An Analysis of Partial Network Partitioning Failures in Modern Distributed Systems

by

Mohammed Alfatafta

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

We present a comprehensive study of system failures from 12 popular systems caused by a peculiar type of network partitioning faults: partial partitions. Partial partitions isolate a set of nodes from some, but not all, nodes in the cluster. Our study reveals the studied failures are catastrophic; they lead to data loss, complete system unavailability, or stale and dirty reads. Furthermore, our study reveals that these failures are easy to manifest, they are deterministic, they can be triggered by isolating a single node, and without any interaction with the system's clients.

We dissected the implemented fault tolerance techniques in eight popular systems. We identified four principled approaches for building a fault tolerance mechanism for partial partitions and identified the shortcomings of the current approaches. The currently implemented fault tolerance techniques are either specific to a particular protocol or implementation or may lead to a complete cluster shut down despite the availability of alternative network paths between the nodes.

Finally, we present NIFTY, a generic communication layer that leverages the capabilities of modern software-defined networking to monitor and recover the connectivity of the cluster in case of partial network partitions. NIFTY is transparent to the application running on top of it. We built NiftyDB, a database system atop NIFTY. NiftyDB implements a set of optimizations that reduce the network overhead and operation latency in case of partial network partitioning. Our analysis and evaluation show that the proposed approach can effectively mask partial network partitioning faults without incurring additional overheads.

Acknowledgements

I would like to thank my supervisor, Samer Al-Kiswany for all the valuable advice and guidance he provided throughout my Master's journey.

To my close friends and office-mates Ibrahim and Ahmed, thank you for the numerous ideas and snacks we shared and for keep coming to the office despite my occasional awful singing. Thank you Hatim and Zuhair for your support and for being great friends and colleagues.

I'm thankful for having a supportive family who has always been there whenever I needed them. To my mom, Suhaila, thank you for your unconditional love and caring. My dad Arafat, thank you for your support and for pushing me to do my best. My brothers and sisters, thank you for being so close to me despite being so far away in distance.

I would like to thank my thesis readers, Professor Ali Mashtizadeh and Professor Khuzaima Daudjee for their valuable feedback and comments.

Finally, I'm very grateful for the many great people I met and the numerous friends I made in the past two years. Thank you for making the Master's program a little bit more fun.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Distributed data management systems are the backbone of a wide range of modern applications and are expected to be highly available [1, 2] and to preserve the data stored in them despite failures of devices, machines, networks, or even entire data centers [3, 4, 5]. Among these infrastructure failures, network partitioning is the most complex to handle[6, 7, 8, 9]. Network-partitioning fault tolerance pervades the design of all system layers, from the communication middle-ware to data replication [6, 7, 9, 10] to API definition and semantics[11, 12], and it dictates the availability and consistency levels a system can achieve[13].

The recent increase in network complexity [14, 15] and the increased softwarization of network components [16, 17], contribute not only to the frequency of network partitioning failures (recent studies[18, 19, 20, 21] report that they occur as frequently as once a week and take hours to repair), but also to the introduction of a peculiar type of network partitioning faults [22]: *partial partitions*[1]. Partial partitions isolate a set of nodes from some, but not all, nodes in the cluster. Figure 1.1 illustrates how a partial network partition divides a cluster into three groups such that two groups (Group 1 and Group 2) are disconnected while Group 3 can communicate with both Group1 and Group 2.

Our goal in this work is threefold: First, to characterize partial network partitioning to understand the impact of these failures, to understand the specific sequence of events that lead to them, and foremost, to identify opportunities for improving systems' resiliency to these faults. Second, to understand the fault-tolerance techniques implemented in popular production-quality systems to tolerate partial network partitioning faults, and to identify

[1]This is commonly used name by practitioners in blogs and failure reports.

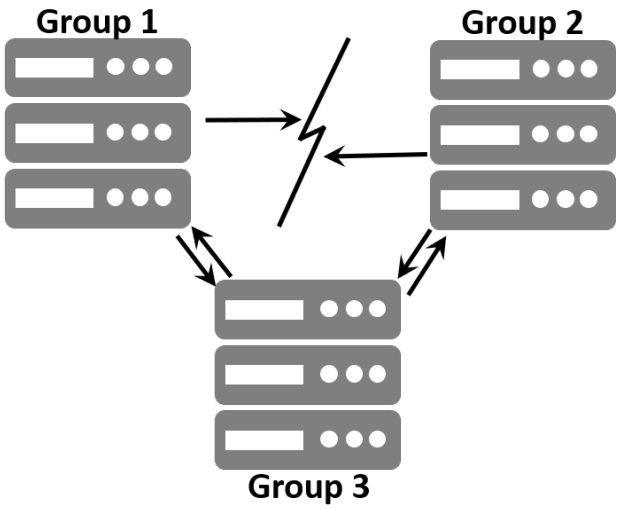


Figure 1.1: Partial network partitioning. Group 1 and Group 2 are disconnected, while Group 3 can communicate with both sides of the partial partition.

their shortcomings. Third, is to design a generic fault tolerance technique for partial network partitions.

While we found 50 reports in 12 production systems' of failures[2] caused by partial network partitioning (Chapter 3), numerous blog posts and discussions of this fault on developers' forums (Chapter 2), and fault tolerance techniques implemented in production systems that are specifically designed to tolerate this type of fault (Chapter 4), we did not find any mention of this type of fault or its fault tolerance techniques in the literature. This is the first work to characterize and design a generic fault tolerance technique for this type of fault.

An analysis of partial network partitioning failures. To characterize this type of fault, we conducted a thorough study of 50 partial network partitioning failures from 12 widely used systems (Table 3.1). For each considered failure, we carefully studied the failure report, logs, discussions between users and developers, source code, code patch, and unit tests.

Failure Impact. Overall, we found that partial network partitioning faults lead to silent catastrophic failures (e.g., data loss, data corruption, and data unavailability), with 24% of failures leaving the system in a lasting erroneous state that persists even after the partition heals.

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

Ease of manifestation. Oddly, it is easy for these failures to occur. The majority of the failures were deterministic and required three or fewer frequently used events (e.g., read and write), and all the failures can be triggered by partially partitioning a single node.

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. Surprisingly, our analysis of the manifestation sequence of each failure, ordering constraints, and timing constraints, shows that almost all the failures can be reproduced through tests and by using only five nodes.

Dissecting the current fault tolerance techniques. We studied the fault tolerance techniques implemented in eight popular systems (Chapter 4), including VoltDB, MapReduce, HBase, MongoDB, ElasticSearch, Mesos, Raft, and RabbitMQ. For each implemented technique we analyzed the code, extracted the design principles, and identified the design shortcomings. We identified four approaches for tolerating partial network partitioning: connectivity monitoring and graph-based recovery, checking neighbours' view of the network, verifying failure reports announced by other nodes, and naturalizing partially partitioned nodes.

Our analysis reveals that all current fault tolerance techniques are either not generic (i.e., are used to patch a specific system mechanism or protocol), or do not adequately tolerate partial partitions (Chapter 4). For instance, all current generic fault tolerance techniques can lead to complete system shutdown or to the loss of up to half of the nodes in the cluster despite the presence of alternative routing paths around the partial partition.

Designing a generic fault tolerance technique. Our findings motivated us to build **n**etwork part**i**tioning **f**ault-**t**olerance s**y**stem (NIFTY). NIFTY leverages the capabilities of software-defined networking (SDN) to build a generic communication layer positioned above the IP layer (Chapter 5). NIFTY monitors the connectivity in a cluster, and when it detects a partial partition, it detours the traffic around the partition through intermediate nodes. NIFTY overcomes all the shortcomings present in modern fault tolerance techniques.

To demonstrate our idea, we built NiftyDB, a VoltDB-based database engine that uses NIFTY to tolerate partial network partitioning (Chapter 6). NiftyDB implements two optimizations to reduce the system overhead, lower operation latency, and reduce the load on intermediate nodes.

Our evaluation (Chapter 8) on a 28-node cluster with synthetic and real workload benchmarks shows that the overhead added by using NIFTY is negligible. Furthermore, we show that NiftyDB optimizations bring tangible performance gains: up to 13.8% higher

throughput, 12.9% lower latency, and two orders of magnitude lesser load on intermediate nodes compared to a vanilla implementation of VoltDB over NIFTY.

Chapter 2

Causes of Partial Network Partitioning

Modern networks are complex. They span multiple data centers [14, 23], use heterogeneous hardware and software [20], and employ a wide range of middle boxes (e.g., NAT, load balancers, route aggregators, and firewalls) [14, 18, 23]. Despite the high redundancy built into modern networks, catastrophic failures are common [18, 19, 20, 21].

Recent reports indicate that network partitioning faults are common and happen at different scales. Network partitioning 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 [20]. Turner et al. found that a network partition occurs almost once every four days in the California-wide CENIC network [21]. In a data center, a network partition can manifest due to failures in the core or aggregation switches [19] or because of a top-of-the-rack (ToR) switch failure. Microsoft and Google report that ToR failures are common and have led to 40 network partitions in two years at Google [18] and caused 70% of the downtime at Microsoft [19]. Furthermore, NIC failures [24] 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 [19, 21, 23]. Correlated switch failures are frequently caused by system-wide upgrades and maintenance tasks [18, 19].

Unlike the classical *complete* network partitions, which split the cluster into two completely disconnected sides, partial network partitions disrupt the communication between some of the cluster nodes. This leads to the division of nodes into three groups (Group 1, Group 2, and Group 3 in Figure 1.1) such that two groups (Group1 and Group 2) are dis-

connected while Group 3 can communicate with both Group1 and Group2. Despite finding 50 failure reports detailing system failures due to partial network partitions, numerous articles and online discussion forums [25, 26, 27, 28] discussing the fault, implementations of fault tolerance techniques in eight popular systems, and unit tests testing for the fault in a popular SDN operating system framework (ONOS [29]), we did not find network failure reports that detailed the root cause of partial network partitioning faults in production networks. A few failure reports and blogs discussed the root cause of the partial partition including the loss of connectivity between two data centers [20] while both are reachable by a third center, the failure of additional links between racks [30, 31], network misconfiguration [32], firewall misconfiguration [32], network upgrades [33], and flaky links between switches [34].

Chapter 3

Analysis of Partial Network Partitioning Failures

We conducted a comprehensive study of partial network partitioning failures reported in 12 production systems (Table 3.1). Our aim is 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.

3.1 Study Methodology

We studied 50 real-world failures from 12 popular distributed systems (Table 3.1). We chose a diverse set of distributed systems to show that partial network partitioning faults do not affect a specific type of distributed system and because these systems are widely used and deployed. Our studied systems include two key-value storage systems and databases, two file systems, an object storage system, three message-queuing systems, a data-processing framework, a search engine, and two resource managers.

The 50 failures included in our study were selected from the publicly accessible issue tracking systems for these projects as follows: First, we used the search tools in the issue-tracking systems to identify tickets related to partial network partitioning. Because most of the users and developers do not classify network partitioning failures when they report them, we had to search for all network partitioning failures and then manually identify the ones related to partial network partitioning failures. To search for the tickets, we used the following keywords: "network partition," "network failure," "switch failure," "network

Table 3.1: List of studied systems, the number of failures, and the number of catastrophic failures. The shaded rows are the systems that implemented a fault tolerance technique for partial network partitioning.

System	Category	Failures	
		Total	Catastrophic
Elasticsearch	Search engine	17	17
MongoDB	Key-value store	9	5
RabbitMQ	Messaging	5	3
MapReduce	Data processing	4	2
HBase	Key-value store	3	2
Mesos	Resource manager	2	1
HDFS	File system	3	1
Ceph	Storage system	2	2
MooseFS	File system	2	2
Kafka	Messaging	1	1
ActiveMQ	Messaging	1	1
DKron	Resource manager	1	1
Total	**-**	**50**	**38**

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 failures and excluded tickets that appeared to be part of the development cycle; for instance, those that discuss a feature design. For each ticket, we studied the failure description, system logs, developers' and users' comments, code patch, and unit tests. For tickets that lacked enough details, we reproduced them using NEAT [22]. Table 3.1 shows the number of failures we studied for each system.

3.2 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 12 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 3.1). 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: Kafka is the most popular message-queuing system; MapReduce, HDFS, and HBase are the core of the dominant Hadoop data analytics platform; and MongoDB is a popular key-value-based database.

2) *Sampling bias.* The way we choose the tickets may bias the results. We designed our methodology to include high-impact tickets. To make sure the tickets we studied are of high impact, 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 reviewed and analyzed using the same detailed classification methodology, then discussed in a group meeting before an agreement was reached.

3.3 Findings

In this section, we present nine findings our study revealed. In general, our study shows that partial network partitioning failures have catastrophic effects and are silent. However, our study also reveals ways to make systems more resilient to partial partitioning. For example, we show which of a system's components are more vulnerable to these failures and we show that most of these failures are deterministic and can be reproduced with testing.

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. 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 a part of the system to carry on normal operations, while another part executes fault tolerance routines. Apparently, the mixing of these two modes is poorly tested.

Table 3.2: Partial network partitioning failures impact. The percentages show how many failures caused the corresponding impact.

Impact	**%**	
Data loss	24%	Catastrophic (76%)
Complete system unavailability	20%	
Stale read	16%	
Data corruption	6%	
Dirty read	4%	
Data unavailability	4%	
Reduced availability	24 %	
Other	2%	

Finding 1: *A significant percentage (76%) of the studied failures have a catastrophic impact.*
A failure is said to be catastrophic if it leads to a system crash or if it violates the system's guarantees. Table 3.2 shows the impact each of the failures we studied had on the system. This finding indicates that partial partitioning failures have grave impacts and should be considered in all stages of system development in order to reduce the chances of their occurrence.

We found that data loss is the most common impact of partial network failures. For example, in a MongoDB cluster consisting of three nodes, whenever there is a partial partitioning that isolates a replica from the primary replica but not from the third replica, the primary would keep operating as it can reach a majority of nodes, and the isolated replica will also start an election as it is connected to a majority and is not connected to the primary [35]. This leads to having two primaries in the cluster. If two different clients are connected to each of the primaries and they update the same key, when the partitioning is healed one of the updates will be lost.

Stale reads are another common effect of partial partitions. For instance, in an Elasticsearch cluster, a partial partition isolating the leader from some nodes in the cluster can lead to electing a new leader while the old one still acts as a leader. If one of the leader processes a write operation while the other process a read operation, the read operation may return stale data [33].

In Elasticsearch, a dirty read happens when there is a partial partition between a shard's primary and one of its replicas, while the master node reaches both. If the primary node receives a write request followed by a read request, the primary then replies with unacknowledged data (as the write request did not reach one of the replicas). If the

primary fails before the partition is healed, and the master chooses the replica on the other side of the partition to become the new primary, then the previous read received by the old primary is a dirty read as that write will never get committed.

In 24% of the failures, a partial partition unnecessarily leads to reduced system availability. For example, MongoDB's 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 [36]. MongoDB does not serve client requests during leader election; consequently, this failure significantly reduces availability.

Finding 2: *Most of the studied failures (84%) are silent – the user is not informed about their occurrence.*
Despite the dangerous effects of partial partitioning faults, most systems do not report to the user that a failure has happened. Furthermore, the systems that send warnings to the user when a failure occurs just send ambiguous warnings that the user cannot make use of to fix the problem. For example, in Elasticsearch, whenever a partial partition isolates a server node from the rest of the cluster (but not from the client), and the client tries to send requests to the cluster he receives a generic error message [37], which does not inform the user of the actual cause of the problem. This usually leads to delayed detection of failures by system administrators, which in turn makes the problem worse.

Finding 3: *Twenty-four percent of failures remain after the partitioning is healed.*
We found that while the majority of failures only persist while there is a partial partitioning in place, a significant number of failures persist even after the partition heals. For example, in MapReduce, if a partial partition isolates the resources manager (RM) and the AppMaster, but not between the AppMaster, shared storage (HDFS), and the client, the RM will spawn a new AppMaster. This leads to running two AppMasters for the same task, which could potentially cause data corruption [38]. Even more alarming is that even after the partitioning is healed, the two AppMasters will continue to work in the same system, which means that healing the partitioning alone would not solve the problem. Similar behaviour is found in other systems.

Finding 4: *Leader election, replication protocol, configuration change, and request routing*

Table 3.3: Percentages of partial partitioning failure by mechanism.

Mechanism	**%**
Leader election	38%
Replication protocol	18%
Configuration change	18%
Request routing	12%
Scheduling	6%
Data migration	6%
Data consolidation	2%

are the mechanisms most vulnerable mechanisms to network partitioning.
We studied the source of the failures that are triggered by partial network partitioning. Table 3.3 presents the percentages of failures related to each of the system mechanisms.

We found that leader election is the mechanism most vulnerable to partial partitioning faults. In most cases, these failures lead to electing two leaders. For example, in both Elasticsearch [39] and MongoDB [35], whenever there is a partial partitioning in which a node cannot communicate with the current master, but can communicate with a majority of nodes, that node will start an election and become a master, even though the previous master is still operational and is connected to a majority of nodes. This failure leads to data loss and stale reads.

We found that the replication protocol mechanism was the second most vulnerable mechanism in systems. This was mostly the case in Elasticsearch and Ceph. For instance, in Elasticsearch (discussed in more detail in Chapter 4), if there is a partial partitioning isolating a shard's primary node from most of that shard's replica nodes and the master node, while the client is connected to all nodes, there is a time window in which the primary does not step down. In that time, the primary will acknowledge update requests to the client without replicating them [40]. If the client then reads from another replica, it may read stale data.

The rest of the failures were caused by flaws in configuration change, request routing, scheduling, data migration or data consolidation protocols.

Finding 5: *Most of the failures (60%) do not require a client to access any of the servers, or require only that a client accesses one side of the partition.*
To mitigate the effects of network partitioning, some systems seek to limit user access to one side of the partition. However, this finding debunks this assumption, as most partial partitioning failures do not require client access to both sides of the partition.

Table 3.4: Number of events required for partial network partitioning failure to manifest.

Number of events	**%**
1 (Just a partial partition)	14%
2	8%
3	32%
4	14%
>4	32%

As an example of a failure that does not require a client access, in MapReduce, if a partial partitioning isolates reducers from some mappers while both can reach the resource manager, the resource manager will blacklist these mappers if they are reported by many reducers event without any new client requests [41].

This finding indicates that system designers must consider the impacts of partial network partitioning faults on all system operations, including asynchronous client operations and offline internal operations.

Finding 6: *The majority of failures (66%) require three or fewer events (other than the partial partitioning) to manifest.*
This finding shows that only a few events need to occur for a failure to happen. This is dangerous, as in a real deployment of a distributed system many users will interact with the system, increasing the probability of failure. Table 3.4 shows the number of events needed to manifest a failure. We found that in 14% of cases, no events other than a single partial partitioning fault needed to happen for a failure to manifest.

This is perilous, as a small number of events (mostly frequently used ones, such as reads and writes) can lead to catastrophic failures.

Finding 7: *All the studied failures manifest by isolating a single node, with 34 % of them happening by isolating any replica.*
We consider a node to be isolated if it happens to be on one of the partial partition sides; that is it cannot communicate with some of the other nodes in the system. It is very worrisome that all of the studied failures can manifest by isolating a single node. Isolating a single node is generally more likely to happen than isolating more than one. This could happen by a single malfunction of a network device such as a NIC or a ToR switch, or by a single misconfigured firewall in one of the machines.

We further studied which nodes needed to be isolated for a failure to manifest (Table 3.5). We found that 34% of the failures manifested by isolating any node in the system

Table 3.5: System connectivity during partial network partitioning.

Network Partition Characteristics		**%**
Partition any replica		34%
Partition a specific node		66%
• Partition the leader	44%	
• Partition a node with a special role	10%	
• Partition a central service	8%	
• Partition a new node	2%	

Table 3.6: Timing constraints of failures.

Timing constraint	**%**
No timing constraints	64%
Known timing constraints	34%
Nondeterministic	2%

regardless of its role. Among the failures that required the isolation of a specific node, we found isolating the leader replica to be the most common (44%). Considering that in most deployments a node plays multiple roles for different shards, isolating a leader is not a rare occasion whenever a partial partitioning occurs. Partitioning a node with a special role (such as an arbiter in MongoDB) caused 10% of the failures.

Finding 8: *All of the studied failures except one are either deterministic or have known time constraints.*
Table 3.6 shows the timing constraints needed for a failure to happen. We noticed that almost all the failures either had no timing constraints at all (i.e., whenever the event sequence happens, a failure happens) or had known timing constraints. These known timing constraints are either hard-coded in the system's code or are configurable by the end users, such as the number of heartbeat periods to wait before declaring that a node has failed. Only one of the failures is non-deterministic, as it involves the interleaving of multiple threads.

Finding 9: *All failures can be reproduced with five nodes, and all but one can be reproduced using a fault injection framework.*
We found that these failures can be easily reproduced with small clusters of five or fewer nodes to manifest (Table 3.7), with 78% of them requiring only three nodes. Furthermore, we found that all the failures except one can be reproduced using a fault injection framework

Table 3.7: Number of nodes needed to reproduce a failure

Number of nodes	**%**
3 nodes	78%
4 nodes	20%
5 nodes	2%

such as NEAT [22] or Jepsen [42].

3.4 Insights

We presented a thorough study of partial network partitioning failures that occurred in well-known production quality distributed systems. Our study revealed surprising findings: It was remarkable that most of the failures we studied had catastrophic impacts on the systems they affected and that the systems produced little to no useful information to users when these failures occur. Our study not only revealed how dangerous partial partitioning failures can be, but it also analyzed each of the failures to understand what leads to their occurrence and which of the different system's mechanisms are more susceptible to these failures. Our findings indicate that the area of partial network partitions is a high-impact research area that needs further research to improve systems' resiliency and fault tolerance.

Although a previous study on network partitioning failures [22] revealed similar characteristics to those of partial network partitioning failures, we found the two to have some differences. For example, all partial network partitioning failures manifested by isolating a single node, compared to 88% of generic network partitioning failures. This highlights the poor understanding and testing of partial partitions in the development cycle.

Most production systems assume unreachable nodes to have failed. Even worse, in the case of partial network partitioning, most of the studied failures are caused by the underlying assumption that if a node can reach a service, all other nodes can reach that service. Our analysis revealed the dangers of such assumptions especially in the presence of a partial partitioning fault; this leads to a confusing system state wherein some parts of the system assume another part is down while others presume the whole system to be healthy.

Our study revealed the dangers of assigning a low priority to ToR switch failures especially as a single node isolation was the cause of all our studied failures. Additionally, our

study shows that system designers need to consider partial partitioning failures through the whole design and development cycle from its early stages.

We identified two approaches for improving systems resiliency to partial network partitioning: testing and rerouting. Our analysis reveals that most of the failures are deterministic, use few common events, and require five nodes or fewer to reproduce them. These characteristics indicate that testing using a framework that can inject network partition faults can reveal most of these bugs. Second, is the feasibility of building a generic fault tolerance technique for partial partitions. During a partial network partition, a partitioned node can still reach some of the nodes in the cluster. One approach to mask this fault is to reroute network packets around the partition using other nodes as proxies. We explore this approach in Chapter 6.

Chapter 4

Study of Current Fault Tolerance Techniques

Our goal is to understand the fault tolerance techniques that are currently implemented in eight production systems to tolerate partial network partitions. We studied the system design of all the systems in Table 3.1 and studied all the code patches related to the bugs we studied. We found that six of the listed systems changed the system design to tolerate partial network partitions, including MongoDB, Elasticsearch, RabbitMQ, HBase, MapReduce, and Mesos. The rest of the systems either patched the code with an implementation-specific workaround or have not fixed the reported bugs yet. In addition to these six systems, we found that VoltDB [43] and Raft [7], although not having failure reports related to partial partitions in their publicly accessible issue tracking systems, implemented fault tolerance techniques to tolerate partial network partitions. We study the fault tolerance techniques implemented in these two systems as well.

Our analysis reveals that modern systems use four main techniques to tolerate partial network partitioning: Cluster-wide connectivity monitoring and building a global connectivity graph, verifying node failure by asking neighbouring nodes, verifying reports of a node failure by checking the suspected node, and neutralizing the partitioned nodes. In the following section, we detail these techniques and discuss their shortcomings.

4.1 Cluster-wide connectivity monitoring

This technique monitors the connectivity of a cluster. When a partial network partition is detected, it constructs a connectivity graph and uses it to inform the decisions of the recov-

ery procedure. VoltDB follows this approach to build a generic fault tolerance technique for partial partitions.

VoltDB is a strictly serializable distributed and replicated relational database system that supports ACID transactions. VoltDB supports both sharding and replication of data to multiple nodes. As our characterization in Chapter 3 shows, tolerating partial network partitions is challenging and requires careful examination and testing of all system operation paths and system modules to avoid the catastrophic effects of partial network partitions, such as data loss and corruption. To avoid these catastrophic failures, VoltDB uses the cluster-wide connectivity monitoring technique to detect a partial partition then shut down all the nodes that are on the minority side of the partition (i.e., the side that has fewer nodes). While this approach may significantly reduce system capacity and may lead to complete system shutdown (as detailed next), it avoids the catastrophic effects of partial network partitions.

VoltDB follows a peer-to-peer approach for implementing this technique. Every node in the system periodically sends a heartbeat to all nodes in the cluster. If a node loses its connectivity to any node in the system, it will suspect that a partial network partition occurred and will start the recovery procedure. The recovery procedure has two phases: In the first phase, the node that detected the failure broadcasts the list of nodes it can directly reach. Upon receiving this message, all nodes in the cluster in their turn will broadcast their connectivity information to all nodes in the cluster. In phase two, every node combines the information received from all other nodes into a connectivity graph for the cluster. Each node independently analyses this graph and uses a deterministic policy to respond to the partial partition. In the current VoltDB implementation, every node will detect the largest fully connected group of nodes in the graph, and each node that is not in this group will shut itself down. To handle the pathological case in which the cluster experiences a complete partition that exactly halves the nodes of the cluster into two equal groups, the group that has the node with the lowest id in the cluster resumes operation, while the group on the other side of the partition shuts down. Finally, the surviving group of nodes will check that they have a replica of every data shard in the new group. If there is shard with no reachable replica then the entire VoltDB cluster shuts down.

The nodes that shut down do not automatically rejoin the cluster even after the network partition heals. To add those nodes back to the cluster an administrator needs to stop the cluster, reconfigure it, and start it again.

Shortcomings. The VoltDB approach has several shortcomings that make it inadequate for modern cloud deployments. First, in the best case, it will unnecessarily shut down up to half of the cluster nodes, reducing the system performance and fault tolerance

ability. Second, the approach may lead to a complete system shut down if all replicas of a shard are on the minority side of the partition or if the cluster is halved into two equal groups and the node will the lowest id crashes. In the latter case, the two halves will assume the node with the lowest id is on the other side of the partition and both will shutdown. Third, the proposed approach is not bulletproof - it takes time until the partial partition is detected and the nodes on the minority side shut down. During this time, new operations may lead to catastrophic failures.

4.2 Checking with neighbours

This approach is used by RabbitMQ and Elasticsearch. In this approach, every node in the system monitors its connectivity to all nodes in the system through periodic heartbeats. If it detects that a certain node (say node X) is unreachable, it will rely on information provided by other nodes to verify if the other nodes can reach X. If other nodes can reach X, then a partial partition is detected and the node will execute a recovery procedure. Note that, unlike the graph-based approach discussed in Section 4.1, this approach does not build a graph and does not run a deterministic recovery procedure to guarantee that all nodes seeing the partial partition execute the same actions.

In RabbitMQ [44], every node periodically sends heartbeats to all other nodes to detect failures and exchange information. Upon the discovery of a partial partition, a node in RabbitMQ can apply one of the following configurable policies:

- **Change to a complete partition**: A node that uses this policy drops its connection with all nodes that can reach the other side of the partition. This changes the partition from a partial to a complete partition with both sides working and accepting client requests. This configuration may lead to data inconsistency and will require running a data consolidation mechanism after the partition is fixed (detailed next).
- **Pause**: To avoid data inconsistency, once a node discovers the partial network partition it pauses its activities. It resumes its activities only when the partition heals.
- **Pause if anchor nodes are not reachable**: RabbitMQ configuration can specify a subset of nodes to act as anchor nodes. If a node can not reach any of the anchor nodes, it will pause. This approach may not solve a partial partition problem and may lead to multiple complete partitions. Consequently, this policy requires running a data consolidation mechanism when a partition heals.

After a partition heals, RabbitMQ provides two data consolidation policies: administrator intervention, in which the administrator decides which side of the partition should become the authoritative copy of data, and auto-heal, in which the system decides on the winning version of the data based on the number of clients and nodes connected to each partition.

RabbitMQ's approach has two serious shortcomings. First, changing a partial partition to a complete partition may lead to multiple divergent copies of the data that are hard to consolidate. Second, the pause policy may lead to pausing all the nodes on both sides of the partition severely affecting data availability and system performance. In the worst case, it may bring the entire cluster down.

Elasticsearch [45] has a single master that is responsible for cluster-wide operations. The master is deterministically assigned to the node with the smallest id. If a node cannot reach the master, then it contacts all other nodes it can reach and asks them if they can reach the master node, if any node can reach a master then a partial network partition is detected, and the affected nodes will keep trying to reach the master. If none of the nodes can reach the master, then a leader election protocol is executed to elect a new master (i.e., the node with the smallest id among the reachable nodes).

This approach can lead to a complete cluster unavailability [46]; for instance, if none the nodes can reach the master except one node (say node X). If node X has the smallest id among the connected nodes, then all the nodes in the cluster will assume it is the new master, but node X will refuse to be a new master since it can reach the old master. Effectively, the cluster will not have a master and will be unavailable until the network partition is healed [46].

4.3 Failure verification

Using this approach, if a node receives a notification that a certain node is unreachable, the receiving node will try to contact that specified node and verify that it is indeed unreachable before applying the fault tolerance technique. This approach is not generic and is often applied as a part of a particular protocol. This approach is used by MongoDB [47] and Raft [7] as part of their leader election protocols. Upon discovering that the leader is unreachable, a node will call for an election and ask all other nodes to vote for itself. Other nodes only participate in the election if they verify that the leader is unreachable to them as well. If another node in the cluster can reach the leader, then it ignores the call for the election. If a partial network partition isolates a minority of nodes from the leader,

then this approach prevents electing a new leader and avoids the leader election thrashing failure discussed in Section 3.3.

This failure verification is not generic; for instance, in ElasticSearch if a partial partition isolates a primary replica from a secondary replica while both replicas are reachable by the cluster master node, then the primary replica will notify the master node that a replica is unreachable and request an alternative replica. As the master can reach all the replicas, it will ignore the failure report, rendering the partition unavailable for future writes if the cluster is configured with synchronous replication to all replicas [48]. This bug was fixed by assigning new replicas whenever a primary cannot reach one of the old replicas.

4.4 Leader neutralizes partitioned nodes

One challenge of handling partial network partitions is that both sides of the partition may change the shared data. To avoid this problem in leader-based systems, this approach tries to neutralize all the nodes that are not reachable by the leader. How a node is neutralized is specific to the application and implementation.

In Mesos, whenever a node stops receiving heartbeat messages from the master for some time, it pauses and tries to contact the master to rejoin the cluster. Effectively, in case of partial network partitions, the node will pause until the network partition heals.

In HBase, shards are stored on a shared storage system (HDFS) and managed by HBase nodes. If a leader cannot reach one of the HBase nodes, then before assigning the shards of that node to another node, the master will rename the shard directory in HDFS, effectively neutralizing the old replica from making any further changes. If an HBase node can access its shard, then it shuts down.

In MapReduce, if the manager node cannot reach one of the AppMaster nodes, it will reschedule the tasks assigned to that AppMaster to a new AppMaster. This approach introduces the possibility that two AppMasters will work on the same task and lead to data corruption [38] in the shared HDFS file. To avoid this problem, AppMasters use two approaches to indicate that they completed a task: they inform the manager, and write a completion record in a shared log on HDFS. When a manager starts a new AppMaster to re-execute a task, the new AppMaster first checks the shared log for a completion record. If it finds a record then the task is completed and it does not re-execute the task.

Summary

None of the implemented fault tolerance techniques is adequate for modern cloud systems. Failure verification and neutralizing partitioned nodes are protocol- and implementation-specific, and the current approach for checking with neighbours and the graph-based technique may lead to complete system shutdown or significant loss of system performance or storage capacity. This motivated us to explore a generic fault tolerance technique for partial partitions that helps tolerate these failures without significant changes to current distributed systems (Chapter 5).

Chapter 5

System Design

To overcome the limitations of current fault tolerance techniques, we designed a network-partitioning fault-tolerant communication layer (NIFTY). NIFTY leverages the capabilities of software-defined networking (SDN) to build a generic communication layer positioned above the IP layer. NIFTY monitors the connectivity in a cluster; when NIFTY detects a partial partition, it detours the traffic around the partition through intermediate nodes. While NIFTY maintains the connectivity between the nodes in case of partial partitions, it may impose high network overhead. We present two optimization techniques to reduce the network overhead (detailed in the next chapter). We designed NiftyDB, a VoltDB-based database system that can tolerate partial network partitions. NiftyDB changes the fault tolerance technique in VoltDB to use NIFTY and implements the two optimizations.

In the rest of this chapter, we present an overview of the SDN technology in Section 5.1, then present the NIFTY design in Section 5.2. In the following chapter, we present the design of NiftyDB.

5.1 Overview of Software-Defined Networking

The SDN architecture divides the network into two planes: data and control. The data plane is a traffic forwarding plane that uses the information in the switch forwarding tables to forward messages. The control plane is an external process that controls the switch by altering the entries in switch forwarding tables. The communication API between the controller and the switches is based on the widely-adopted OpenFlow standard [16].

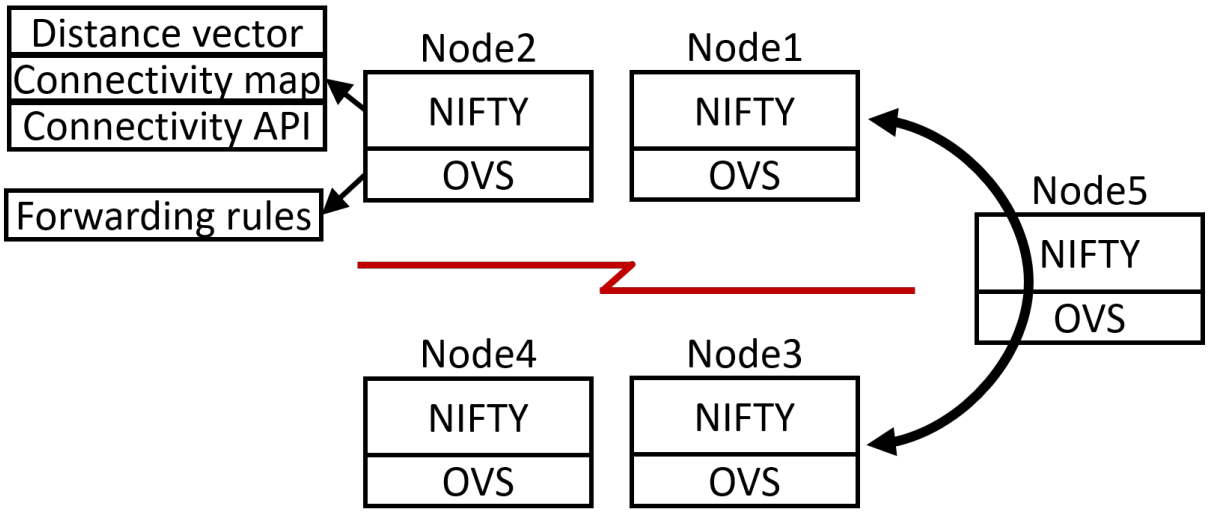


Figure 5.1: NIFTY design. A partial network partition isolates node 1 and 2 from node 3 and 4 while node 5 can communicate with all. Node 1 and node 2 install forwarding rules that forward traffic destined to node 3 or node 4 through node 5.

The OpenFlow standard [16] facilitates external control of single-switch forwarding tables. It allows inserting or deleting forwarding rules. Each forwarding entry includes a matching rule and an action list. If a packet matches a rule, the actions in the actions list are performed in order on the packet. OpenFlow has a rich set of matching rules including wild cards for matching IP and MAC addresses, protocol or port numbers. The actions include packet forwarding to a specific switch port, dropping the packet, sending the packet to the controller, or modifying the packet. The possible modifications include changing the source/destination MAC/IP addresses. OpenFlow controllers can update, delete, or extend the validity of the existing rules at any time. These capabilities enable fine-grained control of network operations and facilitate application-specialized traffic engineering.

5.2 NIFTY Design

NIFTY is a peer-to-peer system in which every machine in the cluster runs a NIFTY process, and all nodes have the same role. NIFTY processes collaborate in monitoring cluster connectivity, recovering from network partition faults, and classifying the cluster nodes based on their connectivity. To mask a network partition, NIFTY tries to reroute packets around the partial partition through the end nodes. For instance, in Figure 5.1, a

partial network partition isolates node 1 and 2 from node 3 and 4. Node 5 is still able to reach all nodes. In this scenario, once NIFTY detects the partial partition, it will reroute packets exchanged between nodes 1 and node 3 through node 5; that is the packets between 1 and 3 will be sent to 5, which will act as a router and forward the packets to the other side of the partition.

Connectivity monitor. To monitor the end-to-end connectivity between cluster nodes, each NIFTY process monitors its connectivity with every other NIFTY process in the cluster by periodically sending a heartbeat to all processes in the cluster and maintaining a connectivity bitmap. The connectivity bitmap is a compact bitmap that indicates which nodes are directly reachable. If a NIFTY process misses three heartbeats from another NIFTY process, it assumes that the communication with that process is broken and updates its connectivity bitmap. To detect when the communication between nodes recovers, NIFTY processes continue to send heartbeats to disconnected nodes even after a partition is detected.

Recovery. Every NIFTY process maintains a connectivity graph that tracks the connectivity between all the nodes in a cluster. To build and maintain this graph, each NIFTY process periodically sends its connectivity bitmap to all other nodes. To reduce overhead, the bitmap is piggybacked on heartbeat messages.

When a NIFTY process detects a change in the connectivity graph (e.g., a node becomes unreachable or a partition has healed and connectivity is restored) it initiates the route discovery procedure. The route discovery procedure uses the connectivity graph to find alternative routes to the unreachable nodes or restores direct routes after a network partition is healed. To find the best communication routes, NIFTY can use standard network routing protocols. In our design, we used the classical distance-vector protocols using the Bellman-Ford algorithm [49, 50], as our characterization revealed that most network partitions involve a single partition and require simple rerouting paths. We use hop count as route weight. Using hop counts as a routing metric naturally favours direct connections, when they exist, over rerouting through intermediate nodes.

Route deployment. To deploy the new routes between end nodes, NIFTY leverages the capabilities of software-defined networking. NIFTY uses OpenFlow and Open VSwitch to route packets between end nodes. For instance, to reroute packets sent by node 1 to node 3 through node 5 in Figure 5.1, the NIFTY process on node 1 will install rules on its local Open VSwitch to change the destination MAC address of any packet destined to node 3 to that of node 5. Whenever node 5 receives a packet that is destined to node 3, it changes the destination MAC address back to node 3's MAC address, and sends the packet out where it will finally be received by the final target (node 3).

Listing 5.1: Retrieving the list of bridge nodes in the system and checking if the current node is a bridge node or not

```
vector<string> bridge_nodes = DV::getBridgeNodes();
if(find (bridge_nodes.begin(), bridge_nodes.end(), my_ip)
        != bridge_nodes.end())
    cout<<"This is a bridge node";
    ...
else
    cout<<"This is not a bridge node";
    ...
```

Node classification. Each NIFTY process analyzes the connectivity graph to identify which nodes are on the same side of the network partition (e.g., nodes 1 and 2 in Figure 5.1) and which nodes are not affected by the partition and can reach all cluster nodes. We call these nodes "bridge" nodes (e.g., node 5 in Figure 5.1). The list of cluster nodes and their classification is provided through an API to the database engine running atop NIFTY. Listing 5.1 shows the API for retrieving the list of bridge nodes and checking if the current node is a bridge node or not.

Chapter 6

NiftyDB

NIFTY is transparent to applications running on top of it, and no application changes are required to use NIFTY. As NIFTY routes packets through intermediate nodes, it may increase operation latency as packets need to traverse longer paths, increase the load on bridge nodes, and increase the network load. To lessen this overhead, a system using NIFTY can be optimized to reduce the amount of data forwarded through the bridge nodes. The way to do so is application-specific and may entail relocating processes in a cluster, reducing quality of service [51, 52, 41], or reducing harvest [1].

In this thesis, we explore optimizations that can improve the performance of distributed database systems under partial network partition faults. We built NiftyDB, a database system that extends integrated VoltDB with NIFTY and implements optimizations to reduce the traffic on bridge nodes. We identified two opportunities to reduce the load on bridge nodes (i.e., nodes that can reach all nodes in the cluster and are used to detour traffic around partial partitions). The rest of this chapter first presents an overview of VoltDB, then details the two optimizations.

6.1 VoltDB Design Overview

VoltDB is a strictly serializable distributed and replicated relational database system that supports ACID transactions. VoltDB allows the sharding of tables based on a specific column. Sharding is then automatically handled by VoltDB (the number of shards is not specified by the user). The user specifies the replication factor (known as k-factor) for the cluster. The k-factor specifies how many times each of the shards is replicated, and how

many simultaneous node failures the cluster can tolerate. More specifically, each of the shards is replicated (k+1) times, and the cluster is guaranteed to operate correctly under the presence of a maximum of k simultaneous node failures.

VoltDB replicates the received queries to all shard replicas and then will have the replicas apply the queries locally. To achieve a strict order of the queries across all replicas, VoltDB employs two entities: SPIs and an MPI. An SPI (single partition initiator) is responsible for ordering queries that target a single shard. All queries are directed to the SPI at first. The SPI then replicates the queries to all shard replicas and makes sure that all surviving replicas have executed the queries before marking them as committed. An MPI (multi-partition initiator) is responsible for ordering queries that target multiple shards. It achieves that by dividing a query into multiple segments, each targeting a different shard. It then forwards these to the specific SPI responsible for each shard. When a multi-partition read operation is issued (say a join request), the MPI forwards segments of the request to each involved SPI. It then gathers the data from the different SPIs before sending the result of the request to the VoltDB node the client had contacted to issue the request.

6.2 Optimizing Multi-Shard Operations

Multi-shard operations, such as joins, are complex operations that involve multiple shards. An MPI process is responsible for scheduling, sequencing, and serving multi-partition transactions across shards. The MPI executes part of the query plan on SPIs and gathers all partial results, aggregates the results, and sends the final result to the client (directly or through an intermediary node).

VoltDB chooses the node with the lowest id (i.e., the first node to run in a VoltDB cluster) as the cluster MPI. If a partial network partition isolates the MPI from some of the SPIs, all the communication with those SPIs will be forwarded through the bridge nodes, significantly increasing operation latency, bridge node load, and network overhead. To avoid these drawbacks, when a partial network partition is detected, NiftyDB migrates the MPI processes to a bridge node (if it is not already on one). One approach to performing this migration is to kill the old MPI process and start a new one on the new node. This approach will cause current multi-shard operations to fail and clients to retry them. An alternative approach is to redirect new requests to the new MPI and keep the old MPI running until it finishes its ongoing operations. NiftyDB uses the first approach due to its ease of implementation.

6.3 Optimizing the Client Protocol

VoltDB clients send their queries to any node in the VoltDB cluster. For single shard queries, the node only answers the query if it is the SPI of the requested shard, otherwise, it will forward it to the SPI that can answer the query. For multi-shard queries, the node will forward the query to the MPI process. When a node forwards the client request to another SPI or the MPI, it acts as the *client proxy*. When the MPI or the target SPI completes processing the query, it will send its response to the client proxy, which in turn forwards the results to the client.

In case of a partial network partition, if the client sends a query to a VoltDB node, and the target SPI for that query is on the other side of the partition, then the client query and the result will have to be forwarded through a bridge node.

NiftyDB modifies the client protocol to reduce the instances in which queries or the results are forwarded through the bridge nodes. In NiftyDB, the NIFTY process that receives the request will identify the bridge nodes and reply to the client with their addresses. The client then directly sends queries to one of the bridge nodes at random. Following this approach reduces the number of hops a request and the corresponding reply take before getting the results back to the client.

Chapter 7

Implementation

We used C++ to implement NIFTY as described in Chapter 5. NIFTY directly installs forwarding rules to the underlying Open VSwitch whenever a connectivity change is detected. NIFTY further provides an API through which other systems can know which of the nodes are bridge nodes. Our implementation is comprised of fewer than 600 lines of code.

To create a partial partition, we implemented a simple network partitioning that uses the end-hosts open-flow tables to install rules that would drop packets from some other nodes in the cluster.

NiftyDB was implemented in two parts: We first modified the VoltDB code by editing/adding fewer than a hundred lines of code to the LeaderElector class which is a class that uses ZooKeeper to orchestrate all elections that occur in VoltDB and SpScheduler class that is responsible for SPI allocations and operations in the system. For optimizing the client protocol, we further used the API provided by NIFTY to change the nodes to which the clients are connected to.

Chapter 8

Evaluation

We evaluate the overhead using NIFTY adds and the effectiveness of the optimizations implemented in NiftyDB. First, we show the overhead added to the network and end-hosts by running NIFTY to a cluster of VoltDB nodes and how the cluster performs under a partial partition. We then discuss the effects of the optimizations discussed in Chapter 6 by comparing NiftyDB to a cluster of VoltDB with NIFTY.

8.1 Platform and setup

We used a cluster of 28 nodes in CloudLab [53]. Each of the nodes has a 64-bit Intel Xeon D-1548 with eight cores at running at 2.0 GHz, 64GB of RAM, and a 10 Gbps NIC. We used Open VSwitch v2.5.5 and VoltDB v9.0. Some of the nodes are used to run the database system and the rest of the cluster to run clients.

Workload: We used two kinds of workloads to evaluate different parts of the system: the Yahoo benchmark with a uniform distribution, and a synthetic benchmark that consists of SQL join query requests. Client nodes issue requests and receive results in a closed loop. The number of clients per node varies per benchmark; for the Yahoo benchmark, we have 150 client threads per physical node, while in the synthetic benchmark we have 16 client threads per physical node.

In all settings, VoltDB was configured with 12 sites per node with a redundancy value ($k = 1$); when we run six VoltDB nodes, this gives a total of 36 partitions in the VoltDB cluster. Before running any of the workloads, we pre-populated the database with $10,000,000$

rows, each with a size of 1KB, giving the database a total size of 10 GB (20 GB for the synthetic workload).

We compare the following alternatives.

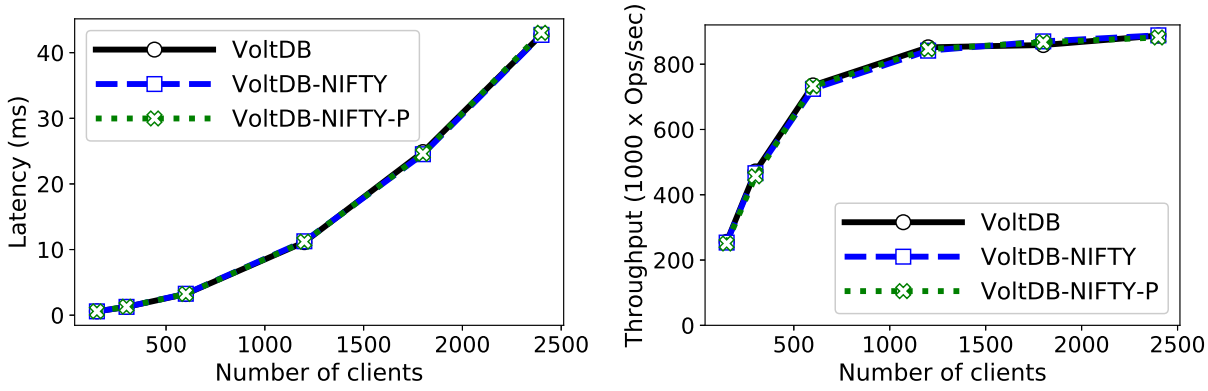
- **VoltDB.** We used VoltDB as our performance baseline when we ran experiments without a network partition. We did not run VoltDB for experiments that involve a partial network partition as it does not provide the same level of fault tolerance as NiftyDB and may halt the entire cluster.
- **VoltDB-NIFTY.** This configuration runs VoltDB atop NIFTY without any changes to VoltDB. With this configuration, NIFTY masks partial partitions, improving VoltDB fault tolerance.
- **NiftyDB.** This is our system with the two optimizations enabled.

We ran each experiment 30 times. What we report is the average of these 30 runs.

8.2 NIFTY Overhead

Every NIFTY process periodically sends its connectivity information to all nodes in the cluster and reacts when a change in connectivity is detected. In this section, we measure the overhead added by NIFTY. In particular, we compare the average latency (Figure 8.1a) and the throughput (Figure 8.1b) of VoltDB and VoltDB over NIFTY (VoltDB-NIFTY). We used VoltDB-NIFTY in two configurations: VoltDB-NIFTY, which runs the system without partial partitions, and VoltDB-NIFTY-P, which runs the system with a network partition that splits the six-node VoltDB deployment into three equal groups (i.e., two nodes on each side of the partition and two bridge nodes). We used the read-only workload C of the Yahoo benchmark [54].

Figure 8.1 compares the throughput and average latency when increasing the number of clients. Figure 8.1 shows that NIFTY overhead is negligible: the difference in throughput and latency of VoltDB compared to VoltDB-NIFTY is negligible even in the presence of a network partition, which requires rerouting a third of the requests through a bridge node. NIFTY overhead is negligible; although it exchanges periodic messages, these messages are small (around 250 Bytes). For perspective, VoltDB nodes exchange around 230 KB of data every second, three orders of magnitude more than NIFTY.



(a) Latency

(b) Throughput

Figure 8.1: NIFTY overhead effect on systems. Yahoo read-only benchmark comparing a cluster of VoltDB, VoltDB with NIFTY, and VoltDB with NIFTY under a partial partition.

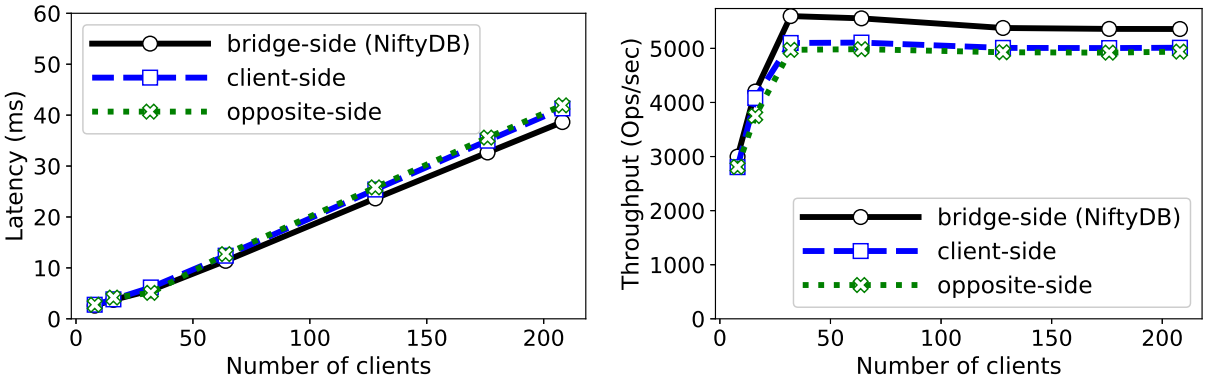
We ran the same experiment with two other Yahoo workloads: workload B with 95% reads and 5% writes, and workload A with 50% reads and 50% writes. Our results with those workloads are similar to workload C.

8.3 NiftyDB Optimizations

In this section we evaluate the effectiveness of the two optimizations implemented in NiftyDB.

8.3.1 Multi-Shard Optimization

The multi-shard optimization in NiftyDB migrates the MPI processes to a bridge node to reduce network load and operation latency. To evaluate the effectiveness of this optimization we deployed VoltDB over NIFTY on nine nodes and created a partial network partition with four nodes on each side of the partition and one bridge node. We evaluated the effect the location of the MPI process has on system performance and imposed overhead. We compare three MPI placements: in the client-side of the partition (client-side in Figure 8.2), in a bridge node (bridge side), and in the side opposite to the client (opposite side). We note that the bridge side placement represents NiftyDB.



(a) Latency

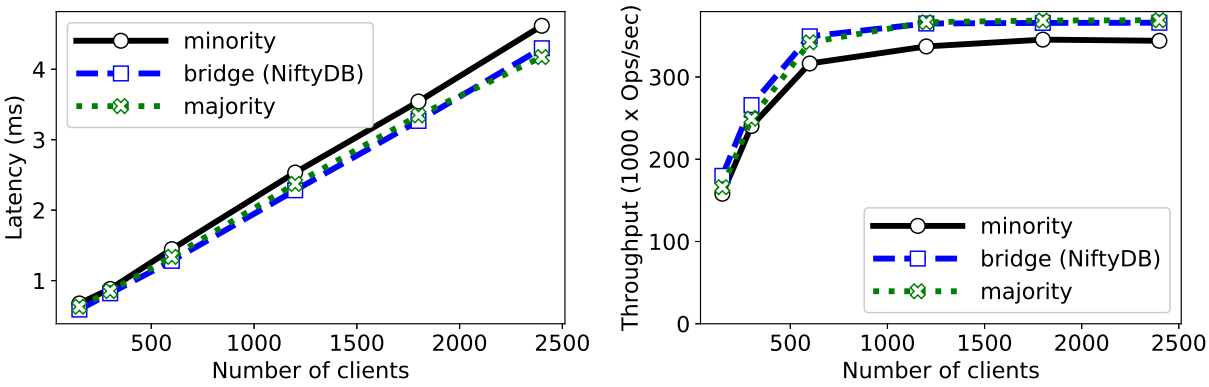
(b) Throughput

Figure 8.2: MPI placement effect on VoltDB-NIFTY's performance using a synthetic benchmark of multi-shard join queries.

Workload. We used a synthetic benchmark that touches multiple shards. We used two tables of 20 fields. Each field is 50 bytes. We populated the database with 20 GB of data before running the experiments. 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 8 KBs. We had to use a simple query in our benchmark as the open-source version of VoltDB has limited support for multi-shard joins.

Figure 8.2 shows (a) the system throughput and (b) the average latency for the three possible MPI placements. The figure shows that the average query's latency is reduced by up to 10.8% and the throughput is improved by 12.6% when the MPI is placed on a bridge node. The main reason for this improvement is that having the MPI on a bridge node reduces the number of hops each of the join queries have to go through before the MPI accumulates all the results and replies back to the client-proxy node.

We measured the amount of data forwarded through the bridge nodes for each one of those configurations and found that placing the MPI on the bridge node imposes the least overhead on the system. While placing the MPI on a bridge node resulted in forwarding 72 MB through bridge nodes, placing the MPI on the client side leads to forwarding 5 GB, and on the opposite side to forwarding 6.5 GB (for the case of 128 clients).



(a) Latency

(b) Throughput

Figure 8.3: Client-proxy location's effect on VoltDB-NIFTY's performance using the read-only Yahoo benchmark

8.3.2 Client Protocol Optimization

The client protocol optimization in NiftyDB reduces the operation latency and reduces the system overhead by limiting the number of times in which queries and their results traverse a bridge node. To evaluate the effectiveness of this optimization we deployed VoltDB over NIFTY on ten nodes and created a partial network partition with one node on one side (minority side), eight nodes on the other side (majority side) and one node as a bridge node. We used this configuration with an asymmetric partition to measure the impact of client location. With VoltDB we evaluated three client placements: on the minority side (minority in Figure 8.3), on bridge (bridge), and majority (majority). We used a single shard workload using the Yahoo benchmark workload detailed at the beginning of this chapter.

Figure 8.3 shows (a) system throughput and (b) average latency while varying the number of clients. The results show that placing the client on the bridge node achieves the highest throughput and lowest operation latency. We observed 13.8% higher throughput and 12.9% lower latency compared to clients connected to the minority side, and 8.3% higher throughput and 6.5% lower latency compared to clients connected to the majority side. This is because this configuration reduces the number of hops a request and the corresponding reply can take. Clients placed on the minority side experience the worst performance as most of the requests/replies will have to traverse a bridge node. The client on the majority side experiences slightly higher latency as some requests need to traverse

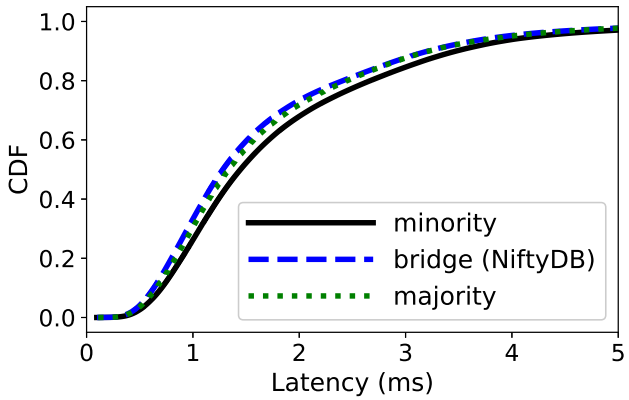


Figure 8.4: CDF of latency for the different client-proxy locations, taken at 600 clients

a bridge node.

Figure 8.4 shows the CDF of latency for the three different settings, taken at the point when 600 clients accessed the system. The figure shows that having the clients connected to a bridge node leads to an improvement in the latency of the system when compared to the two other settings. Overall, we found that having a client-proxy on a bridge node leads to up to 12% better latency than having client-proxies on the minority side (taken at 60th percentile).

Chapter 9

Related Work

To the best of our knowledge, this is the first in-depth analysis of distributed systems fault tolerance mechanisms targeting partial network partitioning. This is further the first work to describe and implement a generic layer that provides partial partitioning fault tolerance. Our manual analysis of a large number of partial network partitioning failures allowed us to identify common vulnerabilities and to find failure characteristics that can improve system designs and 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 [55] and virtual machines [56], network devices [19, 21], storage systems [57, 58], software bugs [59], and job failures [60, 61, 62]. Another set characterized a broader set of failures, either for a specific domain of systems and services, such as HPC [63, 64, 65], IaaS clouds [66], data-mining services [67], hosting services [2, 68], and data-intensive systems [69, 59, 60], or for generic systems such as the work done by Yuan et al. [70]. Our work complements these efforts by focusing on failures triggered by partial network partitioning.

Alquraan et al. [22] studied 136 network partitioning failures from 25 distributed systems. Chapter 3.4 shows how the characteristics of partial partitioning failures differ from generic partitioning failures as reported in [22]. Alquraan et al. further developed the NEAT tool that can be used to test different systems components against the different kinds of partitions. We used NEAT in this paper to reproduce some of the failures to understand their intricate details. Furthermore, using NEAT we were able to find a new failure in Elasticsearch as detailed in Chapter 4.

Recent research projects utilize SDN capabilities to provide load balancing [71, 72], access control [73], seamless VM migration [74], improve MPI performance [75] and to

improve system security, virtualization and network efficiency [76]. Others have used SDN capabilites to build complete systems or protocols, including key-value stores [77, 78], consistency protocols [79, 80, 81], and load balancers [82, 83].

Open VSwitch technologies have mostly been used in the domain of network virtualization, as used by Google in their virtualization platform Andromeda [84], and in overlay networking [85]. Furthermore, Open VSwitch is used in network traffic measurement [86, 87], and in the implementation of SDN-based firewalls [88].

Consistency in a partitioned network: a survey: The CAP theorem as presented in [13], states that in the presence of a network partition, systems can either maintain data consistency or service availability, but not both. We found that system designers either choose one of the two, e.g., data consistency as in VoltDB [43] and HBase [89], or let the system users make that choice via configuration as in RabbitMQ [44].

Chapter 10

Conclusion and Future Work

We conducted a comprehensive study of 50 failures related to partial partitions in 12 widely used distributed systems. We then presented nine findings compiled from the studied failures. Our findings show that partial partitioning failures are some of the most catastrophic and silent failures in modern distributed systems.

We then dissected all the systems that solved tickets related to partial network partitioning to understand their generic techniques for dealing with partial partitions. Overall, from studying eight systems, we found that they all follow one of four approaches to dealing with partial partitioning: constructing a cluster-wide graph of nodes, asking neighbors if they see the same partition, verifying the non-existence of a partial partition before conducting an election, and having the master neutralize unreachable nodes.

We presented NIFTY: A generic layer to work around partial partitions by monitoring the health of cluster nodes and restore connectivity whenever the partition heals. While all the studied systems perform in reduced redundancy, availability, or completely become unavailable in the presence of partial partitions, using NIFTY atop any of the systems makes it tolerant to partial partitions without the underlying system realising the presence of the fault. We show how NIFTY could be augmented with a distributed system and how it can further be optimized to improve their performance under a partial partition by building NiftyDB: A database system that is built atop VoltDB and that leverages the monitoring API of NIFTY.

Our evaluation revealed that NIFTY adds negligible overhead to the network when the system is not under a partition and under most kinds of partitions while providing the underlying system with protection against partial partitions. We further show that NiftyDB's optimization reduces the overhead in the system under some kinds of partitions.

References

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

[2] 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.

[3] 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.

[4] 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.

[5] 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.

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

[7] 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.

[8] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 398–407. ACM, 2007.

[9] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: amazon's highly available key-value store. In *ACM SIGOPS operating systems review*, volume 41, pages 205–220. ACM, 2007.

[10] Flavio P Junqueira, Benjamin C Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*, pages 245–256. IEEE, 2011.

[11] 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.

[12] Douglas B Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 309–324. ACM, 2013.

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

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

[15] Cisco data center infrastructure 2.5 design guide. Cisco Systems, Inc., 2011.

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

[17] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] bnx2 cards intermittantly going offline. https://www.spinics.net/lists/netdev/msg152880.html. Accessed: 2019-07-05.

[25] 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.

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

[27] 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.

[28] Observability in paxos clusters. https://davecturner.github.io/2017/08/18/observability-in-paxos.html. Accessed: 2019-05-18.

[29] Onos 1.4 test plan - ha. https://wiki.onosproject.org/pages/viewpage.action?pageId=7439437. Accessed: 2019-05-18.

[30] Partial network partitions and obstacles to innovation. https://rachelbythebay.com/w/2012/02/16/partition/. Accessed: 2019-05-18.

[31] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. Accessed: 2019-05-18.

[32] Healthchecking is not transitive. https://www.robustperception.io/healthchecking-is-not-transitive. Accessed: 2019-05-18.

[33] cluster broken after switches upgrade. https://github.com/elastic/elasticsearch/issues/9495. Accessed: 2019-05-18.

[34] using map output fetch failures to blacklist nodes is problematic. https://issues.apache.org/jira/browse/MAPREDUCE-1800. Accessed: 2019-05-18.

[35] Asymmetrical network partition can cause the election of two primary nodes. https://jira.mongodb.org/browse/SERVER-9730. Accessed: 2019-07-05.

[36] 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: 2019-07-05.

[37] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. Accessed: 2019-07-05.

[38] Mapreduce ticket 4832. https://issues.apache.org/jira/browse/MAPREDUCE-4832. Accessed: 2019-07-05.

[39] minimum_master_nodes does not prevent split-brain if splits are intersecting. https://github.com/elastic/elasticsearch/issues/2488. Accessed: 2019-05-20.

[40] A network partition can cause in flight documents to be lost. https://github.com/elastic/elasticsearch/issues/7572. Accessed: 2019-07-05.

[41] Nodemanagers die on startup if they can't connect to the rm. https://issues.apache.org/jira/browse/MAPREDUCE-3963. Accessed: 2019-07-05.

[42] Jepsen: A framework for distributed systems verification, with fault injection. https://github.com/jepsen-io/jepsen. Accessed: 2019-07-05.

[43] Voltdb in-memory database platform. https://www.voltdb.com/. Accessed: 2019-07-05.

[44] Rabbitmq message broker. https://www.rabbitmq.com. Accessed: 2019-07-05.

[45] Elasticsearch: Distributed search & analytics. https://www.elastic.co/products/elasticsearch. Accessed: 2019-07-05.

[46] Partial network partitioning leads to cluster unavailabilitys. https://github.com/elastic/elasticsearch/issues/43183. Accessed: 2019-07-05.

[47] Mongodb: The database for modern applications. https://www.mongodb.com/. Accessed: 2019-07-05.

[48] Faulty recovery caused by partial network partitions. https://github.com/elastic/elasticsearch/pull/8720. Accessed: 2019-07-05.

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

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

[51] If block report races with closing of file, replica is incorrectly marked corrupt. https://issues.apache.org/jira/browse/HDFS-2791. Accessed: 2019-07-05.

[52] Splitlogmanger async delete node hangs log splitting when zk connection is lost. https://issues.apache.org/jira/browse/HBASE-5606. Accessed: 2019-07-05.

[53] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[54] 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*, pages 143–154. ACM, 2010.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] Xin Chen, Charng-Da Lu, and Karthik Pattabiraman. Failure analysis of jobs in compute clouds: A google cluster case study. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 167–177. IEEE, 2014.

[62] Peter Garraghan, Paul Townend, and Jie Xu. An empirical failure-analysis of a large-scale cloud computing environment. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 113–120. IEEE, 2014.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

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

[70] 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.

[71] Nikhil Handigol, Mario Flajslik, Srini Seetharaman, Nick McKeown, and Ramesh Johari. Aster* x: Load-balancing as a network primitive. In *9th GENI Engineering Conference (Plenary)*, pages 1–2, 2010.

[72] Richard Wang, Dana Butnariu, Jennifer Rexford, et al. Openflow-based server load balancing gone wild. *Hot-ICE*, 11:12–12, 2011.

[73] Ankur Kumar Nayak, Alex Reimers, Nick Feamster, and Russ Clark. Resonance: dynamic access control for enterprise networks. In *Proceedings of the 1st ACM workshop on Research on enterprise networking*, pages 11–18. ACM, 2009.

[74] Ali José Mashtizadeh, Min Cai, Gabriel Tarasuk-Levin, Ricardo Koller, Tal Garfinkel, and Sreekanth Setty. Xvmotion: Unified virtual machine migration over long distance. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 97–108, 2014.

[75] Mohammed Alfatafta, Zuhair AlSader, and Samer Al-Kiswany. Cool: A cloud-optimized structure for mpi collective operations. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 746–753. IEEE, 2018.

[76] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. Network innovation using openflow: A survey. *IEEE communications surveys & tutorials*, 16(1):493–512, 2013.

[77] 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.

[78] 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.

[79] Dan RK Ports, Jialin Li, Vincent Liu, Naveen Kr Sharma, and Arvind Krishnamurthy. Designing distributed systems using approximate synchrony in data center networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 43–57, 2015.

[80] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say {NO} to paxos overhead: Replacing consensus with network ordering. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 467–483, 2016.

[81] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 104–120. ACM, 2017.

[82] 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.

[83] 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.

[84] 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.

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] Apache hbase. https://hbase.apache.org/. Accessed: 2019-07-05.