has to be persisted explicitly by performing an `fsync` on the parent directory [1, 70] to ensure that the log file is present on disk even after a crash. However, ZooKeeper does not `fsync` the parent directory and so it is possible for the log file to go missing after a crash. On a single node, if the log file is lost, it does not lead to user-visible global data loss as the majority still has the log file. Similar to the unavailability case, a global data loss can happen if the same reordering happens on a majority of nodes even if the data exists on one other node where this reordering did not happen.

Thus, we observe that in any RSM system, it is required that a particular APM relaxation is applied on at least a majority of nodes for a vulnerability to be exposed globally. Also, it is unnecessary to apply an APM relaxation on all possible majority choices; for example, in a system with five nodes, applying a relaxation on three, four, or five nodes (all of which represent a majority) will expose the same vulnerability. This knowledge is not system-specific, but rather protocol-specific.

System-independent. LogCabin is a system similar to ZooKeeper that provides a configuration store on top of the consensus module but uses the Raft protocol to implement an RSM. When applying a particular APM relaxation, LogCabin can lose data. For this data loss vulnerability to be exposed, the relaxation has to be applied on at least a majority of the nodes. This observation is not specific to a particular system; rather, it holds true across ZooKeeper and LogCabin because both systems are RSM protocol implementations.

Using our observation, we derive the following rule that helps Pace eliminate a range of states: *For any RSM system with N replicas, check only states that would result when a particular APM relaxation is applied on an exact majority (where exactly $\lceil N/2 \rceil$ servers are chosen from N) of the nodes.* Note that there are $\binom{N}{\lceil N/2 \rceil}$ ways of choosing the exact majority.

We note that the pruning rule does not guarantee finding all vulnerabilities. It works because it makes an important assumption: the base consensus protocol is implemented correctly. Pace is not intended to catch bugs in consensus protocol implementations.

We now make a further observation about RSM protocols that can further reduce the state space. Consider the data loss vulnerability shown in Figure 3(b). Surprisingly, sometimes a global data loss may *not* be exposed even when the reordering happens on a majority. To see why, consider that the current leader (L) and the first follower (P) lose the log file as the *creat* operation is not persisted before the crash. In this case, the majority has lost the file. On recovery, the possibility of global data loss depends on *who is elected as the leader the next time*. Specifically, the data will be lost, if either L or P is elected as the new leader. On the other hand, if the second follower Q is elected as the leader, then the data will not be lost. In effect, the data will be lost if a node that lost its local data becomes the leader the next time, irrespective of the presence of the same data on other nodes.

In Raft, on detecting an inconsistency, the followers are forced to duplicate the leader's log (i.e., the log entries flow only outward from the leader) [68]. This enforcement is required to satisfy safety properties of Raft. While ZooKeeper's atomic broadcast (ZAB) does not explicitly specify if the log entries only flow outward from the leader, our experiments show that this is the case. Previous work also supports our observation [68].

This brings a question that counters our observation: *Why not apply the relaxation on any one node and make it the leader during recovery*? Consider the reordering shown in Figure 3(b). If this reordering happens on one node, that node will lose the log; it is *not* possible for this node to be elected the leader as other nodes would notice that this node has missing log entries and not vote for it. If this node is not elected the leader, then local data loss would not result in global data loss.

In contrast, if the log is lost on two nodes, the two nodes still constitute a majority and so one of them can become the leader and therefore override the data on the third node causing a global data loss. However, it is possible for the third node Q, where the data was not lost, to become the leader and so hide the global data loss.

Given this information, we observe that it is required only to check states that result from applying a particular APM relaxation on *any one* exact majority of the nodes. In a cluster of five nodes, there are $\binom{5}{3} = 10$ ways of choosing an exact majority and it is enough to check any one combination from the ten. To effectively test if a global vulnerability can be exposed, we strive to enforce the following: *when the cluster recovers from a crashed state, if possible, the leader should be elected from the set of nodes where the APM relaxation was applied.* Sometimes the system may constrain us from enforcing this; however, if possible, we enforce it automatically to drive the system into vulnerable situations.

From the two observations, we arrive at two simple, system-independent, and protocol-aware exploration rules employed by PACE to prune the state space and effectively search for undesired behaviors:

- **R1**: *For any RSM system with N servers where followers duplicate leader's log, generate states that would result if a particular APM relaxation is applied on any exact majority of the servers.*
- **R2**: *For all states generated using R1, if possible, enforce that the leader is elected from exact majority in which the APM relaxation was applied.*

Since we did not see popular practical systems that use RSM approaches where log entries can flow in both directions like in Viewstamped replication [55, 67] or where there can be multiple proposers at the same time like in Paxos, we have not listed the rules for them. We leave this generalization as an avenue for future work.

3.3.2 Other Replication Schemes

PACE also handles replicated systems that do not use RSM approaches: Redis, Kafka, and MongoDB. Applications belonging to this category do not strictly require a majority for electing a leader and committing transactions. For example, in Redis' default configuration, the master is fixed and cannot be automatically re-elected by a majority of slaves if the master fails. Moreover, it is possible for the master to make progress without the slaves. Similarly, Kafka maintains a metadata structure called the *in-sync replicas* and any node in this set can become the leader without consent from the majority.

Systems belonging to this category typically force slaves to sync data from the master. Hence, any problem in the master can easily propagate to the slaves. This hints that applying APM relaxations on the master is necessary. Next, since our workloads ensure that the data is synchronously replicated to all nodes, it is unacceptable to read stale data from the slaves once an acknowledgment is received. This hints that applying APM relaxations on any slave and subsequent reads from the slave can expose the stale data problem. Since systems of this type can make progress even if one node is up, we need to apply APM relaxations on all the nodes to expose cluster unavailability vulnerabilities.

For applications of this type, PACE uses a combination of the following rules to explore the state space:

- **R3**: *Generate states that result when a particular relaxation is applied on the master.*
- **R4**: *Generate states that result when a particular relaxation is applied on any one slave.*
- **R5**: *Generate states that result when a particular relaxation is applied on all nodes at the same time.*

In Redis, we use $R3$ and $R4$ but *not* $R5$: we use $R3$ to impose APM relaxations only on the master because the cluster can become unavailable for writes if only the master fails; we use $R4$ as reads can go to slaves. Similarly, in Kafka, we use $R3$ and $R5$ and *not* $R4$: we do not use $R4$ because all reads and writes go only through the leader; we use $R5$ to test states where the entire cluster can become unavailable because the cluster will be usable even if one node functions. MongoDB can be configured in many ways. We configure it much like an RSM system where it requires a majority for leader election and writes; hence, we use $R1$ and $R2$.

Examining a new distributed system with PACE requires developers to only understand whether the system implements a replicated state machine or not and how the master election works. Once this is known, PACE can be easily configured with the appropriate set of pruning rules. We believe that PACE can be readily helpful to developers given that they already know their system's protocols. We reiterate that the pruning rules do not guarantee finding all vulnerabilities; rather, they provide a set of guidelines to quickly search for problems. In the worst case, if no properties are known about a protocol, PACE can work in brute-force mode to find vulnerabilities.

3.3.3 Effectiveness of Pruning

To demonstrate the effectiveness of our pruning rules, we explored crash states of Redis and LogCabin with PACE and the brute-force approach. In Redis, for a simple workload on a three node cluster, brute-force needs to check 11,351 states whereas PACE only needs to check 1009 states. While exploring 11× fewer states, PACE found the same three vulnerabilities as the brute-force approach. In LogCabin, PACE discovers two vulnerabilities, checking 27,713 states in eight hours; the brute-force approach did not find any new vulnerabilities after running for over a week and exploring nearly 900,000 states. The reduction would be more pronounced as the number of nodes in a system increases.

3.4 Limitations and Caveats

PACE is *not complete* – it can miss vulnerabilities. Specifically, PACE exercises only one and the same reordering at a time across the set of nodes. For instance, consider two reorderings r_i and r_j. It is possible that no vulnerability is seen if r_i or r_j is applied individually on two nodes. But when r_i is applied on one node and r_j on the other, then it may lead to a vulnerability. PACE would miss such vulnerabilities. Note that if r_i and r_j can both individually cause a vulnerability, then PACE would catch both of them individually. This is a limitation in implementation and not a fundamental one. There is *no* similar limitation with partially persisting operations (i.e., PACE can partially persist different operations across nodes). Also, PACE does not focus on finding bugs in agreement protocols. We expand more on this topic later (§5).

System	Configuration	Workload	Checker
Redis	*appendfsync=always, min-slaves-to-write=2* and *wait*	update existing	old and new data (master and slave), *check-aof*, *check-dump*
ZooKeeper	Default	update existing	old and new data
LogCabin	Default	update existing	old and new data
etcd	Default	update existing	old and new data
RethinkDB	*durability=hard, writeack=majority*	update existing, insert new	old and new data
MongoDB	*W=3, journal=true*	update existing	old and new data
iNexus	Default	update existing, insert new	old and new data
Kafka	*flush.interval.msgs=1, min in-sync replicas=3, DirtyElection=False*	create topic, insert message	topic and message

Table 1: **Configurations, Workloads, and Checkers.** *The table shows the configuration, workloads and checkers for each system. We configured all systems with three nodes. The configuration settings ensure data is synchronously replicated and flushed to disk.*

4 Application Vulnerabilities Study

We studied eight widely used distributed systems spanning different domains including database caches (Redis v3.0.4), configuration stores (ZooKeeper v3.4.8, LogCabin v1.0.0, etcd v2.3.0), real-time databases (RethinkDB v2.2.5), document stores (MongoDB v3.0.11), key-value stores (iNexus v0.13), and message queues (Kafka v0.9.0). We tested MongoDB with two storage engines: WiredTiger [64] (MongoDB-WT) and RocksDB [86] (MongoDB-R). PACE found **26** unique vulnerabilities across the eight systems.

We first describe the workloads and checkers we used to detect vulnerabilities (§4.1). We then present a few example protocols and vulnerabilities to give an intuition of our methodology and the types of vulnerabilities discovered (§4.3). We then answer three important questions: Are there common patterns in file-system requirements (§4.4)? What are the consequences of the vulnerabilities discovered by PACE (§4.5)? How many vulnerabilities are exposed on real file systems (§4.6)? We then describe our experience with reporting the vulnerabilities to application developers (§4.7). We finally conclude by discussing the implications of our findings and the difficulties in fixing the discovered vulnerabilities (§4.8).

4.1 Application Workloads and Checkers

Most systems have configuration options that change user-level guarantees. We configured each system to provide the highest level of safety guarantees possible. When guarantees provided are unclear, our checkers check for typical user expectations; for example, data acknowledged as committed should not be lost in any case or the cluster should be available after recovering from crashes. Even though some applications do not explicitly guarantee such properties, we believe it is reasonable to test for such common expectations.

To test a system, we first construct a workload. Our workloads are not specifically crafted to expose vulnera-

```
## Workload ##
# Start cluster
# Insert new data
zk = client(hosts=server_ips)
zk.set("/mykey", "newvalue")
pace.acknowledged = True
# Stop cluster
```

```
## Checker ##
# Start cluster
# Check for data
retry_policy = retry(max_tries = r, delay = d,
backoff = b)
zk = client(hosts=server_ips, retry_policy)
ret, stat = zk.get("/mykey")
if request succeeded:
  if pace.acknowledged and ret == None:
    return ’data loss new commit’
  if pace.acknowledged and ret != ’newvalue’:
    return ’corrupt’
  if not pace.acknowledged and ret == None:
    return ’data loss old commit’
else:
  return ’unavailable’
return ’correct’
# Stop cluster
```

Listing 1: **Workload and Checker.** *Simplified workload and checker for ZooKeeper.*

bilities, but rather are very natural and simple. Our workloads insert new data or update existing data and record the acknowledgment from the cluster. They are usually about 30-40 LOC.

To check each crash state, we implement a checker. The checker is conceptually simple; it starts the cluster with the crash state produced by PACE and checks for correctness by reading the data updated by the workload. If the data is lost, corrupted, or not retrievable, the checker flags the crash state incorrect. Further, our checkers invoke recovery tools mentioned in applications' documentation if an undesired output is observed. If the problem is fixed after invoking the recovery tool, then it is *not* reported as a vulnerability. Our checkers are about 100 LOC. Table 1 shows the configurations (that achieve the strongest safety guarantees), workloads, and checkers for all systems. Listing 1 shows the simplified pseudocode of the workload and the checker for ZooKeeper.

4.2 Vulnerability Accounting

A system has a crash vulnerability if a crash exposes a user-level guarantee violation. Counting such vulnerable places in the code is simple for single-machine applications. In a distributed system, multiple copies of the same code execute and so PACE needs to be careful in how it counts *unique* vulnerabilities.

We count only unique combinations of states that expose a vulnerability. Consider a sequence *S1* that creates (C), appends (A), and renames (R) a file. Assume that a node will not start if it crashes after C but before R. Assume there are three nodes in an RSM system and two crash after C but before R. In this case, the cluster can become unusable in four ways (C-C, CA-CA, C-CA, CA-C). We count all such instances as one vulnerability. If the third node crashes within this sequence, it will also be mapped onto the same vulnerability. If there is another different sequence *S2* that causes problems, a vulnerability could be exposed in many different ways as one node can crash within *S1* and another within *S2*. We associate all such combinations to two unique vulnerabilities, attributing to the atomicity of *S1* and *S2*.

PACE also associates each vulnerability with the application source code line using the stack trace information obtained during tracing. When many vulnerabilities map to the same source line, PACE considers that a single vulnerability. When we are unable to find the exact source lines for two different vulnerabilities, we count them as one. We note that our way of counting vulnerabilities results in a conservative estimate.

4.3 Example Protocols and Vulnerabilities

Figure 4 shows protocols and vulnerabilities in ZooKeeper, etcd, Redis, and Kafka. Due to space constraints, we show protocols only for four systems; protocol diagrams for other systems are publicly available [2].

RSM systems where vulnerabilities are exposed when APM relaxations are applied on a majority of nodes are represented using a grid. Figure 4(a) and 4(b) show the combinations of persistent states across two nodes in a three node ZooKeeper and etcd cluster, respectively. Operations that change persistent state are shown on the left (for one node) and the top (for the other node). A box *(i,j)* corresponds to a crash point where the first node crashes at operation *i* and the second at *j*. At each such crash point, PACE reorders other operations, or partially persists operations or both. A grey box denotes that the distributed execution did not reach that combination of states. A white box means that after applying all APM relaxations, PACE was unable to find a vulnerability. A black box denotes that when a specific relaxation (shown on the left) is applied, a vulnerability is exposed.

As shown in Figure 4(a), to maintain proposal information, ZooKeeper appends epoch numbers to temporary files, and renames them. If the renames are not atomic or reordered after a later write, the cluster becomes unavailable. If a log file creation and a subsequent append of header metadata are not atomically persisted, then the nodes fail to start. Similarly, the immediate truncate after log creation has to be atomically persisted for correct startup. Writes and appends during transactions, if reordered, can also cause node startup failures. ZooKeeper can lose data as it does not `fsync` the parent directory when a log is created.

Figure 4(b) shows the protocol and vulnerabilities in etcd. etcd creates a temporary write-ahead log (WAL),

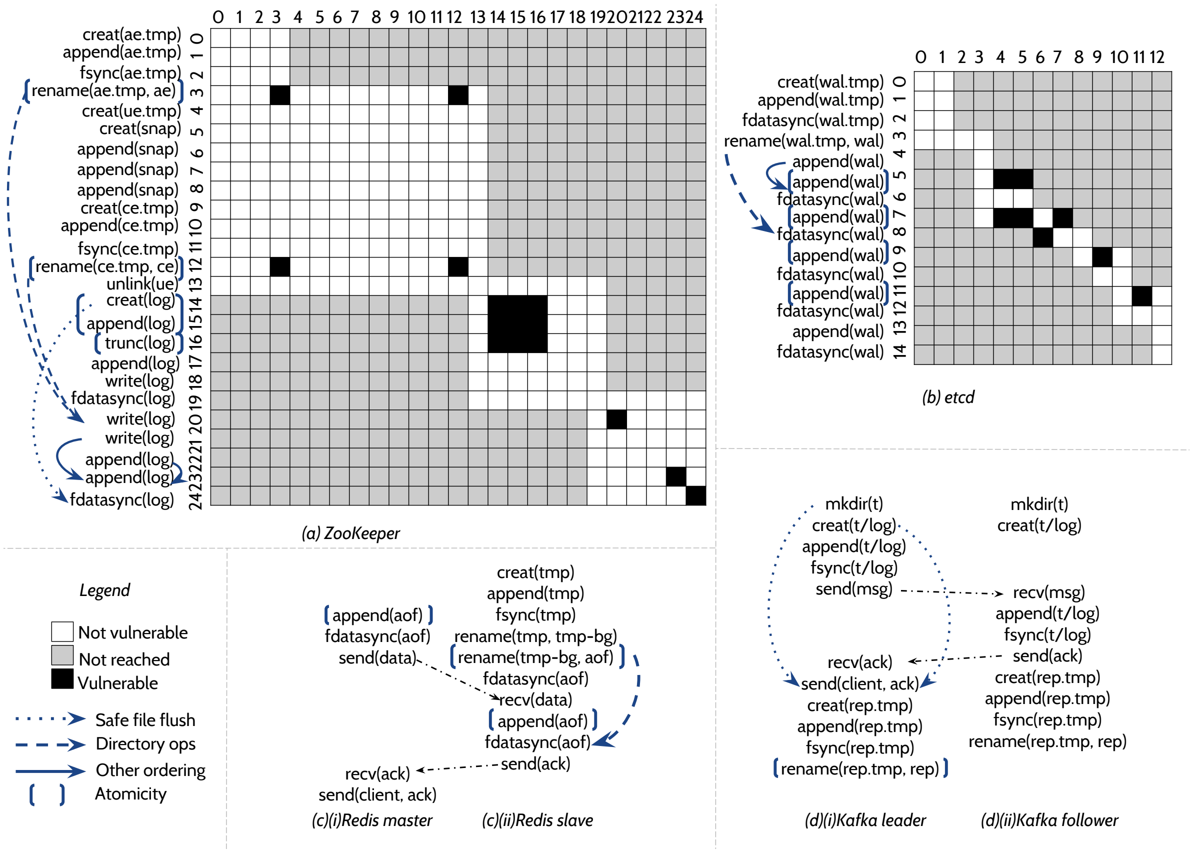


Figure 4: **Protocols and Vulnerabilities.** *(a), (b), (c), and (d) show protocols and vulnerabilities in ZooKeeper, etcd, Redis, and Kafka, respectively. States that are not vulnerable, that were not reached in the execution, and that are vulnerable are shown by white, grey, and black boxes, respectively. The annotations show how a particular state becomes vulnerable. In Zookeeper, box (24, 24) is vulnerable because both nodes crash after the final fdatasync but before the log creation is persisted. Atomicity vulnerabilities are shown with brackets enclosing the operations that need to be persisted atomically. The arrows show the ordering dependencies in the application protocol; if not satisfied, vulnerabilities are observed. Dotted, dashed, and solid arrows represent safe file flush, directory operation, and other ordering dependencies, respectively.*

appends some metadata, and renames it to create the final WAL. The WAL is appended, flushed, and then the client is acknowledged. We find that etcd cluster becomes unavailable if crashes occur when the WAL is appended; the nodes fail to start if the appends to the WAL are reordered or not persisted atomically. Also, if the rename of the WAL is reordered, a global data loss is observed.

Non-RSM systems where vulnerabilities are exposed even when relaxations are applied on a single machine are shown using trace pairs. As shown in Figure 4(c), Redis uses an append-only file to store user data. The master appends to the file and sends the update to slaves. Slaves, on startup, rewrite and rename the append-only file. When the master sends new data, the slaves append it to their append-only file and sync it. After the slaves respond, the client is acknowledged. Data loss windows are seen if the rename of the append-only file is not atomic or reordered after the final fdatasync. When the append is not atomic on the master, a user-visible silent corruption is observed. Moreover, the corrupted data is propagated from the master to the slaves, overriding their correct data. The same append (which maps to the same source line) on the slave results in a window of silent corruption. The window closes eventually since the slaves sync the data from the master on startup.

Figure 4(d) shows the update protocol of Kafka. Kafka creates a log file in the topic directory to store messages. When a message is added, the leader appends the message and flushes the log. It then contacts the followers which perform the same operation and respond. After acknowledging the client, the replication offset (that tracks which messages are replicated to other brokers) is appended to a temporary file, flushed, and renamed to the replication-offset-checkpoint file. The log can be lost after a crash because its parent directory is not flushed after the log creation. If the log is lost on the master, then the data is globally lost since the master instructs the slaves also to drop the messages in the log. Similarly, Kafka can lose a message topic altogether since the parent directory of the topic directory is not explicitly flushed.

We observe that some systems (e.g., Redis, Kafka) do not effectively use redundancy as a source of recovery.

	FS Requirements							Failure Consequences								
		Atomicity			Ordering				Data loss		Unavailable		Window			
System	**Inter-syscall atomicity**	Appends and truncates	Rename (dest link absent)	Rename (dest link exists)	Safe file flush	Directory ops	Other	**Silent corruption**	Old commit	New commit	Metadata corruption	User data corruption	Corruption	Data loss old commit	Data loss new commit	**Unique vulnerabilities**
Redis		1		1		1		1					1	1	1	**3**
ZooKeeper	1	1		1	1	1	1			1	4	1				**6**
LogCabin	1		1						1		1			1		**2**
etcd		1				1	1			1	1	2				**3**
RethinkDB																
MongoDB-WT				1							1					**1**
MongoDB-R		1		1	1	1	1			3	3					**5**
iNexus				1		1	1		1	1	2					**3**
Kafka				1	2					3						**3**
Total	2	4	1	6	4	5	4	1	2	9	12	3	1	2	1	**26**

Table 2: **Vulnerabilities - Types and Consequences.** *The table shows the unique vulnerabilities categorized by file-system requirements and consequences.*

For instance, in these systems, a local problem (such as a local corruption or data loss) which results due to a relaxation on a single node, can easily become a global vulnerability such as a user-visible silent corruption or data loss. In such situations, these systems miss opportunities to use other intact replicas to recover from the local problem. Moreover, such local problems are propagated to other intact replicas, overriding their correct data.

4.4 Patterns in File-system Requirements

Table 2 shows file-system requirements across systems. We group the results into three patterns:

Inter-Syscall Atomicity. ZooKeeper and LogCabin require inter system call atomicity (multiple system calls need to be atomically persisted). In both these systems, when a new log file is initialized, the creat and the initial append of the log header need to be atomically persisted. If the log initialization is partially persisted, the cluster becomes unavailable. Vulnerabilities due to inter system call atomicity requirements can occur on all file systems irrespective of how they persist operations.

Atomicity within System calls. We find that seven applications require system calls to be atomically persisted. Eleven unique vulnerabilities are observed when system calls are not persisted atomically. Six out of the eleven vulnerabilities are dependent on atomic replace by rename (destination link already exists), one on atomic create by rename (destination link does not exist), and four on atomic truncates or appends. Four applications require appends or truncates to be atomic. Redis, ZooKeeper, and etcd can handle appended portions filled with zeros but not garbage.

Ordering between System calls. Six applications expect system calls to be persisted in order. Kafka and ZooKeeper suffer from data loss since they expect the safe file flush property from the file system. To persist a file's directory entry, the parent directory has to be explicitly flushed to avoid such vulnerabilities. We found that reordering directory operations can cause vulnerabilities. We found that five applications depend on ordered renames: Redis exhibits a data loss window, etcd permanently loses data, ZooKeeper, MongoDB-R, and iNexus fail to start. Four applications require other operations (appends and writes) to be ordered for correct behavior.

4.5 Vulnerability Consequences

Table 2 shows the vulnerability consequences. We find that all vulnerabilities have severe consequences like silent corruption, data loss, or cluster unavailability. Redis silently returns and propagates corrupted data from the master to slaves even if slaves have correct older version of data. Redis also has a silent corruption window when reads are performed on slaves. While only one system silently corrupts and propagates corrupted data, six out of eight systems are affected by permanent data loss. Depending on the crash state, previously committed data can be lost when new data is inserted or the newly inserted data can be lost after acknowledgment. Redis exhibits a data loss window that is exposed when reads are performed on the slaves. As slaves continuously sync data from the master, the window eventually closes.

Cluster unavailability occurs when nodes fail to start due to corrupted application data or metadata. ZooKeeper and etcd fail to start if CRC checksums mismatch in user data. MongoDB-WT fails to start if the *turtle* file is missing and MongoDB-R fails to start if the *sstable* file is missing or there is a mismatch in the *current* and *manifest* files. LogCabin and iNexus skip log entries when checksums do not match but fail to start if metadata is corrupted. LogCabin fails to start when an unexpected segment metadata version is found. Similarly, ZooKeeper fails to start on unexpected epoch values. While some of these scenarios can be fixed by expert application users, the process is intricate and error prone.

We note that the vulnerabilities are specific to our simple workloads and all vulnerabilities reported by PACE have harmful consequences. More complex workloads and checkers that assert more subtle invariants are bound to find more vulnerabilities.

4.6 Impact on Real File Systems

We configured PACE with APMs of real file systems. Table 3 shows the vulnerabilities on each file system. We observe that many vulnerabilities can occur on all examined file systems. Only two vulnerabilities are observed in ext3-j (data-journaling) as all operations are persisted in order. All vulnerabilities that occur on our default

	ext2	ext3-w	ext3-o	ext4-o	ext3-j	btrfs
Redis	3	1				1
ZooKeeper	6	3	1	1	1	3
LogCabin	2	1	1	1	1	1
etcd	3	2				
MongoDB-WT	1					
MongoDB-R	5	2	2	2		3
iNexus	2		1	1		2
Kafka	3					
Total	26	9	5	5	2	10

Table 3: **Vulnerabilities on Real File Systems.** *The table shows the number of vulnerabilities on commonly used file systems.*

APM are also exposed on ext2. Applications are vulnerable even on Linux's default file system (ext4 ordered mode). Many of the vulnerabilities are exposed on btrfs as it reorders directory operations. In summary, the vulnerabilities are exposed on many current file systems on which distributed storage systems run today.

4.7 Confirmation of Problems Found

We reported 18 of the discovered vulnerabilities to application developers. We confirmed that the reported issues cause serious problems (such as data loss and unavailability) to users of the system. Seven out of the 18 reported issues were assigned to developers and fixed [32–34, 56, 87, 88]. Another five issues have been acknowledged or assigned to developers. Out of this five, two in Kafka were already known [48]. Other issues are still open and under consideration. We found that distributed storage system developers, in general, are responsive to such bug reports for two reasons. First, we believe developers consider crashes very important in distributed systems compared to single-machine applications. Second, the discovered vulnerabilities due to crashes affect their users directly (for example, data loss and cluster unavailability).

We found that users and random-crash testing have also occasionally encountered the same vulnerabilities that were systematically discovered by PACE. However, PACE diagnoses the underlying root cause and provides information of the problematic source code line, easing the process of fixing these vulnerabilities.

4.8 Discussion

We first discuss the immediate implications of our findings in building distributed storage systems atop file systems. Next, we discuss the difficulties in fixing some of the discovered vulnerabilities.

4.8.1 Implications

We find that redundancy by replication is not the panacea for constructing reliable storage systems. Although replication can help with single node failures, correlated crashes still remain a problem. We find that application protocols, when driven to corner cases, can often override correct versions of data with corrupted data or older versions without considering how the system reached such a state. For example, Redis and Kafka can propagate corrupted data and data loss to slaves, respectively. Similarly, RSM systems override correct newer versions of data on other nodes when a majority of nodes have lost the data; a better recovery strategy could use the unaffected replicas to fix the loss of acknowledged data even if the data is lost on a majority of nodes. We believe replication protocols and local storage protocols should be designed in tandem to avoid such undesired behaviors.

System designers need to be careful about two problems when embracing *layered* software. First, the reliability of the entire system depends on individual components. MongoDB's reliability varies depending on the storage engine (WiredTiger or RocksDB). Second, separate well-tested components when integrated can bring unexpected problems. In the version of MongoDB we tested, we found that correct options are not passed from upper layers to RocksDB, resulting in a data loss. Similarly, iNexus uses a modified version of LevelDB which does not flush writes to disk when transactions commit. Applications need to clearly understand the guarantees provided by components when using them.

We find that a few applications are overly cautious in how they update file-system state. LogCabin flushes files and directories after every operation. Though this avoids many reordering vulnerabilities, it does not fix atomicity vulnerabilities. Issuing `fsync` at various places does not completely avoid reliability problems. Also, the implication of too much caution is clear: low performance. While this approach is reasonable for configuration stores, key-value stores need a better way to achieve the same effect without compromising performance.

All modern distributed storage system run on top of a variety of file systems that provide different crash guarantees. We advocate that distributed storage systems should understand and document on which file systems their protocols work correctly to help practitioners make conscious deployment decisions.

4.8.2 Difficulties in Fixing

Now we discuss the difficulties in fixing the discovered vulnerabilities. The effort to fix the vulnerabilities varies significantly. While some of them are simple implementation-level fixes, many of them are fundamental problems that require rethinking how distributed crash recovery protocols are designed.

Some vulnerabilities (such as those due to non-atomic renames) are automatically masked in modern file systems; these possess a practical concern only when the applications are run on file systems that do not provide such guarantees (e.g., ext2). While only some vulnerabilities can be easily fixed, many vulnerabilities are fundamentally hard to fix and they fall into three categories.

First, some vulnerabilities cannot be fixed by current file system interfaces or straightforward changes in application local update protocols. For example, consider the inter-syscall atomicity vulnerabilities and the non-atomic multi-block appends and truncates. These vulnerabilities are exposed on all current file systems since `POSIX` does not provide a way to atomically persist multiple system calls or a write that spans multiple blocks. While a different interface that allows multiple system calls to be atomically persisted can help, such an interface is far from reality in current commodity file systems.

Second, many reordering vulnerabilities can be fixed by carefully issuing `fsync` at correct places in the local update protocol. However, applications may not be willing to do so, given the clear performance impact in the common case update protocol code.

Third, sometimes the programming environment may constrain applications from utilizing some file system interfaces, leading to vulnerabilities. For example, consider the safe file flush and directory operation reordering vulnerabilities in ZooKeeper and Kafka. These vulnerabilities arise because these systems are written in Java in which `fsync` cannot be readily issued on directories.

In all the above cases, simply fixing the local update protocols is not a feasible solution. Fixing these fundamental problems requires carefully designing local and global recovery protocols that interact correctly to fix the problem using other intact replicas.

5 Related Work

Recent work has demonstrated that file-system behaviors vary widely [4, 14, 70–73]. We derive our APM specifications from our previous work [70]. Our previous work also developed Alice to uncover single-machine crash vulnerabilities. Tools like Alice cannot be directly applied to distributed systems as they do not track cross node dependencies. If applied, such tools may report spurious vulnerabilities. Although such tools can be applied in stand-alone mode like in ZooKeeper [99], many code paths would not be exercised and thus miss important vulnerabilities. Zheng et al. [98] find crash vulnerabilities in databases. Unlike our work, Zheng et al. do not systematically explore all states that can occur in an execution. They find vulnerabilities that can commonly occur: they do not model file-system behavior closely and therefore cannot explore all corner cases.

PACE is complementary to distributed model checkers [29, 40, 42, 54, 95, 96]: bugs due to file-system behaviors discovered by PACE cannot be discovered by existing model checkers and bugs due to network message re-orderings cannot be discovered by PACE. Distributed model checkers use dynamic partial-order based techniques to reduce state space explosion. SAMC [54] can also induce crashes and reboots in addition to reordering messages. To reduce state space, SAMC uses semantic information which requires testers to write protocol-specific rules for a target system. PACE uses only high-level protocol-awareness to prune the state space and does not require any code as input. Jepsen [50] is a tool that tests distributed systems under faulty networks and partial failures. Similar to distributed model checkers, such tools are complementary to PACE.

Previous tools focus solely on either single-node file system behavior or distributed consensus and thus cannot understand the interaction of distributed recovery protocol and a local-storage protocol. To our knowledge, our work is the first to consider file-system behaviors in the context of distributed storage systems. It is difficult for other distributed model checkers to reproduce the vulnerabilities found by PACE because they run the system on top of already implemented storage stacks. PACE models the file system used by the distributed system and thus can check how a distributed storage system will work on any current or future file system.

6 Conclusion

Modern distributed storage systems suffer from correlated crash vulnerabilities and subtleties in local file-system behavior influence the correctness of distributed update protocols. We present PACE a tool that can effectively search for correlated crash vulnerabilities by pruning the search space. We study eight popular distributed storage systems using PACE and find 26 unique vulnerabilities. As modern distributed storage systems are becoming the primary choice for storing and managing critical user data, tools such as PACE are increasingly vital to uncover reliability problems. Source code of PACE, workloads, checkers, and details of the discovered vulnerabilities are publicly available at `http://research.cs.wisc.edu/adsl/Software/pace/`.

Acknowledgments

We thank the anonymous reviewers and Jeffrey Chase (our shepherd) for their insightful comments. We thank the members of the ADSL 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 [85] for providing a great computing 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] Necessary step(s) to synchronize filename operations on disk. http://austingroupbugs.net/view.php?id=672.

[2] Pace Tool and Results. http://research.cs.wisc.edu/adsl/Software/pace/.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[4] 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, pages 20–20, Berkeley, CA, USA, 2015. USENIX Association.

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

[6] Apache. Kafka. http://kafka.apache.org/.

[7] Apache Cassandra. Cassandra Replication. http://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureDataDistributeReplication_c.html.

[8] ArchLinux. f2fs. https://wiki.archlinux.org/index.php/F2FS.

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

[10] Özalp Babaoğlu and Keith Marzullo. Distributed Systems (2Nd Ed.). pages 55–96, 1993.

[11] Mehmet Bakkaloglu, Jay J Wylie, Chenxi Wang, and Gregory R Ganger. On Correlated Failures in Survivable Storage Systems . Technical report, DTIC Document, 2002.

[12] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Trans. Comput. Syst.*, 5(1):47–76, January 1987.

[13] Andrew D. Birrell, Roy Levin, Michael D. Schroeder, and Roger M. Needham. Grapevine: An Exercise in Distributed Computing. *Commun. ACM*, 25(4):260–274, April 1982.

[14] James Bornholt, Antoine Kaufmann, Jialin Li, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Specifying and Checking File System Crash-Consistency Models. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 83–98, New York, NY, USA, 2016. ACM.

[15] Mike Burrows. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[16] Marco Canini, Vojin Jovanovic, Daniele Venzano, Gautam Kumar, Dejan Novakovic, and Dejan Kostic. Checking for Insidious Faults in Deployed Federated and Heterogeneous Distributed Systems. Technical report, 2011.

[17] Cassandra. Apache Cassandra. https://academy.datastax.com/resources/brief-introduction-apache-cassandra.

[18] K. Mani Chandy and Leslie Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, February 1985.

[19] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. pages 15–15, 2006.

[20] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the Frequency of Data Loss in Cloud Storage. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 37–48, Berkeley, CA, USA, 2013. USENIX Association.

[21] ComputerWorldUK. Lightning strikes Amazon and Microsoft data centres. http://www.computerworlduk.com/galleries/infrastructure/ten-datacentre-disasters-that-brought-firms-offline-3593580/#5.

[22] Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008.

[23] CoreOS. etcd. https://coreos.com/etcd/.

[24] Flaviu Cristian, Houtan Aghili, Raymond Strong, and Danny Dolev. *Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement*. Citeseer, 1986.

[25] DataCenterDynamics. Lessons from the Singapore Exchange failure. http://www.datacenterdynamics.com/power-cooling/lessons-from-the-singapore-exchange-failure/94438.fullarticle.

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

[27] Datastax. Netflix Cassandra use case. http://www.datastax.com/resources/casestudies/netflix.

[28] 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. volume 41, pages 205–220, New York, NY, USA, October 2007. ACM.

[29] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering Bugs in Distributed Storage Systems during Testing (Not in Production!). In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 249–262, Santa Clara, CA, Feb 2016. USENIX Association.

[30] Docker. Docker. https://www.docker.com/.

[31] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-value Store. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[32] etcd. Possible cluster unavailability on few file systems. https://github.com/coreos/etcd/issues/6379.

[33] etcd. Possible cluster unavailbility. https://github.com/coreos/etcd/issues/6378.

[34] etcd. Possible data loss – fsync parent directories. https://github.com/coreos/etcd/issues/6378.

[35] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Presented as part of the 9th USENIX Symposium on Operating Systems Design and Implementation*, Berkeley, CA, 2010. USENIX.

[36] Dennis Geels, Gautam Altekar, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Friday: Global Comprehension for Distributed Replay. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 21–21, Berkeley, CA, USA, 2007. USENIX Association.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[38] Google. Google Cloud Status. https://status.cloud.google.com/incident/compute/15056#5719570367119360.

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

[40] Rachid Guerraoui and Maysam Yabandeh. Model Checking a Networked System Without the Network. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 225–238, Berkeley, CA, USA, 2011. USENIX Association.

[41] Haryadi S. Gunawi, Vijayan Prabhakaran, Swetha Krishnan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Improving File System Reliability with I/O Shepherding. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles 2007, SOSP 2007, Stevenson, Washington, USA, October 14-17, 2007*, pages 293–306, 2007.

[42] 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 Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[43] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly Durable, Decentralized Storage Despite Massive Correlated Failures. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 143–158, Berkeley, CA, USA, 2005. USENIX Association.

[44] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael L. Roberts, Srinath Setty, and Brian Zill. IronFleet: Proving Practical Distributed Systems Correct. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 1–17, New York, NY, USA, 2015. ACM.

[45] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[46] iNexus. iNexus. https://github.com/baidu/ins.

[47] Kafka. Kafka Disks and Filesystem. https://kafka.apache.org/081/ops.html.

[48] Kafka. Possible data loss. https://issues.apache.org/jira/browse/KAFKA-4127.

[49] Kimberley Keeton, Cipriano Santos, Dirk Beyer, Jeffrey Chase, and John Wilkes. Designing for Disasters. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 59–62, Berkeley, CA, USA, 2004. USENIX Association.

[50] Kyle Kingsbury. Jepsen. http://jepsen.io/.

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

[52] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, July 1982.

[53] Butler Lampson and Howard Sturgis. *Crash recovery in a distributed data storage system*. Xerox Palo Alto Research Center Palo Alto, California, 1979.

[54] 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 USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 399–414, Berkeley, CA, USA, 2014. USENIX Association.

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

[56] LogCabin. Cluster unavailable due to power failures. https://github.com/logcabin/logcabin/issues/221.

[57] LogCabin. LogCabin. https://github.com/logcabin/logcabin.

[58] Marshall Kirk McKusick and Jeffery Roberson. Journaled Soft-updates. *Proceedings of EuroBSDCon*, 2010.

[59] MongoDB. Introduction to MongoDB. https://docs.mongodb.org/manual/introduction/.

[60] MongoDB. MongoDB. https://www.mongodb.org/.

[61] MongoDB. MongoDB at ebay. https://www.mongodb.com/presentations/mongodb-ebay.

[62] MongoDB. MongoDB Platform Specific Considerations. https://docs.mongodb.org/manual/administration/production-notes/#platform-specific-considerations.

[63] MongoDB. MongoDB Replication. https://docs.mongodb.org/manual/replication/.

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

[65] Suman Nath, Haifeng Yu, Phillip B. Gibbons, and Srinivasan Seshan. Subtleties in Tolerating Correlated Failures in Wide-area Storage Systems. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 17–17, Berkeley, CA, USA, 2006. USENIX Association.

[66] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[67] Brian M. Oki and Barbara H. Liskov. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[68] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, pages 305–320, Berkeley, CA, USA, 2014. USENIX Association.

[69] David A. Patterson, Garth Gibson, and Randy H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[70] 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 USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, Berkeley, CA, USA, 2014. USENIX Association.

[71] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency. *Communications of the ACM*, 58(10):46–51, October 2015.

[72] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Crash Consistency: Rethinking the Fundamental Abstractions of the File System. *Communications of the ACM*, 13(7), July 2015.

[73] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Joo-Young Hwang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Towards Efficient, Portable Application-level Consistency. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems*, HotDep '13, pages 8:1–8:6, New York, NY, USA, 2013. ACM.

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

[75] Redis. Introduction to Redis. http://redis.io/topics/introduction.

[76] Redis. Redis. http://redis.io/.

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

[78] Redis. Redis at GitHub. http://nosql.mypopescu.com/post/1164218362/redis-at-github.

[79] Redis. Redis at Pinterest. http://highscalability.com/blog/2012/4/9/the-instagram-architecture-facebook-bought-for-a-cool-billio.html.

[80] Redis. Redis Replication. http://redis.io/topics/replication.

[81] Redis. Virtual Memory – Redis . http://redis.io/topics/virtual-memory.

[82] Redis. Who's using Redis? http://redis.io/topics/whos-using-redis.

[83] RethinkDB. RethinkDB. https://www.rethinkdb.com/.

[84] RethinkDB. RethinkDB Replication. https://www.rethinkdb.com/docs/sharding-and-replication/.

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

[86] RocksDB. RocksDB. http://rocksdb.org/blog/1967/integrating-rocksdb-with-mongodb-2/.

[87] Mongo RocksDB. Data loss – fsync parent directory on file creation and rename. https://github.com/mongodb-partners/mongo-rocks/issues/35.

[88] Mongo RocksDB. Mongodb - rocksdb data loss bug. https://groups.google.com/forum/#!topic/mongodb-dev/X9LQOorieas.

[89] Yasushi Saito and Marc Shapiro. Optimistic Replication. *ACM Comput. Surv.*, 37(1):42–81, March 2005.

[90] Fred B. Schneider. Implementing Fault-tolerant Services Using the State Machine Approach: A Tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[91] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, pages 172–183, 1995.

[92] TheRegister. Admin downs entire Joyent data center. http://www.theregister.co.uk/2014/05/28/joyent_cloud_down/.

[93] Twitter. Twitter Blogs. https://blog.twitter.com/2015/handling-five-billion-sessions-a-day-in-real-time.

[94] Maysam Yabandeh, Nikola Knežević, Dejan Kostić, and Viktor Kuncak. Predicting and Preventing Inconsistencies in Deployed Distributed Systems. *ACM Trans. Comput. Syst.*, 28(1):2:1–2:49, August 2010.

[95] Maysam Yabandeh and Dejan Kostić. DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems. Technical report, 2009.

[96] 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 USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association.

[97] YCombinator. Joyent us-east-1 rebooted due to operator error. https://news.ycombinator.com/item?id=7806972.

[98] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W. Zhao, and Shashank Singh. Torturing Databases for Fun and Profit. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 449–464, Berkeley, CA, USA, 2014. USENIX Association.

[99] ZooKeeper. ZooKeeper Standalone Operation. https://zookeeper.apache.org/doc/r3.3.3/zookeeperStarted.html#sc_InstallingSingleMode.