Toward a Generic Fault Tolerance Technique for Partial Network Partitioning

Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany, *University of Waterloo, Canada*

https://www.usenix.org/conference/osdi20/presentation/alfatafta

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

Toward a Generic Fault Tolerance Technique for Partial Network Partitioning

Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, Samer Al-Kiswany
University of Waterloo, Canada

Abstract

We present an extensive study focused on partial network partitioning. Partial network partitions disrupt the communication between some but not all nodes in a cluster.

First, we conduct a comprehensive study of system failures caused by this fault in 12 popular systems. Our study reveals that the studied failures are catastrophic (e.g., lead to data loss), easily manifest, and can manifest by partially partitioning a single node.

Second, we dissect the design of eight popular systems and identify four principled approaches for tolerating partial partitions. Unfortunately, our analysis shows that implemented fault tolerance techniques are inadequate for modern systems; they either patch a particular mechanism or lead to a complete cluster shutdown, even when alternative network paths exist.

Finally, our findings motivate us to build Nifty, a transparent communication layer that masks partial network partitions. Nifty builds an overlay between nodes to detour packets around partial partitions. Our prototype evaluation with six popular systems shows that Nifty overcomes the shortcomings of current fault tolerance approaches and effectively masks partial partitions while imposing negligible overhead.

1 Introduction

Modern networks are complex. They use heterogeneous hardware and software [1], deploy diverse middleboxes (e.g., NAT, load balancers, and firewalls) [2, 3, 4], and span multiple data centers [2, 4]. Despite the high redundancy built into modern networks, catastrophic failures are common [1, 3, 5, 6]. Nevertheless, modern cloud systems are expected to be highly available [7, 8] and to preserve stored data despite failures of nodes, networks, or even entire data centers [9, 10, 11].

We focus our investigation on a peculiar type of network fault: *partial network partitions*[1], which disrupts the communication between some, but not all, nodes in a cluster. Figure 1 illustrates how a partial network partition divides a cluster into three groups of nodes, such that two groups (Group 1 and Group 2) are disconnected, but Group 3 can communicate with Groups 1 and 2.

In our previous work [12] we identified this fault and presented examples of how it leads to system failures. Other than our previous preliminary effort, we did not find any in-depth analysis of partial network partition failures and of their fault tolerance techniques. Nevertheless, we found 51 reports of failures caused by partial network partitioning faults[2] in the publicly accessible issue tracking systems of 12 production-quality systems (Section 4), numerous blog posts and discussions of this fault on developers' forums (Section 3), and eight popular systems with fault tolerance techniques specifically designed to tolerate this type of fault (Section 5).

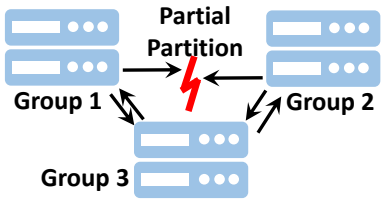


Figure 1: Partial partition. Groups 1 and 2 are disconnected, while Group 3 can reach both sides of the partition.

Our goal in this work is threefold. First, we aim to study failures caused by partial network partitioning to understand their impact and failure characteristics and, foremost, to identify opportunities to improve systems' resiliency to this type of fault. Second, we aim to dissect the fault tolerance techniques implemented in popular production systems and identify their shortcomings. Third, we aim to design a generic fault tolerance technique for partial network partitioning. This is the first work to characterize these failures and explore fault tolerance techniques for partial partitioning faults.

It is important to understand that *partial* partitions are fundamentally different from *complete* partitions [12]. Complete partitions split a cluster into two completely disconnected sides and are well studied with known theoretical bounds (CAP theorem [13]) and numerous practical solutions [14, 15, 16, 17]. On the contrary, a cluster experiencing a partial partition is still connected but not all-to-all connected. Consequently, the theoretical bounds of complete partitions do not apply to partial partitions, and fault tolerance techniques for complete partitions are not effective in handling partial partitions (Section 8).

An analysis of partial network partitioning failures. We conduct an in-depth study of 51 partial network partitioning failures from 12 cloud systems (Section 4). We select a diverse set of systems, including database systems (MongoDB and HBase), file systems (HDFS and MooseFS), an object storage system (Ceph), messaging systems (RabbitMQ, Kafka, and ActiveMQ), a data-processing system (MapReduce), a search engine (Elasticsearch), and resource managers (Mesos and DKron). For each considered failure, we carefully study the

[1] This is the commonly used name in failure reports and discussion forums.

[2] A *fault* is the initial root cause. If not properly handled, it may lead to a user-visible system *failure*.

failure report, logs, discussions between users and developers, source code, and code patches.

Failure Impact. Overall, we find that partial network partitioning faults cause silent failures with catastrophic effects (e.g., data loss and corruption) that affect core system mechanisms (e.g., leader election and replication).

Ease of manifestation. Unfortunately, these failures can easily occur. The majority of the failures are deterministic and require less than four events (e.g., read or write request) for the failure to occur. Even worse, all the studied failures can be triggered by partially partitioning a single node. The majority of failures do not require client access or can be triggered by clients only accessing one side of the partition.

Insights. We identify three approaches to improve system resilience: better testing, focused design reviews, and building a generic fault tolerance communication layer. Our analysis of each failure's manifestation sequence, access patterns, and timing constraints shows that almost all the failures could have been revealed through simple tests and by only using five nodes. Second, the majority of failures are due to design flaws. We posit that design reviews focused on network partitioning could identify these vulnerabilities. Third, building a generic communication layer to mask partial partitions is feasible, simplifies system design, and improves system resiliency.

Finally, we identify that a common deployment approach of Zookeeper introduces a failure vulnerability (Section 5). Our analysis shows that system designers need to design additional mechanisms to handle partial partitions when using Zookeeper or other external coordination services.

Dissecting modern fault tolerance techniques. We dissect the implementation of eight popular systems (VoltDB, MapReduce, HBase, MongoDB, Elasticsearch, Mesos, LogCabin, and RabbitMQ) and study the fault tolerance techniques they employ specifically to tolerate partial partitions (Section 5). For each system, we study the source code, analyze the fault tolerance technique's design, extract the design principles, and identify the technique's shortcomings. We identify four principled approaches for tolerating partial partitions: identifying the surviving clique, checking neighbors' views, verifying failures announced by other nodes, and neutralizing partially partitioned nodes.

Our analysis reveals that the studied fault tolerance techniques are inadequate. They either patch a specific system mechanism, which leaves the rest of the system vulnerable to failures, or unnecessarily shut down the entire cluster or pause up to half of the cluster nodes (Section 5).

Designing a generic fault tolerance technique. Our findings motivate us to build the **n**etwork part**i**tioning **f**ault-**t**olerance la**y**er (Nifty), a simple, generic, and transparent communication layer that can mask partial network partitions (Section 6). Nifty's approach is simple; it monitors the connectivity in a cluster through all-to-all heart beating, and when it detects a partial partition, it detours the traffic around the partition through intermediate nodes. Nifty overcomes all the shortcomings present in the studied fault tolerance techniques.

The main insight of Nifty is that tolerating partial partitioning does not require elaborate techniques such as the ones adopted by current systems (Section 5). Many modern systems already incorporate membership and connectivity monitoring mechanisms based on all-to-all heart beating [18, 19, 20]. Nifty shows that extending these mechanisms with a simple rerouting capability can effectively mask partial partitions.

To demonstrate Nifty's effectiveness, we deploy it with six systems: HDFS, Kafka, RabbitMQ, ActiveMQ, MongoDB, and VoltDB. We choose these systems because they are data intensive and popular systems. Furthermore, RabbitMQ and VoltDB implement generic techniques to tolerate partial partitions. Our prototype evaluation with synthetic and real-world benchmarks shows that Nifty effectively masks partial partitions while adding negligible overhead.

2 Definitions

A *partial network partition* is a network fault that prevents at least one node (e.g., a node in Group 1 in Figure 1) from communicating with another node (Group 2) in the system, while a third node (Group 3) can communicate with both affected nodes. Nodes in a partially partitioned cluster are still connected but are not all-to-all connected (i.e., they do not form a complete graph [21]). A partial partition divides a cluster into three groups: two sides and one bridge group. We identify a node as a *bridge* node if it can reach at least one node on each side of a partition. A partial partition has two *sides*, all the nodes on one side of the partition cannot reach all the nodes on the other side of the partition. We note that a cluster may suffer from multiple concurrent partial partitions.

We define a *single-node partial partition* as a partial partition that has a single node on one side of the partition, while the rest of the cluster nodes are bridge nodes or are on the other side of the partition. For instance, a single-node partial partition can be caused by a firewall misconfiguration that prevents a node from communicating with some other nodes.

3 Causes of Partial Network Partitioning

Recent reports indicate that network partitioning faults are common and happen at various scales. Connectivity loss between data centers [1] leads to network partitions in geo-replicated systems. Wide area network partitions happen as frequently as once every four days [6]. Switch failures can cause a network partition in a data center [5]. Switch failures caused 40 network partitions in two years at Google [3] and 70% of the downtime at Microsoft [5]. On a single node, NIC [22] or software failures can partition a node that may host multiple VMs. Finally, network partitions caused by cor-

Table 1: List of studied systems and the number of studied failures. The shaded rows are systems that implemented a fault tolerance technique for partial network partitioning.

System	Category	Failures	
		Total	Catastrophic
Elasticsearch [32]	Search engine	17	17
MongoDB [33]	Database	9	5
RabbitMQ [18]	Messaging	5	3
MapReduce [34]	Data processing	4	2
HBase [35]	Database	3	2
Mesos [36]	Resource mngr.	2	1
HDFS [34]	File system	3	1
Ceph [20]	Storage system	2	2
MooseFS [37]	File system	2	2
Kafka [38]	Messaging	2	2
ActiveMQ [39]	Messaging	1	1
DKron [40]	Resource mngr.	1	1
Total	**-**	**51**	**39**

related failures are common [4, 5, 6] and often caused by system-wide maintenance tasks [3, 5].

We found 51 failure reports detailing system failures due to partial network partitions, and numerous articles and online discussions discussing the fault [23, 24, 25, 26]. Some of these reports and discussions mention the root cause of the partial partition. Partial partitions are caused by a connectivity loss between two data centers [1] while both are reachable by a third center, the failure of additional links between racks [27, 28], network misconfiguration [29], firewall misconfiguration [29], network upgrades [30], and flaky links between switches [31]. Unfortunately, we did not find failure reports that detail partial partitioning faults in production networks.

4 Analysis of Partial Network-Partitioning Failures

We conduct an in-depth study of partial network partitioning failures reported in 12 popular systems (Table 1). We aim to understand the impact and characteristics of these failures and to identify opportunities for improving system resilience.

4.1 Methodology

We choose 12 diverse and widely used systems (Table 1), including two databases, a data analysis framework, two file systems, three messaging systems, a storage system, a search engine, and two distributed resource managers.

We selected the 51 failures in our study from publicly accessible issue-tracking systems. First, we used the search tools in the issue-tracking systems to find tickets related to partial network partitioning. Users did not classify network partitioning failures based on the partition type, so we had to search for all network partitioning failures and manually identified partial partitioning failures. We used the following keywords: "network partition," "partial network partition," "partial partition," "network failure," "switch failure," "isolation," "split-brain," and "asymmetric partition." Second, we considered tickets that were dated 2011 or later. Third, we excluded tickets marked as "Minor." For each ticket, we studied the failure description, system logs, developers' and users' comments, and code patches. For tickets that lacked enough details (e.g., missing output logs or did not have details about the affected mechanism), we manually reproduced them using NEAT [12]. Finally, during our evaluation, we found and reported bugs in Kafka and Elasticsearch. We included these failures in our study.

We differentiate failures by their manifestation sequences. In a few cases, the same faulty mechanism leads to two different failure paths. We count these as separate failures, even if they are reported in a single ticket. Similarly, although the exact failure is sometimes reported in multiple tickets, we count it once in our study.

4.2 Limitations

As with any characterization study, our findings may not be generalizable. Here, we list four potential sources of bias and describe our best efforts to address them.

1. *Representativeness of the studied systems*. Although we study 12 diverse systems (Table 1), our results may not be generalizable to systems we did not study. The selected systems follow diverse designs from strongly consistent (MongoDB, HBase, and Ceph) to eventually consistent (Elasticsearch) designs and from systems persisting data on disks and replicating data in-memory across nodes to caching systems. They follow a primary-backup or peer-to-peer architecture and use synchronous or asynchronous replication. The selected systems are widely used: Kafka, ActiveMQ, and RabbitMQ are the most popular open-source messaging systems; MapReduce, HDFS, and HBase are the core of the Hadoop platform; Elasticsearch is a popular search system; and MongoDB is a popular database.

2. *Limited number of tickets*. We study all 51 tickets that we found following our methodology. Statistical inference indicates that 30 samples can sufficiently represent the entire population [41]. More rigorously, if we assume the tickets we found represent a random sample of partial network partition failures in the wild, the central limit theorem predicts that our analysis of 51 tickets has a 13% margin of error at a 95% confidence level. To increase confidence in our findings, we only report findings that apply to at least two-thirds of the studied failures. A third of our findings apply to all failures.

Table 2: Failure impact and percentages of how many failures caused the corresponding impact.

Impact	%	
Data loss	23.5%	Catastrophic (74.5%)
System unavailability	21.6%	
Stale read	15.7%	
Data corruption	5.9%	
Dirty read	3.9%	
Data unavailability	3.9%	
Reduced availability	23.5 %	
Other	2%	

3. *Priority bias*. We include only high-impact tickets and avoid tickets marked by the developers as low-priority. This sampling methodology may bias the results.

4. *Observer error*. To reduce the chance for observer errors, two team members study every failure report using the same classification methodology. Then, we discuss the failure in a group meeting before reaching a verdict.

4.3 Findings

This subsection presents nine general findings. Our study indicates that partial network partitioning leads to catastrophic, silent failures. Surprisingly, these failures are easy to manifest. The majority of failures are deterministic, require a single-node partial partition, and require a few events to manifest. However, our study also identifies failure characteristics that can inform system designs and improve testing. Finally, we find that the majority of the studied failures are due to design flaws, indicating that developers do not expect networks to fail in this way.

Finding 1: *A significant percentage (74.5%) of the studied failures have a catastrophic impact.*
A failure is said to be catastrophic if it leads to a system crash or violates the system's guarantees (Table 2). Failures that reduce availability (e.g., crash of a single replica) or degrade performance are not considered catastrophic.

Data loss is the most common impact of partial network failures. For instance, in HBase, region servers store their logs on HDFS. When a log reaches a certain size, the region server creates a new log and informs the master of the new log location. If a partial partition isolates a region server from the master while both can reach HDFS, the master assumes that the region server has failed and assigns its logs to a new region server. If the old region server creates a new log, the master will not know about it, and the entries in the new log will be lost [42].

The second most common catastrophic impact of partial partitions is complete cluster unavailability, from which the majority of the studied systems suffer. A glaring example of this failure is the common deployment approach of Zookeeper. For instance, in ActiveMQ, a ZooKeeper service [43] monitors the cluster master and selects a new master if the current one fails. If a partial partition isolates the master from all ActiveMQ nodes while all nodes are reachable from ZooKeeper, the nodes will pause their operations because they cannot reach the master. Because ZooKeeper can reach the current master, it neither detects the problem nor selects a new master. The cluster remains unavailable until the partial partition heals [44]. Kafka and Mesos use Zookeeper in a similar fashion and suffer from a similar failure. The rest of the catastrophic failures lead to stale reads, data corruption, loss of data availability, and dirty reads.

In 23.5% of the failures, a partial partition unnecessarily reduces system availability. For example, leader election in MongoDB is based on a majority vote, with an arbiter node included to break ties. Unfortunately, this design leads to cluster unavailability under partial network partitions. For instance, consider a shard that has two replicas (A and B), with A being the leader. If a partial partition disrupts the communication between A and B while both can reach an arbiter, B will detect that A is unreachable and calls for a leader election. Because there is only one candidate in the system, the arbiter votes for it, and B becomes the leader. The arbiter will inform A of the new leader, and A steps down. A will detect that the leader (B) is unreachable, call for a leader election, become a leader, and then B steps down. This leader-election thrashing continues until the network partition heals [45]. The system is unavailable during leader election, so this failure significantly reduces system availability. We discuss the resolution of this failure in Section 5.

Finding 2: *Most of the studied failures (84.3%) are silent — the user is not informed about their occurrence.*
Despite the dangerous impact of partial partitioning faults, most systems do not report to the user that a failure has occurred. This is unsettling because a lack of error or warning notification delays failure detection. Some systems return a warning to the user when an operation fails due to partial network partitioning, but these warnings are ambiguous with no clear mechanisms for resolution. For example, in Elasticsearch, if a client sends a request to a replica that is partially isolated from the other replicas, the replica will return "a rejected execution" exception [46]. This confusing warning does not inform the user of the problem's actual cause nor the steps needed to resolve it.

Finding 3: *Leader election, configuration change, and replication protocol are the mechanisms most vulnerable to partial network partitioning.*
Leader election is the mechanism most vulnerable to partial network partitions (Table 3). In most cases, these failures lead to electing two leaders, one at each side of the partition [47, 48].

Configuration change is the second-most affected mechanism. For instance, each node in RabbitMQ maintains a membership log that lists the current nodes in the cluster. If

Table 3: Failure percentages per affected mechanism.

Mechanism	%
Leader election	37.3%
Configuration change	19.6%
Replication protocol	17.6%
Request routing	11.8%
Scheduling	5.9%
Data migration	5.9%
Data consolidation	2%

nodes have conflicting views on which nodes are part of the cluster, the RabbitMQ cluster crashes. For instance, in a cluster with three nodes (A, B, and C), when a partial partition disconnects B and C, B assumes that C crashed and removes it from the membership log, and C assumes that B crashed and removes it from the membership log. This inconsistency in the cluster membership leads to a complete cluster crash [49].

The replication mechanism is the third-most affected mechanism. For instance, if a partial partition in Elasticsearch isolates a shard's leader from the majority of that shard's replicas, the leader will wait for a period of time before stepping down. In this period, the leader continues to accept client write operations and acknowledges them before successfully replicating them [50]. If a client writes to the leader and later reads from one of the other replicas, it may read stale data.

Finding 4: *Most failures (60.8%) do not require client access or require only that clients access one side of the partition.*
To reduce the network partition's impact, some systems limit client access to one side of the partition [51, 52, 53]. However, our analysis shows that 60.8% of failures require no client access at all or only client access to one side of the partition. As an example of a failure that does not require client access, in MongoDB, balancer servers monitor the cluster load and migrate data chunks between nodes to rebalance the load across nodes. After rebalancing the data, a balancer updates Mongo's metadata server with the new data location. If during a re-balance operation of a particular shard a partial partition isolates the balancer from the metadata service, the cluster metadata will be in an inconsistent state, leading to the unavailability of that shard [54].

This finding highlights that system designers should consider the impacts of partial partitioning faults on all operations, including background operations.

Finding 5: *The majority of failures (68.6%) require three or fewer events (other than the partial partition) to manifest.*
Only a few events need to occur for a failure to happen. An event is a user request, a hardware or software fault, or a start of a background operation (e.g., leader election and data rebalancing). This is alarming because a small number of events can lead to catastrophic failures. Especially that in real deployments, many users interact with the system, increasing the probability of failure. Table 4 shows that, in 13.7% of failures, a partial partition, without any additional events, leads to a failure.

Table 4: Number of events required for a partial network partitioning failure to manifest.

Number of events	%
1 (Just a partial partition)	13.7%
2	9.8%
3	31.4%
4	13.7%
>4	31.4%

Table 5: System connectivity during a partial partition.

Network Partition Characteristic		%
Partition any node		33.3%
Partition a specific node		66.6%
• Leader	45.1%	
• Nodes with a special role	9.8%	
• A central service	7.8%	
• New nodes	2%	

Finding 6: *All the studied failures can be triggered by a single-node partial partition, with 33.3% of them happen by partitioning any node.*
Arguably, single-node partial partitions (Section 2) are generally more likely than partitioning more than one node. These partitions could happen due to a single ToR switch malfunction or by misconfiguring a single machine's firewall.

We further study which nodes need to be isolated for a failure to manifest (Table 5). Of the failures, 33.3% manifest by partitioning any node in the system—regardless of its role. Among the failures that require partitioning a specific node, partitioning the leader replica is most common (45.1%). In real deployments, partitioning a leader is likely because almost every node in the cluster is a leader for some shard. Partitioning a node with a special role (such as an arbiter in MongoDB) causes 9.8% of the failures.

Finding 7: *All the studied failures, except one, are deterministic or have known time constraints.*
Table 6 shows the timing constraints of the studied failures. Almost all the failures are either deterministic with no timing constraints (i.e., whenever the event sequence happens, a failure happens) or have known timing constraints, such as the period before considering a node to have failed. Only one failure is nondeterministic, as an interleaving of multiple threads causes it.

Table 6: Failures' timing constraints.

Timing constraint	%
No timing constraints	64.7%
Known timing constraints	33.3%
Nondeterministic	2%

Table 7: Percentage of design and implementation flaws.

Flaw type	%	Average Time to Resolution
Design	41.2%	260 days
Implementation	31.4%	98 days
Unresolved	27.5%	-

Table 8: Number of nodes needed to reproduce a failure.

Number of nodes	%
3 nodes	76.5%
4 nodes	21.6%
5 nodes	2%

Finding 8: *The resolution of 56.8% of the fixed failures required changing the design of a protocol or a mechanism.* We consider a code patch to be fixing a design flaw if it significantly changes the implemented protocol or logic, such as changing the mechanism to select a master in Elasticsearch.

Most of the fixed failures are caused by a design flaw (Table 7). This indicates that system designers overlook partial network partitioning failures in the design phase. We argue that a design review focused on partial partitions would detect a system's vulnerability to these failures.

Finding 9: *All failures can be reproduced with five nodes, and all but one can be reproduced using a fault injection tool.* These failures can be easily reproduced with small clusters of five or fewer nodes (Table 8), and 76.5% require only three nodes. Furthermore, all the failures except one can be reproduced using a fault-injection framework that can inject partial partitioning faults such as NEAT [12].

4.4 Insights

Our analysis shows that partial partitions lead to catastrophic silent failures that are easy to manifest, are deterministic, and can be triggered by a single-node partial partition and a sequence of a few events.

Fortunately, we identify three approaches for improving system resilience to partial partitions. First, because these faults are deterministic and can be reproduced on a small cluster, improved testing can reveal the majority of the studied failures. Our analysis finds timing, client access, and partition characteristics that significantly reduce the number of sufficient test cases. Second, our study of the code patches reveals that focused design reviews can identify system vulnerabilities early in the design process.

Third, partial network partitions have two characteristics that imply that a generic fault tolerance technique is possible. These faults can be detected by exchanging information between the nodes, and by definition, there are alternative paths in the network to reconnect the system. We leverage these two characteristics in building Nifty (Section 6).

Most of the studied failures are caused by the underlying assumption that, if a node can reach a service, all nodes can reach that service, and if a node cannot reach a service then the service is down. Our analysis shows the danger of such assumptions; this leads to a confusing state, wherein some of the system's parts start executing a fault tolerance mechanism, while others presume the whole system is healthy and carry on normal operations. The mix of these two operation modes is poorly understood and tested.

Finally, we identify that a common usage of external coordination services (e.g., Zookeeper) introduces a vulnerability to partial network partitioning fault. System designers need to build additional techniques to detect and handle partial partitions when using external coordination services.

5 Dissecting Modern Fault Tolerance Techniques

We studied the code patches related to the tickets included in our study. Six of the systems in Table 1 (MongoDB, Elasticsearch, RabbitMQ, HBase, MapReduce, and Mesos) changed the system design to incorporate a fault tolerance technique specific to partial network partitioning faults. The rest of the systems either patched the code with an implementation-specific workaround or did not fix the reported bugs yet.

Furthermore, we found that two additional systems, VoltDB [19, 55] and LogCabin [56] (the original implementation of the Raft [14] consensus protocol), implement fault tolerance techniques for partial partitions. For these two systems, we did not find failure reports related to partial partitioning faults in their issue tracking systems, but VoltDB announced that their recent version tolerates partial partitions [57]. We experimented with LogCabin to understand the impact partial partitions have on strongly consistent systems and found that LogCabin incorporates a technique to tolerate partial partitions. We included VoltDB and LogCabin in our study.

For each of the eight systems, we study the source code, and extract and analyze the design principles of their fault tolerance technique. We identify four approaches for tolerating partial partitions: detecting a surviving clique of nodes, checking neighbors' views, verifying failure reports received from other nodes, and neutralizing one side of the partial partition. Unfortunately, these techniques have severe shortcomings that may lead to a complete system shutdown or to the unavailability of a major part of the system. In this section, we detail these techniques and discuss their shortcomings.

5.1 Identifying the Surviving Clique

Main idea. Upon a partial network partition, the system identifies the maximum clique of nodes [58], which is the largest subset of nodes that are completely connected. All nodes that are not part of the maximum clique are shut down. VoltDB follows this approach.

VoltDB Implementation. VoltDB [19, 55] is a popular ACID, sharded, and replicated relational database. VoltDB

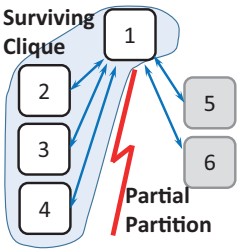


Figure 2: VoltDB's surviving clique. Gray nodes shut down as they are not in the clique.

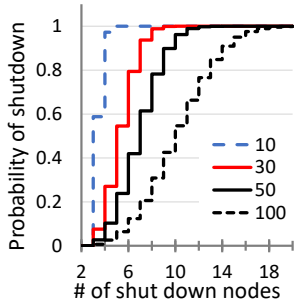


Figure 3: The probability of a VoltDB cluster shutdown. Different lines represent different cluster sizes. The x-axis shows the number of nodes that are not in the clique.

follows a peer-to-peer approach to implement this technique. Every node in the system periodically sends a heartbeat to all nodes. If a node loses connectivity to any node, it suspects that a partial network partition occurred and starts the recovery procedure. The recovery procedure has two phases. In the first phase, the node that detects the failure broadcasts a list of nodes it can reach. When a node in the cluster receives this message, it broadcasts its list of reachable nodes to all nodes in the cluster. In phase two, every node independently combines the information from the other nodes into a graph representing the cluster connectivity. Each node analyzes this graph to detect the maximum completely connected clique of nodes. Every node that finds that it is not part of this "surviving" clique shuts itself down. Figure 2 shows an example in which a partial partition disrupts the communication between nodes 2, 3, and 4 on one side and nodes 5 and 6 on another. Nodes 5 and 6 are not part of the clique and will shut down.

After identifying the surviving clique, the system verifies that it did not lose any data by verifying that the surviving clique has at least one replica of every data shard. If the clique is missing one shard, such as when all the replicas of a shard are shut down, the entire system shuts down.

Shortcomings. This fault tolerance approach has two severe shortcomings. First, it unnecessarily shuts down up to half of the cluster nodes, reducing the system's performance and fault tolerance. Second, this approach causes a complete cluster shutdown if the surviving clique is missing a single data shard. To understand how likely a cluster is to shut down, we conduct a probabilistic analysis (detailed in our technical report [59]). Figure 3 shows the probability of a complete cluster shutdown while varying the cluster size and the number of nodes that shut down (i.e., nodes that are not part of the surviving clique – the x-axis in Figure 3). Each shard has three replicas. Our analysis shows that isolating only 10% of the nodes leads to more than a 50% probability of shutting down the entire cluster, and isolating only 20% of the nodes leads to a staggering 90% chance of a complete cluster shutdown.

5.2 Checking Neighbors' Views

Main idea. When one node (e.g., node S) loses its connection to another node D, it verifies whether the connection is lost due to a partial partition. To this end, S asks all nodes in the cluster whether they can reach D. If a node reports that it can reach D, this indicates that the cluster is suffering a partial network partition.

If S detects a partial network partition, S either disconnects from all nodes that can reach D, which effectively makes the partition a complete partition, or pauses its operation. RabbitMQ and Elasticsearch follow this approach.

5.2.1 RabbitMQ

RabbitMQ [18] is a popular messaging system that replicates message queues for reliability. In RabbitMQ, if a node detects that its communication with another node (e.g., node D) is affected by a partial partition, it applies one of the following policies depending on its configuration.

1. *Escalate to a complete partition.* The node will drop its connection with any node that can reach node D. The goal of this policy is to create a complete partition in which both sides work independently. This configuration leads to data inconsistency and requires running a data consolidation mechanism after the partition heals.

2. *Pause*: To avoid data inconsistency, once a node discovers the partial partition, it pauses its activities. It resumes its activities only when the partition heals. The result of this policy is that a subset of nodes will continue to operate. This subset will be completely connected and will run without sacrificing data consistency.

3. *Pause if anchor nodes are unreachable*: RabbitMQ's configuration can specify a subset of nodes to act as anchor nodes. If a node cannot reach any of the anchor nodes, it pauses. This may lead to creating multiple complete partitions if the anchor nodes become partially partitioned. This may lead to pausing all nodes if all the anchor nodes are isolated.

After a partition heals, RabbitMQ employs two data consolidation techniques: administrator intervention, in which the administrator decides which side of the partition should become the authoritative version of the data, and auto-heal, in which the system makes this determination based on the number of clients connected to each side. Both techniques may lead to data loss or inconsistency [12].

Shortcomings. RabbitMQ's policies have serious shortcomings. Changing a partial partition to a complete partition (policies 1 and 3) may lead to multiple inconsistent copies of the data, whereas the *pause* policy (policy 2) may pause the entire system or the majority of the nodes. For instance, in Figure 4, if every node except node 1 detects that it cannot reach a

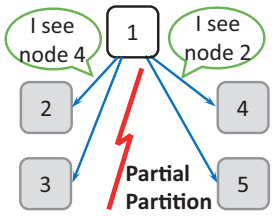


Figure 4: A scenario for RabbitMQ's *pause* policy. Every non-bridge node pauses (gray nodes) as it detects that it cannot reach one node on the other side.

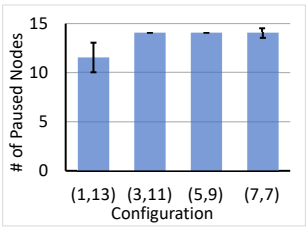


Figure 5: The median number of paused nodes in a cluster of 15 nodes. In all runs, one node is unaffected by the partition. The notation (i, j) shows the number of nodes on each side of the partition.

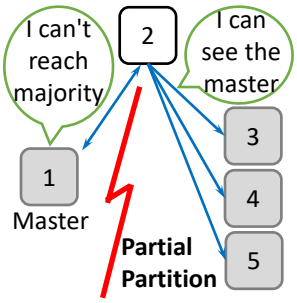


Figure 6: Elasticsearch unavailability scenario. The master pauses because it cannot reach majority of nodes, and all nodes pause because they cannot reach the master.

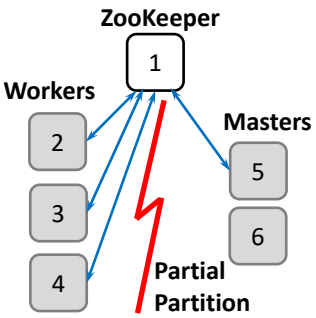


Figure 7: A Mesos cluster becomes unavailable when a partial partition isolates the master node and its backups.

node on the other side of the partition, it pauses, leading to a complete cluster pause.

In the case of the *pause* policy (policy 2), to determine how many nodes pause under different partial partition scenarios, we conduct an experiment in which we deploy a 15-node RabbitMQ cluster, introduce a partial partition, and observe how many nodes pause. In all experiments, we inject a partition such that one node remains unaffected and able to reach all nodes. Figure 5 shows the median number of paused nodes under various partition configurations. We run each configuration 30 times. Surprisingly, in all configurations almost all the cluster nodes pause because each node detects that it cannot reach at least one node on the other side of the partition. Even isolating a single node (configuration (1,13) in Figure 5) leads to pausing 12 nodes. Our investigation reveals that nodes declare another node unreachable after missing its heartbeats for a timeout period. In RabbitMQ, the default timeout period is 1 minute, which gives enough time for many nodes to detect the partition and pause. Using a shorter timeout periods causes some nodes to declare prematurely that other nodes have failed, even without a partial partition.

5.2.2 Elasticsearch

Elasticsearch [32] is a popular search engine. Its master election protocol uses a fault tolerance technique based on checking neighbors' views. In Elasticsearch, the node with the lowest ID is the master. If a node (e.g., S) cannot reach the master, it contacts all nodes to check whether they can reach the master. If any node reports that it can reach the master, S pauses its operations. If none of the nodes can reach the master, the node with the lowest ID becomes the new master.

Shortcomings. First, this approach can affect cluster availability quite severely, as all nodes that cannot reach the master pause. In the worst case, it can cause a complete cluster unavailability. For instance, in Figure 6, none of the nodes can reach the master except node 2, which refuses to become the new master because it can reach a node with a lower ID (node 1). Consequently, all the nodes in the cluster pause. Furthermore, because the master cannot reach a majority of nodes, it also pauses, which leads to system unavailability [60]. Second, Elasticsearch uses this approach only to fortify the master election protocol, which leaves the rest of the system vulnerable to partial partitions.

5.3 Failure Verification

Main idea. If a node (e.g., S) receives a notification from another node that a third node (D) has failed, node S first verifies that it cannot reach D before taking any fault tolerance steps. This approach is used in the leader election protocols of MongoDB [33] and LogCabin [56]. It was also used in an earlier version of Elasticsearch.

In MongoDB and LogCabin, if a leader is on one side of a partial partition but can still reach the majority of nodes, the nodes on the other side of the partition unnecessarily call for leader election. Finding 1 in Section 4 discusses a scenario in which a partial partition leads to continuous leader election thrashing and to system unavailability [45]. To avoid unnecessary elections, when a node receives a call for election, it first verifies that the current leader is unreachable. A node participates in an election only if it cannot reach the current leader, else it will ignore the failure report.

Shortcomings. This approach has two major shortcomings. First, it leads to the unavailability of a large number of nodes. Second, it is mechanism specific. Designing a system-wide fault tolerance mechanism using this approach is tricky because one cannot ignore every failure notification. For instance, using this approach in an earlier version of Elasticsearch backfired [61]. During data migration from a primary replica of a shard to a secondary replica, if a partial partition isolates the primary replica from the secondary replica while both are reachable from the master node, the primary requests a new secondary replica. Because the master can reach the secondary replica, it ignores the failure report. This leads to the unavailability of the affected shard [61]. Broadly applying

Table 9: Summary of shortcomings. (D) indicates that the nodes shut down. (P) indicates that the nodes pause until the partition heals. In the worst case, RabbitMQ pauses all nodes except one. We consider this a complete cluster loss (1). Under different RabbitMQ policies, (2) and (3) can occur. (S) indicates a system-wide technique, whereas (M) is a mechanism-specific technique.

	Surviving Clique	Checking w/ Neighbors		Failure Verification	Neutralizing Nodes		Nifty
	VoltDB	Elasticsearch	RabbitMQ	MongoDB/LogCabin	MapReduce/HBase	Mesos	
Reduced Availability	$\times^{D}$	$\times^{P}$	$\times^{P}$	$\times^{P}$	$\times^{D}$	$\times^{P}$	
Complete Unavailability	$\times$	$\times$	$\times^{1}$				
Complete Partition			$\times^{2}$				
Double Execution						$\times$	
Data Unavailability			$\times^{3}$				
Scope (System/Mechanism)	S	M	S	M	M	M	S

this fault tolerance technique is not feasible because designers have to revisit the design of every system mechanism, consider the consequences of ignoring failure reports, and examine the interaction of various mechanisms under partial partitions.

5.4 Neutralizing Partitioned Nodes

Main idea. One challenge related to handling partial network partitions is that nodes may update a shared state that is reachable from both sides of the partition, leading to data loss and inconsistency. To avoid this problem, this approach attempts to neutralize one side of the partition. However, the neutralization method is implementation-specific. HBase, MapReduce, and Mesos use this approach.

HBase Implementation. In HBase, data shards are managed by an HBase node but are stored on HDFS. If the HBase leader cannot reach one of the HBase nodes, it neutralizes that node by renaming the shard's directory in HDFS. Renaming a shard's directory effectively prohibits the old HBase node from making further changes to the shard [42]. The leader then assigns the shards of that node to a new HBase node.

MapReduce Implementation. In MapReduce, a manager node assigns tasks to AppMaster nodes. If the manager cannot reach an AppMaster, it reschedules the tasks assigned to that AppMaster to a new AppMaster. With partial network partitions, this approach may result in two AppMasters working on the same task, which leads to data corruption [62]. To fix this problem, when an AppMaster completes a task, it writes a completion record in a shared log on HDFS. Before an AppMaster executes a new task, it checks the shared log for a completion record. If it finds one, it does not re-execute the task.

Mesos Implementation. In Mesos, a master node assigns tasks to worker nodes. A Zookeeper instance selects the master node. The master sends periodic heartbeats to workers. If a partial partition isolates a worker node from the master, it pauses its operations. Figure 7 shows a worst-case scenario in which the partial partition isolates the master and its backup from all workers, which leads to a complete cluster unavailability. Finally, if a master detects that one of the workers is unavailable, it marks the tasks that were running on the unreachable worker as lost and reschedules them on new workers. This may lead to the double execution of a task [63].

Shortcomings. First, it is not practical to use this approach for system-wide fault tolerance, as this approach is specific to a certain protocol and implementation. The presented three systems use this approach for different mechanisms. To use this approach broadly, designers must go through the daunting task of independently designing a fault tolerance technique for every mechanism in the system and understanding the interaction between these mechanisms. Second, this approach leaves the nodes on one side of the partition idle, which reduces system performance and availability.

5.5 Summary

Table 9 summarizes the shortcomings of the current fault tolerance techniques, none of which are adequate for modern cloud systems. All current techniques severely affect system availability, as they unnecessarily lose a significant number of nodes. Failure verification and neutralizing partitioned nodes are used to fortify *specific* mechanisms, rather than providing *system-wide* fault tolerance. Using mechanism-specific fault tolerance techniques requires the independent fortification of all system mechanisms and the analysis of the interactions between various mechanisms. This approach complicates system design, fault analysis, and debugging. An example of a system that uses multiple mechanism-specific techniques to tolerate partial partitions is Elasticsearch, which uses checking neighbors' view, failure verification [61], and neutralizing partitioned nodes [64] in different mechanisms. However, Elasticsearch has the highest number of reported failures due to partial partitions (Table 1).

Detecting the surviving clique and checking neighbors' views can be used to build a *system-wide* fault tolerance technique. However, as Table 9 shows, these techniques lead to a complete system shutdown or significant loss of system capacity. This realization motivated us to build Nifty (Section 6), a system-wide fault tolerance technique that overcomes the aforementioned shortcomings.

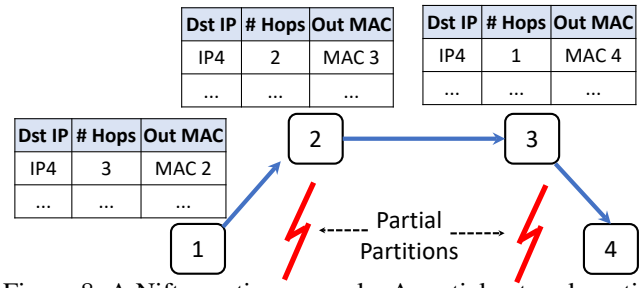


Figure 8: A Nifty routing example. A partial network partition isolates node 1 from nodes 3 and 4, and another partial partition isolates node 4 from nodes 1 and 2. Communication between 1 and 4 is routed through nodes 2 and 3.

6 Nifty Design

To overcome the limitations of current fault tolerance techniques, we design a simple, transparent network-partitioning fault-tolerant communication layer (Nifty).

Nifty follows a peer-to-peer design in which every node in the cluster runs a Nifty process. These processes collaborate in monitoring cluster connectivity. When Nifty detects a partial partition, it reroutes the traffic around the partition through intermediate nodes (i.e., bridge nodes). For instance, in Figure 8, if two partial partitions isolate node 1 from node 4, Nifty reroutes packets exchanged between nodes 1 and 4 through nodes 2 and 3.

Although Nifty keeps the cluster connected, it may increase the load on the bridge nodes, leading to a lower system performance. System designers who use Nifty may optimize the data or process placement or employ a flow-control mechanism to reduce the load on bridge nodes. To facilitate system-specific optimization, Nifty provides an API to identify bridge nodes.

Connectivity monitoring. Each Nifty process uses heart beating to monitor its connectivity with all other Nifty processes. Each Nifty process maintains a distance vector that includes the distance, in number of hops, to every node in the cluster. If a Nifty process misses three heartbeats from another Nifty process, it assumes that the communication with that process is broken and updates its distance vector. To detect when the communication between nodes recovers, Nifty processes continue to send heartbeats to disconnected nodes.

Recovery. Each Nifty process sends its distance vector (piggybacked on heartbeat messages) to all other nodes. Every Nifty process then uses these vectors to build and maintain a routing table.

When a Nifty process detects a change in the cluster (e.g., a node becomes unreachable or reachable), it initiates the route discovery procedure to find new routes. In our prototype, we use the classical Bellman–Ford distance-vector protocol [65, 66]. We use hop count as the link weight. By hop, we mean a hop between end nodes. Using hop count naturally favors direct connections, when they exist, over rerouting through intermediate nodes.

An entry in the routing table has a destination IP address, hop count, and output MAC address. If a packet is received with a destination IP address that matches an entry in the routing table, Nifty will change the destination MAC address of the packet to equal the output MAC address found in the routing table, then send the packet out.

Route deployment. Nifty uses OpenFlow [67] and Open vSwitch [68] to deploy the new routes. For instance, to reroute packets sent from node 1 to node 4 through nodes 2 and 3 in Figure 8, the Nifty process on node 1 installs rules on its local Open vSwitch to change the destination MAC address of any packet destined to node 4 to the MAC address of node 2. Whenever node 2 receives a packet with node 4 IP address as its destination, it changes the destination MAC address to node 3 MAC address and sends the packet out. Finally, when node 3 receives a packet with node 4 IP address, it changes the MAC address to node 4 MAC and sends the packet out.

Node classification. A system using Nifty can be optimized to reduce the amount of data forwarded through bridge nodes. The approach to do so is system-specific and may entail relocating processes in a cluster, dropping client requests, or reducing query result quality [7].

To facilitate the implementation of these mechanisms, Nifty identifies which nodes are on the same side of the network partition and which nodes serve as bridge nodes. It then provides this node classification to the system running atop of it. Section 7.3 demonstrates how this information can facilitate optimizing process placement in a VoltDB cluster.

7 Evaluation

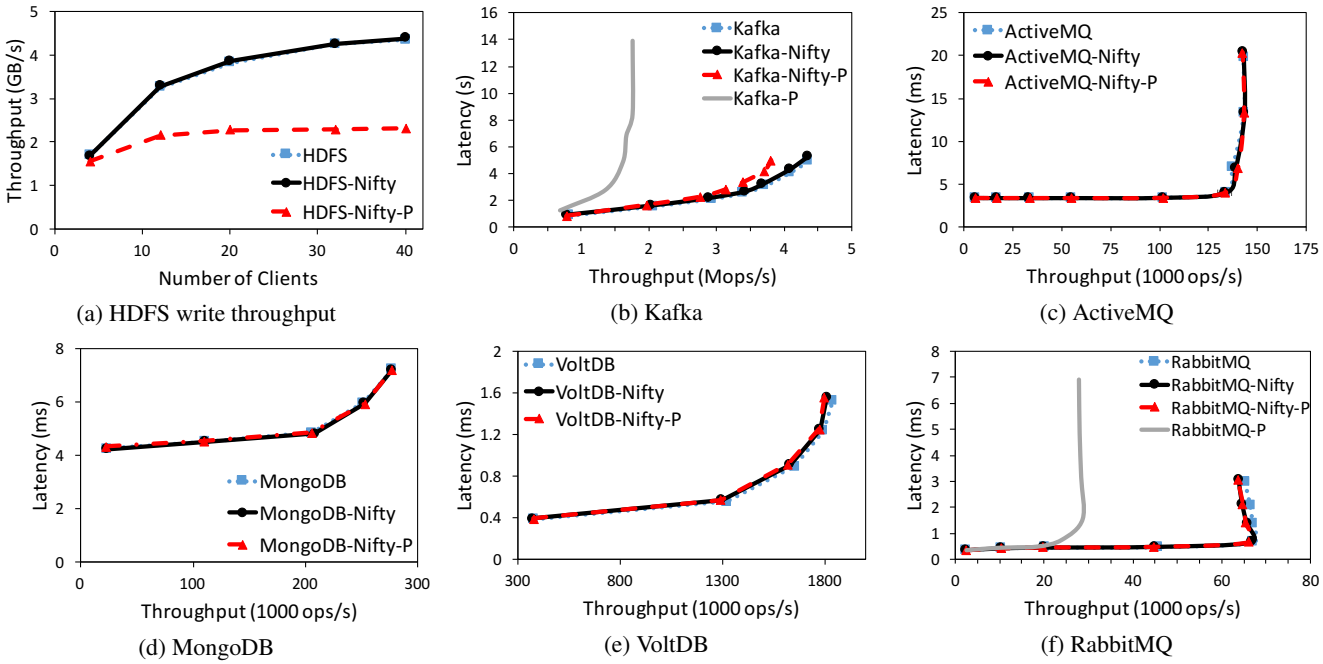
Our evaluation answers three questions. How much overhead does Nifty impose when there are no network partitions? What is a system's performance with Nifty under a network partition? What is the utility of Nifty's classification API?

Testbed. We conduct our experiments using 40 nodes at the Cloudlab Utah cluster. Each node has an Intel Xeon E5 10-core CPU, 64 GB of RAM, and a Mellanox ConnectX-4 25 Gbps NIC. To inject a network partition fault, we modify the Open vSwitch rules on the nodes to drop packets between the affected nodes. In all our experiments, we report the average for 30 runs. We note that the standard deviation in all our experiments is lower than 5%.

7.1 Overhead Evaluation

To evaluate Nifty's overhead, we measure its impact on the performance of a synthetic benchmark using iperf [69] and six data-centric systems (i.e., storage, database, and messaging systems). The iperf experiment uses a 100-node cluster to measure Nifty's impact on larger clusters. The systems we selected are:

- HDFS: We deploy HDFS (v3.3.0) on six nodes (one name node and five data nodes) and with a replication



(a) HDFS write throughput (b) Kafka (c) ActiveMQ

(d) MongoDB (e) VoltDB (f) RabbitMQ

Figure 9: Nifty's overhead. The average throughput for HDFS (a) and the average throughput vs. average latency for the rest of the systems. (-P) denotes the results with a partial partition.

level of three. To avoid disk access, we configure data nodes to use tmpfs. We use the HDFS standard benchmark (TestDFSIO). The benchmark reads and writes 1 GB files.

- Kafka: We deploy Kafka (v2.6.0) on five nodes. We distribute the queues (aka, topics) among nodes to balance the load. Each message is replicated on three nodes. We use Kafka's benchmarking tool to generate load on the system. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer.

- ActiveMQ: We deploy ActiveMQ Artemis (v2.15.0) on five nodes with each queue being replicated on two nodes. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer.

- MongoDB: We deploy MongoDB (v4.4.1) on six nodes (one config server and five mongod nodes) and with a replication level of three. We discuss our results with the Yahoo benchmark workload B (95% reads and 5% writes) with a uniform distribution [70]. We use 10 million records. The rest of the Yahoo benchmark workloads shows similar results.

- VoltDB: We deploy VoltDB (v9.0) on nine nodes, with data sharding enabled and a replication level of three. We use the Yahoo benchmark and the TCP-C benchmark. Figure 9.e shows the throughput-latency curve under Yahoo benchmark workload B (95% reads and 5% writes) with a uniform distribution. The results using the TPC-C benchmark and the Yahoo benchmark workloads A and C with uniform and skewed loads show similar low overhead.

- RabbitMQ: We deploy RabbitMQ (v3.8.2) on three nodes. We use the mirrored mode in which each queue has a leader replica and two backup replicas. We distribute the queue masters among brokers to distribute the load. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each consumer reads messages from a dedicated queue.

Results. We compare the throughput and average latency of each system with and without Nifty when there is no partial network partition. We evaluate Nifty with a partial partition in Section 7.2.

Figure 9 shows the write throughput of HDFS (Figure 9.a) and the throughput-latency curve for Kafka (Figure 9.b), ActiveMQ (Figure 9.c), MongoDB (Figure 9.d), VoltDB (Figure 9.e), and RabbitMQ (Figure 9.f). The results show that Nifty does not add noticeable overhead; for all systems, the curves almost completely overlap. This is because Nifty processes exchange a negligible number of packets. Each Nifty process sends a single UDP heartbeat packet every 200 ms to other nodes in the system. Consequently, in the largest deployment of nine nodes, each node sends only 40 packets every second.

Scalability evaluation. Nifty uses all-to-all heart beating to monitor a cluster's connectivity. Consequently, Nifty's overhead increases with the cluster size. To measure Nifty's scal-

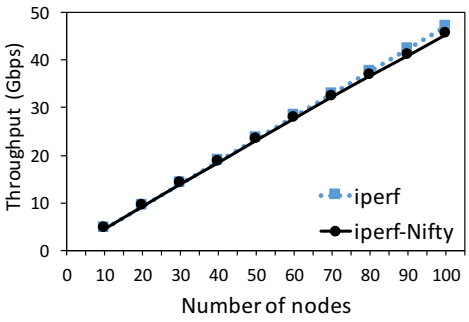


Figure 10: Scalability evaluation. Average throughput while increasing the number of nodes.

ability, we evaluate its overhead on a 100-node CloudLab Utah cluster. For this experiment, we limit the throughput of each node to 1 Gbps, as CloudLab can not support a full 10-Gbps connectivity between the 100 nodes we managed to book. To generate network intensive load, we use iperf [69]. Half of the nodes run an iperf server, and the other half run an iperf client. Each client communicates with a single server. Figure 10 shows the aggregate throughput of the iperf servers when deployed with and without Nifty. The figure shows that Nifty's overhead is negligible. When using 100 nodes, Nifty degrades the aggregate throughput by only 3.5%. Nevertheless, this monitoring approach will not scale to clusters with thousands of nodes. We are currently exploring the design of a fault tolerance technique that can scale to larger clusters.

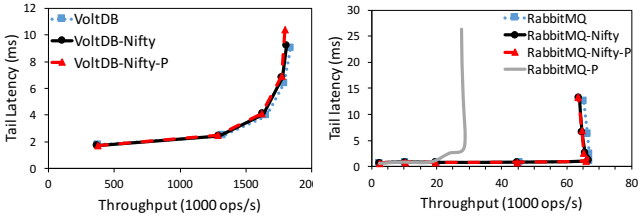
7.2 Handling Partial Partitions

To demonstrate the effectiveness of the proposed approach, we evaluate Nifty's performance with the six aforementioned systems under a partial partition fault. We note that RabbitMQ and VoltDB implemented two different techniques for tolerating partial partitions (Section 5).

Partial partition setup. We use the same deployment of the six aforementioned systems. Each system is deployed on an odd number of replicas. We introduce a partial partition that leaves one node as a bridge node and puts an equal number of nodes on each side of the partition. Client nodes are not affected by the partition. We partition the cluster this way to create maximum pressure on the bridge node.

Figure 9 shows the system performance when the cluster suffers from the partial partition. We notice that all the six systems are severely effected by the partial partition. HDFS, ActiveMQ, MongoDB, and VoltDB suffer a complete cluster pause or shutdown when deployed without Nifty. The VoltDB cluster shuts down because, after detecting the surviving clique, the system misses at least one shard. This confirms our analysis in Section 5.1.

RabbitMQ uses the checking neighbor's views fault tolerance approach. In our deployment, each queue is mirrored on a backup replica. Due to the strong consistency requirement, we configure RabbitMQ to pause in case of partial partition.



(a) VoltDB tail latency. (b) RabbitMQ tail latency.

Figure 11: Tail latency evaluation. Average throughput vs. 99th percentile of latency.

We deploy RabbitMQ on three nodes. Unfortunately, we could not use a larger RabbitMQ cluster because partial partitions often lead to the pause of the entire RabbitMQ cluster when Nifty is not used (Figure 4). Even with three nodes, partial partitions sometimes lead to pausing two out of three nodes. We discard those results and only include results in which one node pauses. Consequently, our results show the best possible performance of RabbitMQ under partial partitions. Pausing a broker in RabbitMQ leads to more than 50% reduction in throughput (RabbitMQ-P in (Figure 9.f)).

Kafka uses Zookeeper to monitor a cluster nodes. If a partial partition isolates a queue leader from the majority of replicas while Zookeeper runs on a bridge node, Zookeeper will not select a new leader and the entire cluster pauses (Finding 1 in Section 4). To mitigate this, we made sure that Zookeeper falls on one side of the partition. In this case, all the nodes on the other side of the partition that cannot reach Zookeeper are removed from the cluster. In our experiment, the partial partition causes two nodes to pause, which leads to almost a 50% reduction in system throughput (Figure 9.b).

Figure 9 shows that Nifty effectively masks the partial partition, so none of the nodes shut down or pause. Figure 9.a shows the write operation throughput for HDFS. With a replication level of three, each file has replicas on both sides of a partial partition. Consequently, for every 1 GB of data written, 1 or 2 GB of data are rerouted through the bridge node. This reduces the system throughput by up to 45%. We note that having a partial partition result in a performance degradation is better than a complete system unavailability when HDFS is deployed without Nifty. For the rest of the systems, during the partial partition, almost 50% of client requests and responses are rerouted through the bridge node. Even so, the system throughput only decreases by 2-6.7% and latency only increases by 3-7.8%. This shows that Nifty can effectively mask partial partitions and is able to utilize remaining connections to reduce the performance impact.

Figure 11 shows the tail latency for VoltDB and RabbitMQ for the same experiments presented in Figure 9. The figure shows the average throughput and the 99th percentile of latency while increasing the load on the system. The figure shows that Nifty increases the 99th percentile latency by up to 6.8% without a partial partition and by 15% under a partial partition failure.

7.3 Classification API Utility

In this section, we demonstrate the utility of Nifty's classification API. In VoltDB, a single server (aka, multi-data-partition initiator or MPI) processes all multi-shard operations. The MPI divides a multi-shard query (e.g., a join) to sub-queries, such that each sub-query targets a single shard. The MPI forwards each sub-query to its shard leader, gathers the intermediate results, performs final query processing, and sends the result to the client.

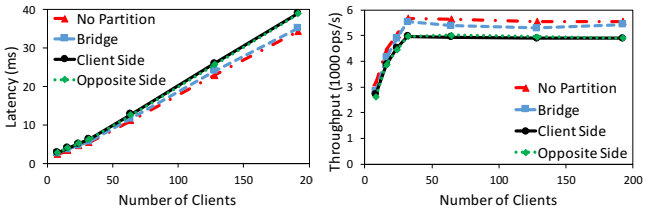
When deploying VoltDB atop Nifty, if the MPI node is on one side of the partition, a potentially significant volume of intermediate data passes through the bridge node. In our setup, when the MPI is on one side of the partition, 50% of the intermediate results are rerouted through the bridge node. This increases operation latency and the load on bridge nodes.

To improve the performance of multi-shard operations, the MPI process can be migrated to a bridge node. This effectively eliminates the need to reroute any traffic for multi-shard queries. We modify VoltDB to use Nifty's API to identify bridge nodes and migrate the MPI to a bridge node.

To evaluate this optimization's effectiveness, we evaluate the effect of the MPI's location on system performance. We restrict clients to contacting VoltDB nodes on one side of the partition and compare the system performance of three MPI placements: on clients side of the partition (client side in Figure 12), on the bridge node (bridge), and on the side opposite to the clients (opposite side). Bridge placement represents our optimization.

Setup and Workload. We use the same VoltDB configuration and partial partition setup detailed in the previouse sections. Unfortunately, VoltDB has limited support for join queries, so it cannot run standard benchmarks such as TPC-H [71]. In our experiments, we use a simple synthetic benchmark that joins two tables. The benchmark has two sharded tables of 20 fields each. Each field is 50 bytes, leading to approximately 1 KB rows. To use multiple shards, clients issue a range query that joins the two tables on the primary key. The client issues a query with a range that includes four primary keys. Consequently, the query result size is limited to four rows, with a total size of almost 8 KB. We populate the database with 20 GB of data before running the experiments. We report the average and standard deviation for 30 runs.

Results. Figure 12 shows the system throughput (a) and the average latency (b) for the three possible MPI placements. During a partial partition fault, placing the MPI on a bridge node decreases the latency by up to 11% and improves throughput by 11% compared to client and opposite side placements. Placing the MPI on a bridge node reduces the number of hops the join query must make before the MPI accumulates all the results and sends the query reply. Furthermore, bridge placement achieves throughput and latency within 4% of VoltDB's performance when there is no partition ("no partition" in Figure 12).



(a) Latency (b) Throughput

Figure 12: The impact of MPI placement on VoltDB's performance. Figure shows the average latency (a) and average throughput (b). Standard deviation was less than 2%.

We measure the amount of data forwarded through the bridge nodes for each one of those configurations; placing the MPI on the bridge node imposes the least overhead. When using 128 clients, 72 MB, 5 GB, and 6.5 GB of data are forwarded through the bridge node when the MPI is placed on the bridge, client side, and opposite side, respectively. The opposite side rerouts more data than the client side placement, as the client request and the result are also rerouted through the bridge node.

8 Related Work

To the best of our knowledge, this is the first study to focus on partial network partitioning, characterize its failures, dissect modern fault tolerance techniques, and explore the design of a generic fault tolerance technique for this type of fault.

A number of previous efforts analyzed failures in distributed systems, including characterizing specific component failures [5, 6, 72, 73, 74, 75] and characterizing failures in a specific domain such as HPC [76, 77, 78], IaaS clouds [79], data-mining services [80], hosting services [8, 81], data-intensive systems [82, 83, 84], and cloud systems [85]. Our work complements these efforts by focusing on failures triggered by partial network partitions.

In our previous work [12], we studied 136 network partitioning failures focusing on complete partitions. This previous work identified partial partitions, presented examples of how they can lead to system failures, and presented NEAT, a testing tool that can inject complete and partial network partitioning faults. We use NEAT to reproduce some of the reported failures. This paper presents an in-depth analysis of partial partition failures and fault tolerance techniques and proposes a novel fault-tolerant communication layer.

Comparing the characteristics of partial and complete partitions [12] shows that they have similar catastrophic impact and manifestation and reproducibility characteristics. Partial partitions seem easier to manifest. While all partial partition failures are triggered by a single-node partial partition and almost all of the failures are deterministic, 88% of the complete partitions manifest by isolating a single node and 80% of them are deterministic. Furthermore, we found twice as many failure reports reporting complete partitions than partial partitions.

Despite their similarity in causing catastrophic failures and being easy-to-manifest, partial and complete partitions are fundamentally different faults. Unlike complete partitions, a cluster suffering a partial partition is still connected but not all-to-all connected. Consequently, the CAP theorem bounds [13] do not apply to partial partitions. Furthermore, fault tolerance techniques for complete partitions cannot handle partial partitions or lead to pausing up to half of the cluster nodes. For instance, using majority vote to elect a leader is an effective mechanism to tolerate complete partitions. This approach alone is not effective in handling partial partitions, as there could be multiple completely connected subgroups with each connecting a majority of nodes. Section 5 shows how using only majority voting can lead to leader election thrashing and system unavailability.

Software-defined networking capabilities have been used to engineer traffic and optimize system operations, including offering network virtualization [86]; building network overlays [87]; performing network measurements [88, 89]; and implementing in-network firewalls [90], load balancers [91, 92], and key-value-based routing [93, 94]. Nifty is similar in spirit to these systems, as we use Open vSwitch capabilities to implement an overlay to mask partial partitions.

9 Concluding Remarks

Our work sheds light on a peculiar type of infrastructure fault and highlights the need for further research to understand such faults and explore techniques to improve systems' resiliency.

This is the first work to focus on partial network partitioning fault and present an in-depth analysis of system failures triggered by this fault. We identify characteristics that can facilitate better test design. Our findings highlight that focused design reviews can identify vulnerabilities early in the design process. We dissect the implementation of eight popular systems and study their fault tolerance techniques. In doing so, we identify four main approaches for tolerating partial partitions. Unfortunately, all implemented fault tolerance techniques have severe shortcomings.

We, therefore, build Nifty to overcome the limitations of modern fault tolerance techniques. Nifty is a simple, transparent communication layer that reroutes packets around partial partitions. We note that modern systems already incorporate a membership and connectivity monitoring. We show that extending the current implementations with a detour mechanism is an effective and low overhead fault tolerance technique to partial partitions. The source code for Nifty is available at https://github.com/UWASL/NIFTY

Acknowledgment

We thank the anonymous reviewers, our shepherd, Jason Flinn, Omid Abari, Ali Mashtizadeh, and Khuzaima Daudjee for their insightful feedback. We thank the artifact evaluation committee members for their effort in evaluating Nifty and for their feedback. We thank Joslin Goh for her feedback on our probabilistic analysis of VoltDB's failure probability. This research was supported by an NSERC Discovery grant, Canada Foundation for Innovation (CFI) grant, and a Waterloo-Huawei Joint Innovation lab grant.

References

[1] Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, Alex C Snoeren, Daniel Turner, Kirill Levchenko, Jeffrey C Mogul, Stefan Savage, and Alex C Snoeren. On failure in managed enterprise networks. *HP Labs HPL-2012-101*, 2012.

[2] Data center: Load balancing data center, solutions reference nework design. Technical report, Cisco Systems, Inc., 2004.

[3] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 58–72. ACM, 2016.

[4] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 3–14. ACM, 2013.

[5] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: measurement, analysis, and implications. *ACM SIGCOMM Computer Communication Review*, 41(4):350–361, 2011.

[6] Daniel Turner, Kirill Levchenko, Alex C Snoeren, and Stefan Savage. California fault lines: understanding the causes and impact of network failures. *ACM SIGCOMM Computer Communication Review*, 41(4):315–326, 2011.

[7] Eric A Brewer. Lessons from giant-scale services. *IEEE Internet computing*, 5(4):46–55, 2001.

[8] David Oppenheimer, Archana Ganapathi, and David A Patterson. Why do internet services fail, and what can be done about it? In *USENIX symposium on internet technologies and systems*, volume 67. Seattle, WA, 2003.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. TAO: Facebook's distributed data store for the social graph. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 49–60, 2013.

[10] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[11] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[12] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 51–68, 2018.

[13] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, June 2002.

[14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.

[15] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[16] Douglas B Terry, Marvin M Theimer, Karin Petersen, Alan J Demers, Mike J Spreitzer, and Carl H Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, volume 95, pages 172–182, 1995.

[17] Barbara Liskov and James Cowling. Viewstamped replication revisited. Technical Report MIT-CSAIL-TR-2012-021, MIT, July 2012.

[18] Rabbitmq message broker. https://www.rabbitmq.com. Accessed: Apr. 2020.

[19] Voltdb in-memory database platform. https://www.voltdb.com/. Accessed: Apr. 2020.

[20] The ceph object store. https://ceph.io/. Accessed: Apr. 2020.

[21] Robin J. Wilson. *Introduction to Graph Theory*. Prentice Hall/Pearson, New York, 2010.

[22] bnx2 cards intermittantly going offline. https://www.spinics.net/lists/netdev/msg152880.html. Accessed: Apr. 2020.

[23] Simon J Maple and Ian Robinson. Transaction recovery in a transaction processing computer system employing multiple transaction managers, October 20 2015. US Patent 9,165,025.

[24] Christian Maihofer. A survey of geocast routing protocols. *IEEE Communications Surveys & Tutorials*, 6(2):32–42, 2004.

[25] Matthew Milano and Andrew C Myers. Mixt: a language for mixing consistency in geodistributed transactions. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 226–241. ACM, 2018.

[26] Observability in paxos clusters. https://davecturner.github.io/2017/08/18/observability-in-paxos.html. Accessed: Apr. 2020.

[27] Partial network partitions and obstacles to innovation. https://rachelbythebay.com/w/2012/02/16/partition/. Accessed:Apr. 2020.

[28] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. Accessed: Apr. 2020.

[29] Healthchecking is not transitive. https://www.robustperception.io/healthchecking-is-not-transitive. Accessed: Apr. 2020.

[30] Cluster broken after switches upgrade. https://github.com/elastic/elasticsearch/issues/9495. Accessed: Apr. 2020.

[31] Using map output fetch failures to blacklist nodes is problematic. https://issues.apache.org/jira/browse/MAPREDUCE-1800. Accessed: Apr. 2020.

[32] Elasticsearch: Distributed search & analytics. https://www.elastic.co/products/elasticsearch. Accessed: Apr. 2020.

[33] Mongodb: The database for modern applications. https://www.mongodb.com/. Accessed: Apr. 2020.

[34] The apache hadoop project. http://hadoop.apache.org/. Accessed: Apr. 2020.

[35] Apache hbase. https://hbase.apache.org/. Accessed: Apr. 2020.

[36] Apache mesos. http://mesos.apache.org/. Accessed: Apr. 2020.

[37] Moosefs: Distributed file system. https://moosefs.com/. Accessed: Apr. 2020.

[38] Kafka: A distributed streaming platform. https://kafka.apache.org/. Accessed: Apr. 2020.

[39] Activemq: Flexible & powerful open source multi-protocol messaging. http://activemq.apache.org/. Accessed: Apr. 2020.

[40] Dkron: A distributed cron service. https://dkron.io/. Accessed: Apr. 2020.

[41] Robert V. Hogg, Elliot Tanis, and Dale Zimmerman. *Probability and Statistical Inference*. Pearson, 9 edition, 2013.

[42] Possible data loss when rs goes into gc pause while rolling hlog. https://issues.apache.org/jira/browse/HBASE-2312. Accessed: Apr. 2020.

[43] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010.

[44] Activemq cluster blocks indefinitely in the presence of partial network partition. https://issues.apache.org/jira/browse/AMQ-7064. Accessed: Apr. 2020.

[45] Arbiters in pv1 should vote no in elections if they can see a healthy primary of equal or greater priority to the candidate. https://jira.mongodb.org/browse/SERVER-27125. Accessed: Apr. 2020.

[46] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. Accessed: Apr. 2020.

[47] minimum_master_nodes does not prevent split-brain if splits are intersecting. https://github.com/elastic/elasticsearch/issues/2488. Accessed:Apr. 2020.

[48] Asymmetrical network partition can cause the election of two primary nodes. https://jira.mongodb.org/browse/SERVER-9730. Accessed: Apr. 2020.

[49] Mirrored queue crash with out of sync acks. https://github.com/rabbitmq/rabbitmq-server/issues/749. Accessed: Apr. 2020.

[50] A network partition can cause in flight documents to be lost. https://github.com/elastic/elasticsearch/issues/7572. Accessed: Apr. 2020.

[51] Hazelcast: the leading in-memory data grid. https://hazelcast.com/. Accessed: Apr. 2020.

[52] Redis: in-memory data structure store. https://redis.io/. Accessed: Apr. 2020.

[53] A. Herr. Veritas cluster server 6.2 I/O fencing deployment considerations. Technical report, Veritas Technologies, 2016.

[54] Balancer can cause cascading mongod failures during network partitions. https://jira.mongodb.org/browse/SERVER-19550. Accessed: Apr. 2020.

[55] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[56] Logcabin. https://github.com/logcabin/logcabin. Accessed: Apr. 2020.

[57] How does voltdb handle partial network partitions? https://www.voltdb.com/resources/transaction-consistency-faq#net. Accessed: Apr. 2020.

[58] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[59] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Understanding partial network partitioning. Technical Report WASL-TR-2020-02, Waterloo Advanced Systems Lab, University of Waterloo, October 2020.

[60] Partial network partitioning leads to cluster unavailability. https://github.com/elastic/elasticsearch/issues/43183. Accessed: Apr. 2020.

[61] Faulty recovery caused by partial network partitions. https://github.com/elastic/elasticsearch/pull/8720. Accessed: Apr. 2020.

[62] Mapreduce ticket 4832. https://issues.apache.org/jira/browse/MAPREDUCE-4832. Accessed: Apr. 2020.

[63] Designing highly available mesos frameworks. http://mesos.apache.org/documentation/latest/high-availability-framework-guide/. Accessed: Apr. 2020.

[64] Wait on shard failures. https://github.com/elastic/elasticsearch/issues/14252. Accessed: Apr. 2020.

[65] Deep Medhi and Karthik Ramasamy. *Network routing: algorithms, protocols, and architectures*. Morgan Kaufmann, 2017.

[66] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. *Data networks*, volume 2. Prentice-Hall International New Jersey, 1992.

[67] Openflow switch specification, version 1.5.1 (onf ts-025). Open Networking Foundation, 2015.

[68] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The design

and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, 2015.

[69] iperf: The ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/. Accessed: Apr. 2020.

[70] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[71] TPC-H benchmark (decision support) standard specification. Transaction Processing Performance Council, December 2018. Revision 2.18.0.

[72] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 193–204. ACM, 2010.

[73] Robert Birke, Ioana Giurgiu, Lydia Y Chen, Dorothea Wiesmann, and Ton Engbersen. Failure analysis of virtual and physical machines: patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 1–12. IEEE, 2014.

[74] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. 2010.

[75] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)*, 4(3):7, 2008.

[76] Nosayba El-Sayed and Bianca Schroeder. Reading between the lines of failure logs: Understanding how hpc systems fail. In *2013 43rd annual IEEE/IFIP international conference on dependable systems and networks (DSN)*, pages 1–12. IEEE, 2013.

[77] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*, pages 425–434. IEEE, 2006.

[78] Bianca Schroeder and Garth Gibson. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing*, 7(4):337–350, 2009.

[79] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh. A first look at problems in the cloud. *HotCloud*, 10:15, 2010.

[80] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 17–26. IEEE Press, 2015.

[81] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16. ACM, 2016.

[82] Ariel Rabkin and Randy Howard Katz. How hadoop clusters break. *IEEE software*, 30(4):88–94, 2012.

[83] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[84] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 963–972. IEEE Press, 2013.

[85] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed data-intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, 2014.

[86] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, et al. Andromeda: performance, isolation, and velocity at scale in cloud network virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 373–387, 2018.

[87] Piyush Raman Srivastava and Saket Saurav. Networking agent for overlay l2 routing and overlay to underlay external networks l3 routing using openflow and open vswitch. In *2015 17th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 291–296. IEEE, 2015.

[88] An Wang, Yang Guo, Songqing Chen, Fang Hao, TV Lakshman, Doug Montgomery, and Kotikalapudi Sriram. vprom: Vswitch enhanced programmable measurement in sdn. In *2017 IEEE 25th International Conference on Network Protocols (ICNP)*, pages 1–10. IEEE, 2017.

[89] Zili Zha, An Wang, Yang Guo, Doug Montgomery, and Songqing Chen. Instrumenting open vswitch with monitoring capabilities: designs and challenges. In *Proceedings of the Symposium on SDN Research*, page 16. ACM, 2018.

[90] Pakapol Krongbaramee and Yuthapong Somchit. Implementation of sdn stateful firewall on data plane using open vswitch. In *2018 15th International Joint Conference on Computer Science and Software Engineering (JCSSE)*, pages 1–5. IEEE, 2018.

[91] Anat Bremler-Barr, David Hay, Idan Moyal, and Liron Schiff. Load balancing memcached traffic using software defined networking. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 1–9. IEEE, 2017.

[92] Alex FR Trajano and Marcial P Fernandez. Two-phase load balancing of in-memory key-value storages through nfv and sdn. In *2015 IEEE Symposium on Computers and Communication (ISCC)*, pages 409–414. IEEE, 2015.

[93] I. Kettaneh, A. Alquraan, H. Takruri, S. Yang, A. S. Dusseau, R. Arpaci-Dusseau, and S. Al-Kiswany. The network-integrated storage system. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[94] Xiaozhou Li, Raghav Sethi, Michael Kaminsky, David G Andersen, and Michael J Freedman. Be fast, cheap and in control with switchkv. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 31–44, 2016.