An Analysis of Network-Partitioning Failures in Cloud Systems

Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany, ***University of Waterloo***

https://www.usenix.org/conference/osdi18/presentation/alquraan

This paper is included in the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18).

October 8–10, 2018 • Carlsbad, CA, USA

ISBN 978-1-939133-08-3

Open access to the Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.

An Analysis of Network-Partitioning Failures in Cloud Systems

Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, Samer Al-Kiswany
University of Waterloo, Canada

Abstract

We present a comprehensive study of 136 system failures attributed to network-partitioning faults from 25 widely used distributed systems. We found that the majority of the failures led to catastrophic effects, such as data loss, reappearance of deleted data, broken locks, and system crashes. The majority of the failures can easily manifest once a network partition occurs: They require little to no client input, can be triggered by isolating a single node, and are deterministic. However, the number of test cases that one must consider is extremely large. Fortunately, we identify ordering, timing, and network fault characteristics that significantly simplify testing. Furthermore, we found that a significant number of the failures are due to design flaws in core system mechanisms.

We found that the majority of the failures could have been avoided by design reviews, and could have been discovered by testing with network-partitioning fault injection. We built NEAT, a testing framework that simplifies the coordination of multiple clients and can inject different types of network-partitioning faults. We used NEAT to test seven popular systems and found and reported 32 failures.

1 Introduction

With the increased dependency on cloud systems [1, 2, 3, 4], users expect high—ideally, 24/7—service availability and data durability [5, 6]. Hence, cloud systems are designed to be highly available [7, 8, 9] and to preserve data stored in them despite failures of devices, machines, networks, or even entire data centers [10, 11, 12].

Our goal is to better understand the impact of a specific type of infrastructure fault on modern distributed systems: network-partitioning faults. We aim to understand the specific sequence of events that lead to user-visible system failures and to characterize these system failures to identify opportunities for improving system fault tolerance.

We focus on network partitioning for two reasons. The first is due to the complexity of tolerating these faults [13, 14, 15, 16]. Network-partitioning fault tolerance pervades the design of all system layers, from the communication middleware and data replication [13, 14, 16, 17] to user API definition and semantics [18, 19], and it dictates the availability and consistency levels a system can achieve [20]. Second, recent studies [21, 22, 23, 24] indicate that, in production networks, network-partitioning faults occur as frequently as once a week and take from tens of minutes to hours to repair.

Given that network-partitioning fault tolerance is a well-studied problem [13, 14, 17, 20], this raises questions about *how these faults sill lead to system failures. What is the impact of these failures? What are the characteristics of the sequence of events that lead to a system failure? What are the characteristics of the network-partitioning faults? And, foremost, how can we improve system resilience to these faults?*

To help answer these questions, we conducted a thorough study of 136 network-partitioning failures[1] from 25 widely used distributed systems. The systems we selected are popular and diverse, including key-value systems and databases (MongoDB, VoltDB, Redis, Riak, RethinkDB, HBase, Aerospike, Cassandra, Geode, Infinispan, and Ignite), file systems (HDFS and MooseFS), an object store (Ceph), a coordination service (ZooKeeper), messaging systems (Kafka, ActiveMQ, and RabbitMQ), a data-processing framework (Hadoop MapReduce), a search engine (Elasticsearch), resource managers (Mesos, Chronos, and DKron), and in-memory data structures (Hazelcast, Ignite, and Terracotta).

For each considered failure, we carefully studied the failure report, logs, discussions between users and developers, source code, code patch, and unit tests. We manually reproduced 24 of the failures to understand the specific manifestation sequence of the failure.

Failure impact. Overall, we found that network-partitioning faults lead to *silent catastrophic* failures (e.g., data loss, data corruption, data unavailability, and broken locks), with *21% of the failures leaving the system in a lasting erroneous state* that persists even after the partition heals.

Ease of manifestation. Oddly, it is easy for these failures to occur. *A majority of the failures required three or fewer frequently used events (e.g., read, and write), 88% of them can be triggered by isolating a single node, and 62% of them were deterministic.* It is surprising that catastrophic failures manifest easily, given that these systems are generally developed using good software-engineering practices and are subjected to multiple design and code reviews as well as thorough testing [5, 25].

[1] A *fault* is the initial root cause, including machine and network problems and software bugs. If not properly handled a fault may lead to a user-visible system *failure*.

Partial Network Partitions. Another unexpected result is that a *significant number of the failures (29%) were caused by an unanticipated type of fault: partial network partitions.* Partial partitions isolate a set of nodes from some, but not all, nodes in the cluster, leading to a confusing system state in which the nodes disagree whether a server is up or down. The effects of this disagreement are poorly understood and tested. This is the first study to analyze the impact of this fault on modern systems.

Testability. We studied the testability of these failures. In particular, we analyzed the manifestation sequence of each failure, ordering constraints, timing constraints, and network fault characteristics. While the number of event permutations that can lead to a failure is excessively large, we identified characteristics that significantly reduce the number of test cases (Section 5). We also found that *the majority of the failures can be reproduced through tests and by using only three nodes.*

Our findings debunk two common presumptions. First, network practitioners presume that systems, with their software and data redundancy, are robust enough to tolerate network partitioning [22]. Consequently, practitioners assign low priority to the repair of top-of-the-rack (ToR) switches [22], even though these failures isolate a rack of machines. Our findings show that this presumption is ill founded, as *88% of the failures can occur by isolating a single node.* Second, system designers assume that limiting client access to one side of a network partition will eliminate the possibility of a failure [28, 29, 30, 31, 32, 33, 34]. Our findings indicate that *64% of the failures required no client access at all or client access to only one side of the network partition.*

We examined the unit tests that we could relate to the studied code patches and we found that developers lack the proper tools to test these failures. In most cases, developers used mocking [26, 27] to test the impact of network partitioning on only one component and on just one side of the partition. However, this approach is inadequate for end-to-end testing of complete distributed protocols.

Our findings motivated us to build the network partitioning testing framework (NEAT). NEAT simplifies testing by allowing developers to specify a global order for client operations and by providing a simple API for creating and healing partitions as well as crashing nodes. NEAT uses OpenFlow [35] to manipulate switch-forwarding rules and create partitions. For deployments that do not have an OpenFlow switch, we built a basic version using `iptables` [36] to alter firewall rules at end hosts.

We used NEAT to test seven systems: Ceph [37], ActiveMQ [38], Apache Ignite [39], Terracotta [40], DKron [41], Infinispan [42], and MooseFS [43]. We found and reported 32 failures that led to data loss, stale reads, reappearance of deleted data, unavailability, double locking, and broken locks.

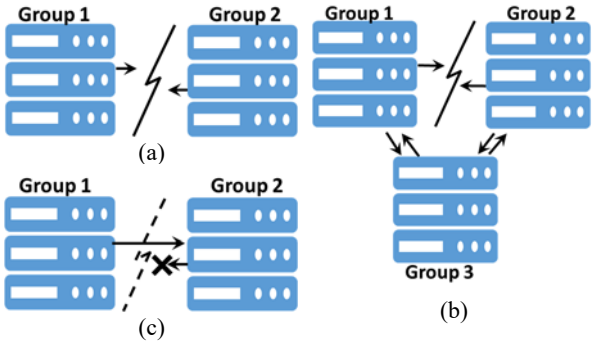

Figure 1. Network partitioning types. (a) Complete partition: The system is split into two disconnected groups (b) Partial partition: The partition affects some, but not all, nodes in the system. Group 3 in Figure (b) can communicate with the other two groups. (c) Simplex partition, in which traffic flows only in one direction.

The rest of this paper is organized as follows: In Section 2, we present a categorization of network-partitioning faults, discuss the theoretical limit on system design, and discuss the current testing techniques. In Section 3, we present our methodology and its limitations. Then, we present our findings in Sections 4 and 5 and discuss a number of related observations in Section 6. We present the NEAT framework in Section 7. We present additional related work in Section 8. We share our insights in Section 9 and conclude our paper in Section 10.

2 Background

In this section, we present the three types of network-partitioning faults (Section 2.1), discuss the theoretical limit for systems design (Section 2.2), and survey the current approaches for testing systems' resilience to network-partitioning faults (Section 2.3).

2.1 Types of Network Partitions

Modern networks are complex. They span multiple data centers [44, 45], use heterogeneous hardware and software [23], and employ a wide range of middle boxes (e.g., NAT, load balancers, route aggregators, and firewalls) [21, 44, 45]. Despite the high redundancy built into modern networks, catastrophic failures are common [21, 22, 23, 24]. We surveyed network-partitioning failures and identified three types:

Complete network partitioning leads to dividing the network into two disconnected parts (Figure 1.a). Complete partitions can happen at different scales; for example, they can manifest in geo-replicated systems due to the loss of connectivity between data centers. HP reported that 11% of its enterprise network failures lead to site connectivity problems [23]. Turner et al. found that a network partition occurs almost once every 4

days in the California-wide CENIC network [24]. In a data center, a complete partition can manifest due to failures in the core or aggregation switches [22] or because of a ToR switch failure. Microsoft and Google report that ToR failures are common and have led to 40 network partitions in two years at Google [21] and caused 70% of the downtime at Microsoft [22]. Finally, NIC failures [46] or bugs in the networking stack can lead to the isolation of a single node that could be hosting multiple VMs. Finally, network-partition faults caused by correlated failures of multiple devices are not uncommon [22, 24, 44]. Correlated switch failures are frequently caused by system-wide upgrades and maintenance tasks [21, 22].

Partial network partitioning is a fault that leads to the division of nodes into three groups (Group1, Group2, and Group3 in Figure 1.b) such that two groups (say, Group1 and Group2) are disconnected while Group3 can communicate with both Group1 and Group2 (Figure 1.b). Partial partitions are caused by a loss of connectivity between two data centers [23] while both are reachable by a third center, or due to inconsistencies in switch-forwarding rules [21].

Simplex network partitioning permits traffic to flow in one direction, but not in the other (Figure 1.c). This is the least common failure and can be caused by inconsistent forwarding rules or hardware failures (e.g., the Broadcom BCM5709 chipset bug [46]). The impact of this failure is mainly manifested in UDP-based protocols. For instance, a simplex network partitioning dropped all incoming packets to a primary server while allowing the primary server heartbeats to reach the failover server. The system hang as the failover server neither detected the failure nor took over [46].

2.2 Theoretical Limit

The data consistency model defines which values a read operation may return. The strong consistency model [47] (a.k.a. sequential consistency) is the easiest to understand and use. Strong consistency promises that a read operation will return the most recent successfully written value. Unfortunately, providing strong consistency reduces system availability and requires complex consistency protocols [13, 14, 17]. Gilbert and Lynch [20] presented a theoretical limit on system design. Their theorem, famously known as the CAP theorem, states that in the presence of a network partition, designers need to choose between keeping the service available and maintaining data consistency.

To maintain system availability, system designers choose a relaxed consistency model such as the read-your-write [11, 18, 19, 48], timeline [19, 48, 49], and eventually consistent [16, 19, 50, 51] models.

Modern systems often implement consensus protocols that have not been theoretically proven. Eventually consistent systems implement unproven protocols (Hazelcast [29] and Redis [32]), and systems that implement proven, strongly consistent protocols (e.g., Paxos [13] and Raft [14]) often tweak these protocols in unproven ways [15, 31, 52]. These practices make modern systems vulnerable to unforeseen failure scenarios, such as the ones caused by different types of network partitions.

2.3 Testing with Network Partitioning

A common testing technique for network-partitioning failures is mocking. Mocking frameworks (e.g., Mockito [26]) can be used to imitate communication problems. Mocking can be employed to test the impact of a failure on a single component, but it is not suitable for system-level testing or for testing distributed protocols. A few systems use hacks to emulate a network partition; for instance, Mesos' unit tests emulate a network partition by ignoring test-specific messages received by the protobuf middleware [53].

Another possible testing approach is to use the Jepsen testing framework [54]. Jepsen is written in Clojure [55] and is tuned toward random testing. Jepsen testing typically involves running an auto-generated testing workload while the tool injects network-partitioning faults. Jepsen does not readily support unit testing or all types of network partitioning.

We built NEAT, a Java-based, system-level testing framework. NEAT has a simple API for deploying systems, specifying clients' workloads, creating and healing partitions, and crashing nodes. Unlike Jepsen, NEAT readily supports injecting the three types of network-partitioning faults.

3 Methodology and Limitations

We studied 136 real-world failures in 25 popular distributed systems. We selected a diverse set of distributed systems (Table 1), including 10 key-value storage systems and databases, a coordination service, two file systems, an object storage system, three message-queueing systems, a data-processing framework, a search engine, three resource managers, and three distributed in-memory caches and data structures. We selected this diverse set of systems to understand the wide impact of network-partitioning faults on distributed systems and because these systems are widely used and are considered production quality.

The 136 failures[2] we studied include 88 failures extracted from the publicly accessible issue-tracking systems, 16 Jepsen reports [54], and 32 failures detected by our NEAT framework (Section 7). The majority of the studied tickets contain enough details to

[2] We differentiate failures by their manifestation sequence of events. In a few cases, the same faulty mechanism leads to two different failures and impacts depending on workload. We count these as separate failures, even if they were reported in a single ticket. Similarly, although the exact failure is sometimes reported in multiple tickets, we count it once in our study.

Table 1. List of studied system. The table shows systems' consistency model, number of failures, and number of catastrophic failures. Highlighted rows indicate systems we tested using NEAT, and the number of failures we found.

System	Consistency Model	Failures	
		Total	Catastrophic
MongoDB [31]	Strong	19	11
VoltDB [33]	Strong	4	4
RethinkDB [52]	Strong	3	3
HBase [56]	Strong	5	3
Riak [57]	Strong/Eventual	1	1
Cassandra [58]	Strong	4	4
Aerospike [59]	Eventual	3	3
Geode [60]	Strong	2	2
Redis [32]	Eventual	3	2
Hazelcast [29]	Best Effort	7	5
Elasticsearch [28]	Eventual	22	21
ZooKeeper [61]	Strong	3	3
HDFS [1]	Custom	4	2
Kafka [30]	-	5	3
RabbitMQ [62]	-	7	4
MapReduce [4]	-	6	2
Chronos [63]	-	2	1
Mesos [64]	-	4	0
Infinispan [42]	Strong	1	1
Ignite [39]	Strong	15	13
Terracotta [40]	Strong	9	9
Ceph [37]	Strong	2	2
MooseFS [43]	Eventual	2	2
ActiveMQ [38]	-	2	2
DKron [41]	-	1	1
Total	-	**136**	**104**

understand the failure. These tickets document failures that were confirmed by the developers and include discussions between the users and the developers, steps to reproduce the failure, outputs and logs, code patch, and sometimes unit tests.

The 88 failures we included in our study were selected as follows: First, we used the search tool in the issue-tracking systems to identify tickets related to network partitioning. We searched using the following keywords: "network partition," "network failure," "switch failure," "isolation," "split-brain," and "correlated failures." Second, we considered tickets that were dated 2011 or later. Third, we excluded low-priority tickets that were marked as "Minor" or "Trivial." Fourth, we examined the set of tickets to verify that they were indeed related to network-partitioning failures and excluded tickets that appeared to be part of the development cycle; for instance, they discuss a feature design. Finally, some failures that are triggered by a node crash can also be triggered by a network partition isolating that node. We excluded failures that can be triggered by a node crash and studied failures that can only be triggered by a network partition. Out of all Jepsen blog posts (there is 25 in total), we included 16 that are related to the systems we studied. Table 1 shows the number of failures and the consistency model of the systems we studied.

For each ticket, we studied the failure description, system logs, developers' and users' comments, code patch, and unit tests. Using NEAT, we also reproduced 13 failures reported in the issue-tracking systems, as well as 11 failures reported by Jepsen to understand their intricate details.

Limitations: As with any characterization study, there is a risk that our findings may not be generalizable. Here we list three potential sources of bias and describe our best efforts to address them.

1) *Representativeness of the selected systems*. Because we only studied 25 systems, the results may not be generalizable to the hundreds of systems we did not study. However, we selected a diverse set of systems (Table 1). These systems follow diverse designs, from persistent storage and reliable in-memory storage to volatile caching systems. They use leader-follower or peer-to-peer architectures; are written in Java, C, Scala, or Erlang; adopt strong or eventual consistency; use synchronous or asynchronous replication; and use chain or parallel replication. The systems we selected are widely used: ZooKeeper is a popular coordination service; Kafka is the most popular message-queueing system; MapReduce, HDFS, and HBase are the core of the dominant Hadoop data analytics platform; MongoDB, Riak, Aerospike, Redis, and VoltDB are popular key-value-based databases; and Hazelcast, Ignite, and Terracotta are popular tools in a growing area of in-memory distributed data structures.
2) *Sampling bias*. The way we choose the tickets may bias the results. We designed our methodology to include high impact tickets. Modern systems take node unreachability as an indicator of a node crash. Consequently, a network partition that isolates a single node can trigger the same failures that are caused by a single node crash. We excluded failures that can be caused by a node crash and considered those that are solely triggered by a network partitioning fault (i.e., the nodes on both sides of the partition must be running for a failure to manifest). Furthermore, we eliminated all low-priority tickets and focused on tickets the developers considered important. All presented findings should be interpreted with this sampling methodology in mind.
3) *Observer error*. To minimize the possibility of observer errors, all failures were independently reviewed by two team members and discussed in a group meeting before agreement was reached, and all team members used the same detailed classification methodology.

4 General Findings

This section presents the general findings from our study. Overall, our study indicates that network partitioning leads to catastrophic failures. However, it identifies failure characteristics that can improve testing. We show that most of the studied failures can be reproduced using only three nodes and are deterministic or have bounded timing constraints. We show that core distributed system mechanisms are the most vulnerable, including leader election, replication, and request routing. Finally, we show that a large number of the failures are caused by partial network-partitioning faults.

4.1 Failure Impact

Overall, our findings indicate that network-partitioning faults cause silent catastrophic failures that can result in lasting damage to systems.

Finding 1. *A large percentage (80%) of the studied failures have a catastrophic impact, with data loss being the most common (27%)* (Table 2).

We classify a failure as catastrophic if it violates the system guarantees or leads to a system crash. Table 2 lists the different types of catastrophic failures. Failures that degrade performance or crash a single node are not considered catastrophic. Stale reads are catastrophic only when the system promises strong consistency. However, they are not considered failures in eventually consistent systems. Dirty reads happen when the system returns the value of a preceding unsuccessful write operation. For instance, a client reading from the primary replica in MongoDB may get a value that is simultaneously being written by a concurrent write operation [65]. If the write fails due to network partitioning, the read operation has returned a value that was never successfully written (a.k.a. dirty read).

Compared to other causes of failures, this finding indicates that network partitioning leads to a significantly higher percentage of catastrophic failures. Yuan et al. [66] present a study of 198 randomly selected, high-priority failures from five of the systems we include in our study: Cassandra, HBase, HDFS, MapReduce, and Redis. They report that only 24% of failures had catastrophic effects[3], compared to 80% in the case of network-partitioning failures (Table 2). Consequently, developers should carefully consider this fault in all phases of system design, development, and testing.

Table 2. The impacts of the failures. The percentage of the failures that cause each impact. Broken locks include double locking, lock corruption, and failure to unlock.

Impact	%	
Data loss	26.6%	Catastrophic (79.5%)
Stale read	13.2%	
Broken locks	8.2%	
System crash/hang	8.1%	
Data unavailability	6.6%	
Reappearance of deleted data	6.6%	
Data corruption	5.1%	
Dirty read	5.1%	
Performance degradation	19.1%	
Other	1.4%	

Finding 2. *The majority (90%) of the failures are silent, whereas the rest produce warnings that are unatonable.*

We inspected the failure reports for returned error messages and warnings. The majority of the failures were silent (i.e., no error or warning was returned to the client), with some failures (10%) returning warning messages to the client. Unfortunately, all returned warnings were confusing, with no clear mechanism for resolution. For instance, in Riak [67] with a strict quorum configuration, when a write fails to fully replicate a new value, the client gets a warning indicating that the write operation has updated a subset of replicas, but not all of them. This warning is confusing because it does not indicate the necessary action to take next. Similarly, MongoDB returns a generic socket exception if a proxy node cannot reach the data nodes [68].

This is alarming because users and administrators are not notified when a failure occurs, which delays failure discovery, if the failure is discovered at all.

Finding 3. *Twenty one percent of the failures lead to permanent damage to the system. This damage persists even after the network partition heals.*

While 79% of the failures affect the system only while there is a network partition, 21% of the failures leave the system in an erroneous state that persists even after the network partition heals. For instance, if a new node is unable to reach the other nodes in RabbitMQ [69] and Ignite (section 7.4), the node will assume that the rest of the cluster has failed and will form a new independent cluster. These clusters will remain separated, even after the network partition heals.

Overall, as recent studies [21, 22, 23, 24] indicate that network-partitioning faults occur as frequently as once a week and take from tens of minutes to hours to repair, it is alarming that these faults can lead to silent catastrophic failures. This is surprising, given that these systems are designed for deployments in which component failure is the norm. For instance, all of the systems we studied replicate their data. In MongoDB, Hazelcast, Kafka, Elasticsearch, Geode, Mesos, Redis,

[3] We note that these percentages are not directly comparable as our definition of catastrophic failure is more conservative. For instance, while Yuan et al. [66] count a loss of a single replica or a crash of a single node as catastrophic, we do not.

VoltDB, and RethinkDB, if a leader node is partitioned apart from the majority, then the rest of the nodes will quickly elect a new leader. Hazelcast and VoltDB employ "split-brain protection," a technique that continuously monitors the network and pauses nodes in the minority partition if a network partition is detected. Furthermore, ZooKeeper and MongoDB include a mechanism for data consolidation. How, then, do these failures still occur?

4.2 Vulnerability of System Mechanisms

Finding 4. *Leader election, configuration change, request routing, and data consolidation are the most vulnerable mechanisms to network partitioning* (Table 3).

Leader election is the most vulnerable to network partitioning (was affected by 40% of the failures). We further analyzed leader election failures (Table 4) and found that the most common leader election flaw is the simultaneous presence of two leaders. This failure typically manifests as follows: A network partition isolates the current leader from the majority of replicas. The majority partition elects a new leader. The old leader may eventually detect that it no longer has a majority of replicas at its side and step down. However, there is a period of time in which each network partition has a leader. The overlap between the two leaders may last until the network partition heals (which may take hours [21]). In MongoDB [70], VoltDB [71], and Raft-based RethinkDB [72], if a network partition isolates a leader, the isolated leader will not be able to update the data, but it will still respond to read requests from its local copy, leading to stale and dirty reads.

In all of the systems we studied, the leader trusts that its data set or log is complete and all replicas should update/trim their data sets to match the leader copy. Consequently, it is critical to elect the leader with a complete and consistent data set. Table 4 shows that 20% of leader election failures are caused by electing a bad leader. This is caused by using simple criteria for leader election, such as the node with the longest log wins (e.g., VoltDB), the node that has the latest operation timestamp wins (e.g., MongoDB), or the node with the lowest id wins (e.g., Elasticsearch). These criteria can cause data loss when a node from the minority partition becomes a leader and erases all updates performed by the majority partition.

Conflicting election criteria lead to 3.7% of the leader election failures and are only reported in MongoDB. MongoDB leader election has multiple criteria for electing a leader. One can assign a priority for a replica to become a leader. The priority node will reject any leader proposal; similarly, the node with the latest operation timestamp will reject all leader proposals, leaving the cluster without a leader [73].

Table 3. The percentage of the failures involving each system mechanism. Some failures involve multiple mechanisms.

Mechanism		%
Leader election		39.7%
Configuration change		19.9%
▪ Adding a node	10.3%	
▪ Removing a node	3.7%	
▪ Membership management	3.7%	
▪ Other	2.2%	
Data consolidation		14.0%
Request routing		13.2%
Replication protocol		12.5%
Reconfiguration due to a network partition		11.8%
Scheduling		2.9%
Data migration		3.7%
System integration		1.5%

Table 4. Leader election flaws.

Leader election failure	%
Overlapping between successive leaders	57.4%
Electing bad leaders	20.4%
Voting for two candidates	18.5%
Conflicting election criteria	3.7%

The second most affected mechanism is configuration change, including node join or leave and role changes (e.g., changing the primary replica). We discuss two examples of these failures in Section 4.4.

The third most affected mechanism is data consolidation. Failures in this mechanism typically lead to data loss in both eventually and strongly consistent systems. For instance, Redis, MongoDB, Aerospike, Elasticsearch, and Hazelcast employ simple policies to automate data consolidation, such as the write with the latest timestamp wins and the log with the most entries wins. However, because these policies do not check the replication or operation status, they can lose data that is replicated on the majority of nodes and that was acknowledged to the client.

The three ZooKeeper failures that we studied are related to data consolidation. For instance, ZooKeeper has two mechanisms for synchronizing data between nodes: storage synchronization that is used for syncing a large amount of data, and in-memory log synchronization that is used for a small amount of data. If node A misses many updates during a network partition, then ZooKeeper will use storage synchronization to bring node A up to date. Unfortunately, storage synchronization does not update the in-memory log. If A becomes a leader, and other nodes use in-memory log synchronization, then A will replicate its incomplete in-memory log [74].

Request routing represents the mechanism for routing requests or responses between clients and the specific nodes that can serve the request. Failures in request routing represent 13.2% of the failures. The

majority of those failures are caused by failing to return a response. For instance, in Elasticsearch, if a replica (other than the primary) receives write requests, it acts as a coordinator and forwards the requests to the primary replica. If a primary completes the write operation but fails to send an acknowledgment back to the coordinator, then the coordinator will assume the operation has failed and will return an error code to the client. The next client read will return the value written by a write operation that was reported to have failed. Moreover, if the client repeats the operation, then it will be executed twice [75].

The rest of the failures were caused by flaws in the replication protocol, scheduling, data migration mechanism, system integration with ZooKeeper, and system reconfiguration in response to network partitioning failures, in which the nodes remove the unreachable nodes from their replica set.

These findings are surprising because 15 of the systems use majority voting for leader election to tolerate exactly this kind of failure. Similarly, the primary purpose of a data consolidation mechanism is to correctly resolve conflicting versions of data. To improve resilience, this finding indicates that developers should enforce tests and design reviews focusing on network-partitioning fault tolerance, especially on these mechanisms.

4.3 Network Faults Analysis

Finding 5. *The majority (64%) of the failures either do not require any client access or require client access to only one side of the network partition* (Table 5).

This finding debunks a common presumption that network partitioning mainly leads to data conflicts, due to concurrent writes at both sides of the partition. Consequently, developers ensure that clients can only access one side of the partition to eliminate the possibility of a failure [28, 29, 30, 31, 32, 33, 34]. As an example of a failure that requires client access to one side of the partition, in HBase, region servers process client requests and store them in a log in HDFS. When the log reaches a certain size, a new log is created. If a partial partition separates a region server from the HMaster but not from HDFS, then the HMaster will assume that the region server has crashed and will assign the region logs to other servers. At this time, if the old region server creates a new log, HMaster will not be aware of the new log and will not assign it to any region server. All client operations stored in the new log will be lost [76]. We discuss a MapReduce failure that does not require any client access in section 4.4.

This finding indicates that system designers must consider the impact of a network partition fault on all system operations, including asynchronous client operations and offline internal operations.

Table 5. Percentage of the failures that require client access *during* the network partition

Client Access	%
No client access necessary	28%
Client access to one side only	36%
Client access to both sides	36%

Table 6. Percentage of the failures caused by each type of network-partitioning fault.

Partition type	%
Complete partition	69.1%
Partial partition	28.7%
Simplex partition	2.2%

Finding 6. *While the majority (69%) of the failures require a complete partition, a significant percentage of them (29%) are caused by partial partitions* (Table 6).

Partial network partitioning failures are poorly understood and tested, even by expert developers. For instance, most of the network-partitioning failures in Hadoop MapReduce and HDFS are caused by partial network-partitioning faults. In the following section, we discuss these failures in detail.

Simplex network partitioning caused 2% of the failures. This type of fault only confuses UDP-based protocols and leads to performance degradation. For instance, in HDFS [77], a data node that can send a periodic heartbeat message but is unable to receive requests is still considered a healthy node.

The overwhelming majority (99%) of the failures were caused by a single network partition. Only 1% of the failures required two network partitions to manifest.

4.4 Partial Network-Partitioning Failures

To the best of our knowledge, this the first study to analyze and highlight the impact of partial network partitions on systems. Consequently, we dedicate this section to discussing our insights and presenting detailed examples of how these failures manifest.

We found that the majority of partial network-partitioning failures are due to design flaws. This indicates that developers do not anticipate networks to fail in this way. Other than that, partial partitions failures had impact, ordering, and timing characteristics that are similar to complete partition failures.

Tolerating partial network partitions is complicated because these faults lead to inconsistent views of a system state; for instance, nodes disagree on whether a server is up or down. This confusion leads part of the system to carry on normal operations, while another part executes fault tolerance routines. Apparently, the mix of these two modes is poorly tested. The following are four examples:

- *Scheduling in MapReduce and Elasticsearch.* In MapReduce, if a partial network partition isolates an AppMaster from the resource manager while both

can still communicate with the cluster nodes, the AppMaster will finish executing the current task and return the result to the client. The resource manager will assume that the AppMaster has failed and will rerun the task using a new AppMaster. The new AppMaster will execute the task again and send a second result to the client. This failure will confuse the client and will lead to data corruption and double execution [78]. Note that in this failure, there is no client access after the network partition.

Elasticsearch has a similar failure [75]—if a coordinator does not get the result from a primary node, the coordinator will run the task again, leading to double execution.

- *Data placement in HDFS.* If a partial network partition separates a client from, say, rack 0, while the NameNode can reach that rack. If the NameNode allocates replicas on rack 0, then a client write operation will fail, and the client will ask for a different replica. The NameNode, following its rack-aware data placement, will likely suggest another node from the same rack. The process repeats five times before the client gives up [79].
- *Leader election in MongoDB and Elasticsearch.* MongoDB design includes an arbiter process that participates in a leader election to break ties. Assume a MongoDB cluster with two replicas (say A and B) and an arbiter, with A being the current leader. Assume a partial network partition separates A and B, while the arbiter can reach both nodes. B will detect that A is unreachable and will start a leader election process; being the only contestant, it will win the leadership. The arbiter will inform A to step down. After missing three heartbeats from the current leader (i.e., B), A will assume that B has crashed, start the leader election process, and become a leader. The arbiter will inform B to step down. This thrashing will continue until the network partition heals [80]. MongoDB does not serve client requests during leader election; consequently, this failure significantly reduces availability.

  Elasticsearch has a similar failure [81], in which a partial partition leads to having two simultaneous leaders because nodes that can reach the two partitions become followers of the two leaders. Note that these failures do not require any client access.
- *Configuration change in RethinkDB and Hazelcast.* RethinkDB is a strongly consistent database based on Raft [52]. Unlike Raft, when an admin removes a replica from RethinkDB cluster, the removed replica will delete its Raft log. This apparently minor tweak of the Raft protocol leads to a catastrophic failure. For instance, if a partial network partition breaks a replica set of five servers (A, B, C, D, and E) such that the (A, B) partition cannot reach (D, E) while C can reach all nodes, then if D receives a request to change the replication to two, D will remove A, B, and C from the set of replicas. C will delete its log. A and B will be unaware of the configuration change and still think that C is an active replica. C, having lost its Raft log that contains the configuration change request, will respond to A and B requests. This scenario creates two replica sets for the same keys. D and E are a majority in the new configuration, and A, B, and C are a majority in the old configuration [72].

  Hazelcast has a similar failure [82]. In Hazelcast, nodes delete their local data on configuration change. If a partial partition separates the new primary replica, then one replica will promote itself to become the primary. If the central master can reach both partitions, it will see that the old primary is still alive and inform the self-promoted replica to step down. That replica will step down, delete its data, and try to download the data from the primary. If the primary permanently fails before the partition heals, the data will be lost [82].

5 Failure Complexity

To understand the complexity of these failures, we studied their manifestation sequence, importance of input events order, network fault characteristics, timing constraints, and system scale. The majority of the failures are deterministic, require three or fewer input events, and can be reproduced using only three nodes. These characteristics indicate that it is feasible to test for these failures using limited resource.

5.1 Manifestation Sequence Analysis

Finding 7. *A majority (83%) of the failures triggered by a network partition require an additional three or fewer input events to manifest* (Table 7).

Table 8 lists the events that led to failures. All of the listed operations are frequently used. Read and write operations are part of over 50% of the failures, and 12.6% of the failures do not require any events other than a single network-partitioning fault. As an example of a failure without any client access, in Redis [83], if a network partition separates two nodes during a sync operation, the data log on the receiving node will be permanently corrupted. Similarly, in RabbitMQ [84], if a partial partition isolates one node from the leader, but not from the rest of the replicas, that node will assume the leader has crashed. The isolated node will become the new leader. When the old leader receives a notification to become a follower, it will start a follower thread but will not stop the leader thread. The contention between the follower and leader threads results in a complete system hang.

This is perilous, as a small number of frequently used events can lead to catastrophic failures.

Table 7. The minimum number of events required to cause a failure. The table counts a network-partitioning fault as an event. Note that 12.5% of the failures require no client access, neither during a network partition nor after it heals. Note that 28% of the failures reported in Table 5 do not require client access *during* the partition, but around 15.5% require client access before or after the network partition occurs.

Number of events	%
1 (just a network partition)	12.6%
2	13.9%
3	42.6%
4	14.0%
> 4	16.9%

Table 8. Percentage of faults each event is involved in.

Event type	%
Only a network-partitioning fault	12.6%
Write request	48.5%
Read request	34.6%
Acquire lock	8.1%
Admin adding/removing a node	8.0%
Delete request	4.4%
Release lock	3.7%
Whole cluster reboot	1.5%

Finding 8. *All of the failures that involve multiple events only manifest if the events happen in a specific order.*

All of the 87% of failures that require multiple events (2 events or more in Table 7) need the events to occur in a specific order. This implies that to expose these failures we not only need to explore the combination of these events, but also the different permutations of events, which makes the event space extremely large.

Fortunately, we identified characteristics that significantly prune this large event space and make testing tractable (Table 9). First, 84% of the manifestation sequences start with a network-partitioning fault. For 27.7% of the sequences, the order of the rest of events is not important, and in 27% of the sequences the events follow a natural order; that is, lock() comes before unlock(), and write() before read().

While this finding indicates that reproducing a failure can be complex, the probability of a failure in production is still high. The majority of multi-event failures require three or fewer events (Table 7); consequently, it is highly likely for a system that experiences a network partitioning for hours to receive all possible permutations of these common events.

Table 9. Ordering charactrisitcs.

Ordering Charactrisitcs		%
Network partition does not come first		16.0%
Network partition comes first		84.0%
▪ Order is not important	27.7%	
▪ Natural order	26.9%	
▪ Other	29.4%	

Table 10. System connectivity during the network partition. Examples of a central service include a ZooKeeper cluster and HBase master. Examples of nodes with a special role include MongoDB arbiter and MapReduce AppMaster.

Network Partition Characteristics		%
Partition any replica		44.9%
Partition a specific node		55.1%
▪ Partition the leader	36.0%	
▪ Partition a central service	8.8%	
▪ Partition a node with a special role	3.7%	
▪ Other (e.g., new node, source of data migration)	6.6%	

Finding 9. *The majority (88%) of the failures manifest by isolating a single node, with 45% of the failures manifest by isolating any replica*.

It is alarming that the majority of the failures can occur by isolating a single node. Conceivably, isolating a single node is more likely than other network-partitioning cases; it can happen because of an NIC failure, a single link failure, or a ToR switch failure. ToR switch failures are common in production networks leading to 40 network partitions in two years at Google [21] and 70% of the downtime at Microsoft [22]. This finding invalidates the common practice of assigning a low priority to ToR switch failures based on the presumption that data redundancy can effectively mask them [22]. Our results show that this practice aggravates the problem by prolonging the partition.

We further studied the connectivity between replicas (Table 10) of the same object and found that 45% of failures manifest by isolating any replica, and the rest requires isolating a specific node or service (e.g., ZooKeeper cluster). Among the failures that isolate a specific node, isolating a leader replica (36%) and central services (8.8%) are the most common. This does not reduce the possibility of a failure because, as in many systems, every node is a leader for some data and is a secondary replica for other data. Consequently, isolating any replica in the cluster will most likely isolate a leader.

This finding highlights the importance of testing these specific faults that isolate a leader, a central service, and nodes with special roles (e.g., scheduler, and MapReduce App Master).

5.2 Timing Constraints

Finding 10. The majority (*80%) of the failures are either deterministic or have known timing constraints*.

The majority of the failures (Table 11) are either deterministic (62%), meaning they will manifest given the input events, or have known timing constraints (18%). These known constraints are configurable or hard coded, such as the number of heartbeat periods to wait before declaring that a node has failed.

Table 11. Timing constraints.

Timing constraints		%
No timing constraints		61.8%
Has timing constraints		31.2%
▪ Known	18.4%	
▪ Unknown – but still can be tested	12.8%	
Nondeterministic		7%

Furthermore, we found that the timing constraints immediately follow network-partitioning faults. For instance, if a partition isolates a leader, for a failure to happen, events at the old leader side should be invoked right after the partition, so they are processed before the old leader steps down; while on the majority side, the test should sleep for a known period until a new leader is elected. For instance, in RabbitMQ, Redis, Hazelcast, and VoltDB, a failure will happen only if a write is issued before the old leader steps down (e.g., within three heartbeats) after a partitioning fault.

The 13% of the failures that have unknown timing constraints manifest when the sequence of events overlaps with a system internal operation. For instance, in Cassandra, a failure [85] will only occur if a network partition takes place during a data sync operation between the handoff node and a replica. However, these failures can still be tested by well-designed unit tests. For instance, to test the aforementioned Cassandra failure, a test should (1) isolate a replica to make the system add a handoff node. (2) Write a large amount of data. (3) Heal the partition. Now, the handoff node will start syncing the data with the replica. Finally, (4) create a network partition that isolates the replica during the sync operation and triggers the failure.

Only 7% of the failures are nondeterministic; these failures are caused by multithreaded interleavings and by overlapping the manifestation sequence with hard-to-predict internal system operations.

This finding implies that testers should pay close attention to timing. However, we identified that timing constraints usually follow the partitioning fault, which significantly simplifies testing.

5.3 Resolution Analysis

Finding 11. *The resolution of 47% of the failures required redesigning a system mechanism* (Table 12).

We consider a code patch to be fixing a design flaw if it involves significant changes to the affected mechanism logic, design, or protocol, such as implementing a new leader election protocol in MongoDB and changing configuration change protocols in Elasticsearch.

Table 12. Percentage of design and implementation flaws for failures reported in issue-tracking systems.

Category	%	Average Resolution Time
Design	46.6%	205 days
Implementation	32.2%	81 days
Unresolved	21.2%	-

The large percentage of the failures that led to changes in the mechanism design indicates that network-partitioning faults were not considered in the initial design phase. We expect that a design review focusing on network partitioning fault tolerance would have discovered systems vulnerability to these faults.

Table 12 also reports the resolution time, which is the period from the time a developer acknowledges a failure to the time the issue is fixed. Obtaining an accurate resolution time is tricky. We removed outliers that take minutes to commit a complex patch or take over two years to add a simple patch. In addition, it is not necessary for the time reported to be spent actively solving the issue. Nevertheless, because these are high-priority tickets, we think that the reported times give some indication of the resolution effort. Table 12 shows that design flaws take 2.5 times longer to resolve than implementation bugs.

We noticed that some systems opted to change the system specification instead of fixing the issue. For instance, Redis documentation states that "there is always a window of time when it is possible to lose writes during partitions" [86]. RabbitMQ's documentation was updated to indicate that locking does not tolerate network partitioning [87], and Hazelcast's documentation [88] states that it provides "best effort consistency," in which data updated through atomic operations may be lost. This could imply that some of the systems unnecessarily selected a strong consistency model where an eventual model was sufficient or the developers do not believe that these are high priority issues.

5.4 Opportunity for Improved Testing

Finding 12. *All failures can be reproduced on a cluster of five nodes, with the majority (83%) of the failures being reproducible with three nodes only* (Table 13).

This finding implies that it is not necessary to have a large cluster to test these systems. In fact, it is enough to test them using a single physical machine that runs five virtual machines.

Finding 13. *The majority of the failures (93%) can be reproduced through tests by using a fault injection framework such as NEAT.*

Considering our findings, perhaps it is not surprising that the majority of the failures can be reproduced using unit and system-level tests with a framework that can inject network-partitioning faults. The majority of the failures result from a single network-partitioning fault, need fewer than three common input events, and are

Table 13: Number of nodes needed to reproduce a failure.

Number of Nodes	%
3 nodes	83.1%
5 nodes	16.9%

deterministic or have bounded timing constraints. The 7% that cannot be easily tested are nondeterministic failures or have short vulnerability intervals.

6 Discussion

In this section, we address two additional observations:

- *Overlooking network-partitioning faults.* We found in many cases that the system designer did not consider the possibility of network partitioning. For example, Redis promises data reliability even though it uses asynchronous replication, leading to data loss [89]. Similarly, the Hazelcast locking service relies on asynchronous replication, leading to double locking [90]. Earlier versions of Aerospike assumed that the network is reliable [91].

  We found implicit assumptions made in the studied systems that are untrue. For instance, tickets from MapReduce, RabbitMQ, Ignite, and HBase indicate that the developer assumed an unreachable node to have halted, which is not true with network partitioning. Finally, all partial network-partitioning failures are caused by an implicit assumption that if a node can reach a service, then all nodes can reach that service, which is not always true.
- *Lack of adequate testing tools*. In general, we found that systems lack rigorous testing for network-partitioning. For unit tests related to the code patches we studied, the developers typically used mocking techniques to test the impact of network partitioning on one component on one side of the partition. This makes us believe that the community lacks a network-partitioning fault injection tool that can be integrated with the current testing frameworks.

7 NEAT Framework

We built the *ne*twork p*a*rtitioning *t*esting framework (NEAT), a testing framework with network-partitioning fault injection. NEAT supports the three types of partitions, has a simple API for creating and healing partitions, and simplifies the coordination of events across clients. NEAT is implemented in 1553 lines of Java and uses OpenFlow and the `iptables` tool to inject network-partitioning faults.

7.1 API

NEAT is a generic testing framework. It does not have any constraints on the target system. To test a system, the developer should implement three classes. First is the ISystem interface, which provides methods to install, start, obtain the status of, and shut down the target system. Second, is a Client class that provides wrappers around the client API (e.g., put or get calls). Third is the test workload and verification code.

Listing 1 presents a test for an Elasticsearch data loss failure [92] with partial network partitioning. The network partition (line 7) isolates s1 (the primary replica) and client 1 from s2 and client 2. However, all nodes can reach s3. s2 will detect that the primary replica (s1) is unreachable and start a leader election process. s3 will vote for s2, although it can reach s1, resulting in two leaders. Consequently, writes on both sides of the partition will succeed (line 11 and 12). After healing the partition (line 13), s2 will detect that s1 is reachable. As in Elasticsearch, the replica with a smaller ID wins the election, so s2 will step down and become a follower of s1. s2 will replicate s1's data and, consequently, all writes served by s2 during the partition will be lost and the check on line 16 will fail.

Listing 1. An Elasticsearch test for data loss. The system has three servers: s1 (primary node), s2, and s3, and two clients.

```
1  public static void testDataLoss(){
2   List<Node> side1 = asList(s1, client1);
3   // other servers and clients in one group
4   List<Node> side2 = asList(s2, client2);
5   // create a partial partition. s3 can reach
6   // all nodes
7   Partition netPart = Partitioner.partial(
8                              side1, side2);
9   sleep(SLEEP_LEADER_ELECTION_PERIOD);
10  // write to both sides of the partition
11  assertTrue(client1.write(obj1, v1));
12  assertTrue(client2.write(obj2, v2));
13  Partitioner.heal(netPart);
14  // verify the two objects
15  assertEquals(client2.read(obj1), v1);
16  assertEquals(client2.read(obj2), v2);    }
```

Listing 2. An ActiveMQ test for a double dequeue failure. The system has three servers and two clients.

```
1  public static void testDoubleDequeueu(){
2   assertTrue(client1.send(q1, msg1));
3   assertTrue(client1.send(q1, msg2));
4   // get the master node
5   Node master = AMQSys.getMaster(q1);
6   List<Node> minority= asList(master, client1);
7   List<Node>majority=Partitioner.rest(minority);
8   Partition netPart = Partitioner.complete(
9                              minority, majority);
10  // dequeue at both sides of the partition
11  Msg minMsg = client1.receive(q1);
12  sleep(SLEEP_PERIOD);
13  Msg majMsg = client2.receive(q1);
 assertNotEqual(minMsg, majMsg);     }
```

Listing 2 presents an ActiveMQ test for double dequeueing with complete network partitioning. The network partition (line 8) isolates the master and client1 from the rest of the cluster. The test then pops the queue at both sides of the partition (lines 11-13). If the two sides obtain the same value, then the value was dequeued twice and the test fails.

7.2 Creating and Healing Network Partitions

To create or heal a network partition, the developer calls one of the following methods.

- `Partition complete(List<Node> groupA, List<Node> groupB)`: creates a complete partition between groupA and groupB.

- `Partition partial(List<Node> groupA, List<Node> groupB)`: creates a partition between groupA and groupB without effecting their communication with the rest of the cluster.
- `Partition simplex(List<Node> groupSrc, List<Node> groupDst)`: creates a simplex partition such that packets can only flow from groupSrc to groupDst, but not in the other direction.
- `void heal(Partition p)`: heals partition p.

7.3 NEAT Design

NEAT has three components (Figure 2): server nodes, which run the target system; client nodes, which issue client requests; and a test engine. The test engine is a central node that runs the test workload (e.g., Listing 1).

The test engine simplifies testing by providing a global order for all client operations. The test engine invokes all client operations using Java RMI. The current NEAT prototype has two implementations of the network partitioner module: using OpenFlow and using the `iptables` tool. Furthermore, the test engine provides an API for crashing any group of nodes.

The OpenFlow-based partitioner is a network controller [35] that first installs the rules for a basic learning switch [93]. Then it installs partitioning rules to drop packets from a specific set of source IP addresses to a specific set of destination addresses. The partitioning rules are installed at a higher priority than the learning switch rules. The partitioner is implemented in 152 lines of code using Floodlight [94].

Our choice to use SDN to build a testing framework for distributed systems is research based. Connecting the nodes to a single switch and having the ability to monitor and control every packet in the system is a powerful capability for distributed systems testing. Our first attempt to explore this capability is to build a network partitioner for NEAT. Our current research effort explores techniques to collect detailed system traces under different failure scenarios and build tools to verify and visualize system protocols. This will help developers test, debug, and inspect protocols under different failure scenarios.

For deployments that do not have an OpenFlow switch, we implemented a partitioner by using the `iptables` tool to modify the firewall configuration on every node to create the specified partitions.

7.4 Testing Systems with NEAT

We used NEAT to test seven systems: Ceph [37] (v12.2.5), an object storage system; Apache Ignite [39] (v2.4.0), a key-value store and distributed data structures; Terracotta [40] (v4.3.4), a suite of distributed data structures; DKron [41] (v0.9.8), a job scheduling system; ActiveMQ [38] (v5.15.3), a message-queueing system; Infinispan [42] (v9.2.1), a key-value store; and MooseFS [43] (v3), a file system. All systems were

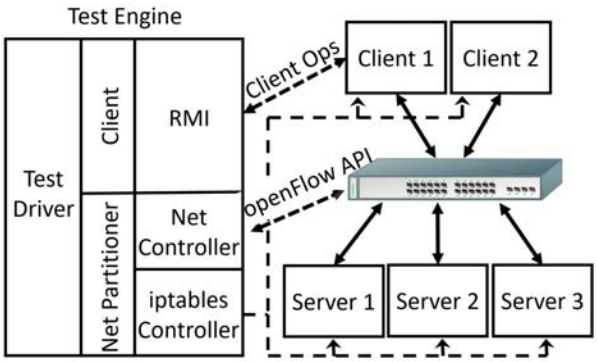

Figure 2. NEAT architecture.

configured with the most reliable configuration. For instance, when possible we persist data on disk, use synchronous replication, and set the minimum replication per operation to equal the majority or the number of all replicas.

Testing setup. We used two testbeds to run our experiments: CloudLab [95] and our own cluster. We used six nodes in our tests. The nodes were connected by a single switch. One node ran the test engine, three nodes ran the system, and two nodes acted as clients.

Our tests involved creating complete and partial partitions, then issuing simple client requests to the two sides of the partition, followed by performing a verification step. On average, tests are implemented in 30 lines of Java code.

The highlighted entries in Table 1 summarize our testing results. Our testing revealed 32 network-partitioning failures, out of which 30 are catastrophic. The failures we found lead to data loss, stale reads, data unavailability, double locking, and lock corruption. It is plausible that a single design flaw or implementation bug (e.g., flawed replication protocol) may cause failures in different operations (e.g., adding to a list and pushing to a queue). We count these as separate failures.

To demonstrate the versatility of NEAT, the following discusses failures that NEAT discovered.

Examples of complete network partition failures: We found that all Ignite atomic synchronization primitives, including semaphores, compare_and_set, increment_and_get, and decrement_and_get, are violated or corrupted when a complete network partition isolates one of the replicas. The main culprit of such failures is the assumption that an unreachable node has crashed; consequently, nodes on both sides of a partition remove the nodes they cannot reach (i.e., the nodes on the other side of the partition) from their replica set and continue to use the semaphore, which may lead to over counting the semaphore. Furthermore, an unreachable client that is holding a semaphore is assumed to have crashed. In this case, the system will reclaim the client's semaphore. If the partition heals

and the client signals the semaphore, the semaphore will be corrupted. These failures lead to lasting damage that persists after the partition heals.

Examples of partial network partition failures: ActiveMQ uses ZooKeeper to keep track of the current leader. If a partial network partition isolates the leader from the replicas, but not from ZooKeeper, the system will hang. The leader will not be able to forward messages to replicas and the replicas will not elect a new leader as ZooKeeper does not see the failure.

In DKron, if a partial partition separates the leader from the rest of DKron's nodes—but not from the central data store service—then the client requests processed by the leader will be successfully executed at the local level. However, DKron will indicate that the task has failed.

8 Additional Related Work

To the best of our knowledge, this is the first in-depth study of the manifestation sequence of network-partitioning failures. The manual analysis allowed us to examine the sequence of events in detail, identify common vulnerabilities, and find failure characteristics that can improve testing.

A large body of previous work analyzed failures in distributed systems. A subset of these efforts focused on specific component failures such as physical [96] and virtual machines [97], network devices [22, 24], storage systems [98, 99], software bugs [100], and job failures [101, 102, 103]. Another set characterized a broader set of failures, but only for specific domain of systems and services, such as HPC [104, 105, 106], IaaS clouds [107], data-mining services [108], hosting services [6, 109], and data-intensive systems [101, 100, 110]. Our work complements these efforts by focusing on failures triggered by network partitioning.

Yuan et al. [66] studied 198 randomly selected failures from six data analytics systems. Comparing our results, we find that a higher percentage of network-partitioning failures (80%) lead to catastrophic effects, compared to 24% reported by Yuan et al. [66]; and while 26% of general failures are nondeterministic, only 7% of network-partitioning failures are non-deterministic. These findings indicate that network-partitioning failures are more critical than general system failures, and testers need to pay close attention to timing.

Jepsen's blog posts report network-partitioning failures that were found using the Jepsen tool [54]. However, they do not detail the manifestation sequences, correlate failures across systems, study the impact of different types of network-partitioning faults, study client access requirements, characterize network faults, or analyze timing constraints.

Majumdar et al. [111] theoretically analyzed the space for faulty executions in the presence of complete network partitioning faults. They discussed the extreme size of the test space and the effectiveness of random testing if tests isolate a specific node, place a leader in a minority, and test with a random order of short sequences of operations.

While we identify characteristics to improve testing, our findings can inform other fault tolerance techniques. Previous efforts explored model checking [112, 113, 114, 115, 116], systematic fault injection [117, 118], and runtime verification techniques [119, 120] for improving systems' fault tolerance. Our findings inform these techniques to consider all types of network partitions and discovered characteristics that can improve these techniques' time and efficiency.

9 Insights

We conducted a comprehensive study of network-partitioning failures in modern cloud systems. It is surprising that these production systems experience silent catastrophic failures due to a frequently occurring infrastructure fault, when a single node is isolated, and under simple and common workloads. Our analysis identified that improvements to the software development process and testing can significantly improve systems' resilience to network partitions. These findings indicate that this is a high-impact research area that needs further effort to improve system design, engineering, testing, and fault tolerance. Our initial results with NEAT are encouraging; even our preliminary testing tool found bugs in production systems, indicating that there is a significant room for improvement.

Another interesting area of research that our analysis identified is partial network partitions fault tolerance. It is surprising that a large number of failures in production systems are triggered by this network fault, yet we could not find any discussion, failure model, or fault tolerance techniques that address this type of infrastructure fault.

Modern systems use unreachability as an indicator for node failure. Our analysis shows the dangers of this approach, as complete network partitions can isolate healthy nodes that lead to both sides assuming that the other side has crashed. Worse yet, partial partitions lead to a confusing state in which some nodes declare part of the system down while the rest of the nodes do not. Further, research is needed for building more accurate node-failure detectors and fault tolerance techniques.

While we identify better testing as one approach for improving system fault tolerance, we highlighted that the number of test cases one needs to consider is excessive. Luckily, our analysis found operations,

timing, ordering, and network failure characteristics that limit the testing space.

Our analysis highlights that the current network maintenance practice of assigning a low priority to ToR switch failure is ill founded and aggravates the problem. Finally, we highlight that system designers need to pay careful attention to internal and offline operations, need be wary of tweaking established protocols, and need to consider network partitioning failures early in their design process.

10 Conclusion and Future Work

We conducted an in-depth study of 136 failure reports from 25 widely used systems for failures triggered by network-partitioning faults. We present 13 main findings that can inform system designers, developers, testers, and administrators; and highlight the need for further research in network partitioning fault tolerance in general and with partial partitions in particular.

We built NEAT, a testing framework that can inject different types of network-partitioning faults. Our testing of seven systems revealed 32 failures.

In our current work, we are focusing on two directions: Extending NEAT to automate testing through workload and network fault generators and exploring fault tolerance techniques for partial network partitioning faults. Our data set and the source code are available at: https://dsl.uwaterloo.ca/projects/neat/

Acknowledgment

We thank the anonymous reviewers and our shepherd, Marcos Aguilera, for their insightful feedback. We thank Matei Ripeanu, Remzi Arpaci-Dusseau, Abdullah Gharaibeh, Ken Salem, Tim Brecht, and Bernard Wong for their insightful feedback on early versions of this paper. We thank Nicholas Lee, Alex Liu, Dian Tang, Anusan Sivakumaran, and Charles Wu for their help in reproducing some of the failures. This research was supported by an NSERC Discovery grant, NSERC Engage grant, Canada Foundation for Innovation (CFI) grant, and in-kind support from Google Canada.

References

[1] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System," in *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: cluster computing with working sets," in *USENIX conference on Hot topics in cloud computing (HotCloud)*, 2010.

[3] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long and C. Maltzahn, "Ceph: a scalable, high-performance distributed file system," in *Symposium on operating systems design and implementation (OSDI)*, 2006.

[4] "Apache Hadoop," [Online]. Available: https://hadoop.apache.org/. [Accessed May 2018].

[5] E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing,* vol. 5, no. 4, pp. 46-55, 2001 .

[6] D. Oppenheimer, A. Ganapathi and D. A. Patterson, "Why do internet services fail, and what can be done about it?," in *Conference on USENIX Symposium on Internet Technologies and Systems (USITS)*, Seattle, WA, 2003.

[7] "Apache Hadoop 2.9.0 – HDFS High Availability," [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html. [Accessed 20 April 2018].

[8] "Linux-HA: Open Source High-Availability Software for Linux," [Online]. Available: http://www.linux-ha.org/wiki/Main_Page. [Accessed 27 April 2018].

[9] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson and A. Warfield, "Remus: High Availability via Asynchronous Virtual Machine Replication," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, 2008.

[10] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost and J. Furman, "Spanner: Google's Globally-Distributed Database," in *USENIX symposium on operating systems design and implementation (OSDI)*, Hollywood, CA, 2012.

[11] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding and J. Ferris, "TAO: Facebook's Distributed Data Store for the Social Graph," in *USENIX Annual Technical Conference (USENIX ATC)*, San Jose, CA, 2013.

[12] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett and H. V, "SPANStore: cost-effective geo-replicated storage spanning multiple cloud services," in *ACM symposium on operating systems principles (SOSP)*, Farminton, Pennsylvania, 2013.

[13] L. Lamport, "Paxos Made Simple," *ACM SIGACT News,* vol. 32, no. 4, pp. 18-25, 2001.

[14] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *USENIX Annual Technical Conference*, Philadelphia, PA, 2014.

[15] T. D. Chandra, R. Griesemer and J. Redstone, "Paxos made live: an engineering perspective," in *ACM symposium on principles of distributed computing*, Portland, Oregon, USA, 2007.

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: amazon's highly available key-value

store," in *Symposium on Operating systems principles (SOSP)*, Washington, USA, 2007.
[17] F. P. Junqueira, B. C. Reed and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, Hong Kong, 2011.
[18] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer and C. H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *ACM SIGOPS Operating Systems Review,* vol. 29, no. 5, pp. 172-182, 1995.
[19] D. B. Terry , V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera and H. Abu-Libdeh, "Consistency-based service level agreements for cloud storage," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2013.
[20] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News,* vol. 33, no. 2, pp. 51-59, 2002.
[21] R. Govindan, I. Minei, M. Kallahalla , B. Koley and A. Vahdat, "Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure," in *ACM SIGCOMM*, Florianopolis, Brazil, 2016.
[22] G. Phillipa , N. Jain and N. Nagappan, "Understanding network failures in data centers: measurement, analysis, and implications," in *ACM SIGCOMM*, Toronto, 2011.
[23] D. Turner, K. Levchenko, J. C. Mogul, S. Savage, A. C. Snoeren, D. Turner, K. Levchenko, J. C. Mogul, S. Savage and A. C. Snoeren, "On failure in managed enterprise networks," *Technical report HPL-2012-101, HP Labs,* 2012.
[24] D. Turner, K. Levchenko, A. C. Snoeren and S. Savage, "California fault lines: understanding the causes and impact of network failures," in *ACM SIGCOMM*, New York, NY, USA, 2010.
[25] "Apache Hadoop 2.6.0 - Fault Injection Framework and Development Guide," [Online]. Available: https://hadoop.apache.org/docs/r2.6.0/hadoop-project-dist/hadoop-hdfs/FaultInjectFramework.html. [Accessed April 2018].
[26] "Mockito framework," [Online]. Available: http://site.mockito.org/. [Accessed 27 April 2018].
[27] K. Beck, Test Driven Development: By Example., Addison-Wesley Professional, 2003.
[28] "Elasticsearch: RESTful, Distributed Search & Analytics," [Online]. Available: https://www.elastic.co/products/elasticsearch. [Accessed October 2018].
[29] Hazelcast, "Hazelcast: the Leading In-Memory Data Grid," [Online]. Available: https://hazelcast.com/. [Accessed October 2018].
[30] J. Kreps, N. Narkhede and J. Rao, "Kafka: a Distributed Messaging System for Log Processing," in *NetDB*, 2011.
[31] "MongoDB," [Online]. Available: https://www.mongodb.com/. [Accessed October 2018].
[32] "Redis: in-memory data structure store," [Online]. https://redis.io/. [Accessed October 2018].
[33] "VoltDB: In-Memory Database," [Online]. Available: https://www.voltdb.com/. [Accessed October 2018].
[34] A. Herr, "Veritas Cluster Server 6.2 I/O Fencing Deployment Considerations," Techincal report, Veritas Technologies, 2016.
[35] "OpenFlow Switch Specification, Version 1.5.1 (ONF TS-025)," Open Networking Foundation, 2015.
[36] "iptables: administration tool for IPv4 packet filtering and NAT," [Online]. Available: https://linux.die.net/man/8/iptables. [Accessed October 2018].
[37] "Ceph: distributed storage system," [Online]. Available: https://ceph.com/. [Accessed October 2018].
[38] "Apache ActiveMQ," [Online]. Available: http://activemq.apache.org/. [Accessed October 2018].
[39] "Ignite: Database and Caching Platform," [Online]. Available: https://ignite.apache.org/. [Accessed October 2018].
[40] "Terracotta data management platform," [Online]. Available: http://www.terracotta.org/. [Accessed October 2018].
[41] "Dkron: Distributed job scheduling system," [Online]. Available: https://dkron.io. [Accessed October 2018].
[42] "Infinispan: distributed in-memory key/value data store," [Online]. Available: http://infinispan.org/. [Accessed October 2018].
[43] "MooseFS: Moose file system," [Online]. Available: https://moosefs.com/. [Accessed October 2018].
[44] S. Jain, A. Kumar, M, S. al, J. Ong, L. Poutievski, A. Singh, S. Venkata, W, J. erer, J. Zhou and M. Zhu, "B4: Experience with a globally-deployed software defined WAN," *ACM SIGCOMM Computer Communication Review,* 2013.
[45] "Data Center: Load Balancing Data Center, Solutions Reference Nework Design," Technical report, Cisco Systems, Inc., 2004.
[46] "bnx2 cards intermittantly going offline," [Online]. https://www.spinics.net/lists/netdev/msg210485.html. [Accessed October 2018].

[47] A. S. Tanenbaum and Maarten van Steen, Distributed Systems: Principles and Paradigms, 2nd ed., Pearson, 2006.

[48] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver and R. Yerneni, "PNUTS: Yahoo!'s hosted data serving platform," *VLDB Endowment,* vol. 1, no. 2, pp. 1277-1288, 2008.

[49] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards and V. Bedekar, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *ACM Symposium on Operating Systems Principles (SOSP)*, New York, NY, USA, 2011.

[50] V. Ramasubramanian and E. G. Sirer, "Beehive: O(1)lookup performance for power-law query distributions in peer-to-peer overlays," in *Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, 2004.

[51] "OpenStack Swift object storage," [Online]. Available: https://www.swiftstack.com/product/openstack-swift. [Accessed October 2018].

[52] "RethinkDB: the open-source database for the realtime web," [Online]. Available: https://www.rethinkdb.com/. [Accessed October 2018].

[53] "Protocol Buffers," [Online]. Available: https://developers.google.com/protocol-buffers/. [Accessed Octrober 2018].

[54] jepsen-io, "Jepsen: A framework for distributed systems verification, with fault injection," [Online]. Available: https://github.com/jepsen-io/jepsen. [Accessed October 2018].

[55] "Clojure programming language," [Online]. Available: https://clojure.org/. [Accessed October 2018].

[56] "Apache HBase," [Online]. Available: https://hbase.apache.org/. [Accessed October 2018].

[57] "Riak KV: distributed NoSQL key-value database," [Online]. Available: http://basho.com/riak/. [Accessed October 2018].

[58] "Apache Cassandra," [Online]. Available: http://cassandra.apache.org/. [Accessed October 2018].

[59] "Aerospike database 4," [Online]. Available: https://www.aerospike.com/. [Accessed October 2018].

[60] "Apache Geode data management solution," [Online]. Available: http://geode.apache.org/. [Accessed October 2018].

[61] "Apache ZooKeeper," [Online]. Available: https://zookeeper.apache.org/. [Accessed October 2018].

[62] "RabbitMQ message broker," [Online]. Available: https://www.rabbitmq.com/. [Accessed October 2018].

[63] "Chronos: Fault tolerant job scheduler for Mesos," [Online]. Available: https://mesos.github.io/chronos/. [Accessed October 2018].

[64] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker and I. Stoica, "Mesos: a platform for fine-grained resource sharing in the data center," in *USENIX conference on Networked systems design and implementation (NSDI)*, Boston, MA, 2011.

[65] "Jepsen: MongoDB stale reads," [Online]. Available: https://aphyr.com/posts/322-jepsen-mongodb-stale-reads. [Accessed 17 March 2018].

[66] D. Yuan, Y. Luo, X. Zhuang, G. R. Rodrigues, X. Zhao, Y. Zhang, P. U. Jain and M. Stumm, "Simple testing can prevent most critical failures: an analysis of production failures in distributed data-intensive systems," in *USENIX conference on Operating Systems Design and Implementation (OSDI)*, Broomfield, CO, 2014.

[67] "Jepsen: Riak," [Online]. Available: https://aphyr.com/posts/285-call-me-maybe-riak. [Accessed 17 March 2018].

[68] "[SERVER-7008] socket exception [SEND_ERROR] on Mongo Sharding - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-7008. [Accessed 17 March 2018].

[69] "Network partition during peer discovery in auto clustering causes two clusters to form · Issue #1455 · rabbitmq/rabbitmq-server," [Online]. Available: https://github.com/rabbitmq/rabbitmq-server/issues/1455. [Accessed 17 March 2018].

[70] "[SERVER-17975] Stale reads with WriteConcern Majority and ReadPreference Primary - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-17975. [Accessed 17 March 2018].

[71] "[ENG-10389] Possible dirty read with RO transactions in certain partition scenarios - VoltDB JIRA," [Online]. Available: https://issues.voltdb.com/browse/ENG-10389. [Accessed 17 March 2018].

[72] "Possible write loss during cluster reconfiguration · Issue #5289 · rethinkdb/rethinkdb," [Online]. Available: https://github.com/rethinkdb/rethinkdb/issues/5289. [Accessed 17 March 2018].

[73] "[SERVER-14885] replica sets that disable chaining may have trouble electing a primary if

members have different priorities - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-14885. [Accessed 11 April 2018].
[74] "[ZOOKEEPER-2099] Using txnlog to sync a learner can corrupt the learner's datatree - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/ZOOKEEPER-2099. [Accessed 30 April 2018].
[75] "Disconnect between coordinating node and shards can cause duplicate updates or wrong status code · Issue #9967 · elastic/elasticsearch," [Online]. Available: https://github.com/elastic/elasticsearch/issues/9967. [Accessed 22 March 2018].
[76] "[HBASE-2312] Possible data loss when RS goes into GC pause while rolling HLog - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/HBASE-2312. [Accessed 1 May 2018].
[77] "[HDFS-577] Name node doesn't always properly recognize health of data node - ASF JIRA," [Online]. [Accessed 22 March 2018].
[78] "[MAPREDUCE-4819] AM can rerun job after reporting final job status to the client - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/MAPREDUCE-4819. [Accessed 18 March 2018].
[79] "[HDFS-1384] NameNode should give client the first node in the pipeline from different rack other than that of excludedNodes list in the same rack. - ASF JIRA," [Online]. Available: https://issues.apache.org/jira/browse/HDFS-1384. [Accessed 17 March 2018].
[80] "[SERVER-27125] Arbiters in pv1 should vote no in elections if they can see a healthy primary of equal or greater priority to the candidate - MongoDB," [Online]. Available: https://jira.mongodb.org/browse/SERVER-27125. [Accessed 17 March 2018].
[81] "minimum_master_nodes does not prevent split-brain if splits are intersecting · Issue #2488 · elastic/elasticsearch," [Online]. Available: https://github.com/elastic/elasticsearch/issues/2488. [Accessed 17 March 2018].
[82] "Avoid Data Loss on Migration - Solution Design," [Online]. Available: https://hazelcast.atlassian.net/wiki/spaces/COM/pages/66519050/Avoid+Data+Loss+on+Migration+-+Solution+Design. [Accessed 28 March 2018].
[83] "PSYNC2 partial command backlog corruption · Issue #3899 · antirez/redis," [Online]. Available: https://github.com/antirez/redis/issues/3899. [Accessed 1 May 2018].
[84] "Deadlock while syncing mirrored queues · Issue #714 · rabbitmq/rabbitmq-server," [Online]. Available: https://github.com/rabbitmq/rabbitmq-server/issues/714. [Accessed 1 May 2018].
[85] "Cassandra removenode makes Gossiper Thread hang forever," [Online]. Available: https://issues.apache.org/jira/browse/CASSANDRA-13562. [Accessed January 2018].
[86] "Redis Cluster Specification," [Online]. Available: https://redis.io/topics/cluster-spec. [Accessed 2018].
[87] "Distributed Semaphores with RabbitMQ," [Online]. Available: https://www.rabbitmq.com/blog/2014/02/19/distributed-semaphores-with-rabbitmq/. [Accessed 22 March 2018].
[88] "Consistency and Replication Model - Hazelcast Reference Manual," [Online]. Available: http://docs.hazelcast.org/docs/latest-development/manual/html/Consistency_and_Replication_Model.html. [Accessed 22 March 2018].
[89] "Jepsen: Redis," [Online]. Available: https://aphyr.com/posts/283-jepsen-redis. [Accessed 22 March 2018].
[90] "Jepsen: Hazelcast 3.8.3," [Online]. Available: https://jepsen.io/analyses/hazelcast-3-8-3. [Accessed 22 March 2018].
[91] "Jepsen: Aerospike," [Online]. Available: https://aphyr.com/posts/324-jepsen-aerospike. [Accessed 22 March 2018].
[92] "minimum_master_nodes does not prevent split-brain if splits are intersecting · Issue #2488 · elastic/elasticsearch," [Online]. Available: https://github.com/elastic/elasticsearch/issues/2488. [Accessed 17 May 2018].
[93] J. Kurose and K. Ross, Computer Networking: A Top-Down Approach, 7th ed., Pearson, 2016.
[94] "Floodlight OpenFlow Controller," [Online]. Available: http://www.projectfloodlight.org/floodlight/. [Accessed 19 March 2018].
[95] "CloudLab," [Online]. Available: https://www.cloudlab.us/. [Accessed May 2018].
[96] K. V. Vishwanath and N. Nagappan, "Characterizing cloud computing hardware reliability," in *ACM symposium on Cloud computing (SoCC)*, New York, NY, USA, 2010.
[97] R. Birke, I. Giurgiu, L. Y. Chen, D. Wiesmann and T. Engbersen, "Failure Analysis of Virtual and Physical Machines: Patterns, Causes and Characteristics," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, USA, 2014.
[98] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes and S. Quinlna, "Availability in Globally Distributed Storage Systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, BC, 2010.

[99] W. Jiang, C. Hu, Y. Zhou and A. Kanevsky, "Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics," *ACM Transactions on Storage ,* vol. 4, no. 3, p. Article 7, 2008.

[100] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin and A. D. Satria, "What Bugs Live in the Cloud? A Study of 3000+ Issues in Cloud Systems," in *ACM Symposium on Cloud Computing (SOCC)*, New York, NY, USA, 2014.

[101] S. Li, H. Zhou, H. Lin, T. Xiao, H. Lin, W. Lin and T. Xie, "A Characteristic Study on Failures of Production Distributed Data-Parallel Programs," in *International Conference on Software Engineering (ICSE)*, 2013.

[102] X. Chen, C. D. Lu and K. Pattabiraman, "Failure Analysis of Jobs in Compute Clouds: A Google Cluster Case Study," in *IEEE International Symposium on Software Reliability Engineering*, Naples, Italy, 2014.

[103] P. Garraghan, P. Townend and J. Xu, "An Empirical Failure-Analysis of a Large-Scale Cloud Computing Environment," in *IEEE International Symposium on High-Assurance Systems Engineering*, 2014.

[104] N. El-Sayed and B. Schroeder, "Reading between the lines of failure logs: Understanding how HPC systems fail," in *Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Budapest, Hungary, 2013.

[105] Y. Liang, Y. Zhang, A. Sivasubramaniam, M. Jette and R. Sahoo, "BlueGene/L Failure Analysis and Prediction Models," in *International Conference on Dependable Systems and Networks (DSN)*, Philadelphia, PA, USA, 2006.

[106] B. Schroeder and G. Gibson, "A Large-Scale Study of Failures in High-Performance Computing Systems," *IEEE Transactions on Dependable and Secure Computing,* vol. 7, no. 4, pp. 337 - 350, 2010.

[107] T. Benson, S. Sahu, A. Akella and A. Shaikh, "A first look at problems in the cloud," in *USENIX conference on Hot topics in cloud computing (HotCloud)*, Boston, MA, 2010.

[108] H. Zhou, J.-G. Lou, H. Zhang, H. Lin, H. Lin and T. Qin, "An empirical study on quality issues of production big data platform," in *International Conference on Software Engineering (ICSE)*, Florence, Italy, 2015.

[109] H. S. Gunawi, M. Hao, R. O. Suminto, A. Laksono, A. D. Satria, J. Adityatama and K. J. Eliazar, "Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages," in *ACM Symposium on Cloud Computing (SoCC)*, Santa Clara, CA, USA, 2016.

[110] A. Rabkin and R. H. Katz, "How Hadoop Clusters Break," *IEEE Software,* vol. 30, no. 4, pp. 88 - 94, 2012.

[111] R. Majumdar and F. Niksic, "Why is random testing effective for partition tolerance bugs?," in *ACM Journal on Programming Languages*, 2017.

[112] P. Godefroid, "Model checking for programming languages using verisoft.," in *ACM symposium on principles of programming languages (POPL)*, Paris, 1997.

[113] S. Qadeer and D. Wu, "Kiss: keep it simple and sequential," in *Conf. on Programming Language Design and Implementation (PLDI)*, 2004.

[114] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman and H. S. Gunawi, "SAMC: Semantic-Aware Model Checking for Fast Discovery of Deep Bugs in Cloud Systems," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[115] C. Baier and J.-P. Katoen, Principles of model checking, MIT press, 2008.

[116] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang and L. Zhou, "MODIST: Transparent model checking of unmodified distributed systems," in *USENIX Symposium on Networked Systems Design and Implementation, NSDI 2009*, 2009.

[117] P. Alvaro, J. Rosen and J. M. Hellerstein, "Lineage-driven Fault Injection," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, 2015.

[118] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen and D. Borthakur, "FATE and DESTINI: A framework for cloud recovery testing," in *Proceedings of NSDI'11: 8th USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[119] M. Pradel and T. R. Gross, "Automatic testing of sequential and concurrent substitutability," in *International Conference on Software Engineering (ICSE)*, 2013.

[120] T. Elmas, S. Tasiran and S. Qadeer, "VYRD: Verifying concurrent programs by runtime refinement-violation detection.," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.