Partial Network Partitioning

BASIL ALKHATIB, University of Waterloo, Canada
SREEHARSHA UDAYASHANKAR, University of Waterloo, Canada
SARA QUNAIBI, University of Waterloo, Canada
AHMED ALQURAAN, University of Waterloo, Canada
MOHAMMED ALFATAFTA, University of Waterloo, Canada
WAEL AL-MANASRAH, University of Waterloo, Canada
ALEX DEPOUTOVITCH, Huawei Research Canada, Canada
SAMER AL-KISWANY, University of Waterloo, Canada

We present an extensive study focused on partial network partitioning. Partial network partitions disrupt the communication between some but not all nodes in a cluster. First, we conduct a comprehensive study of system failures caused by this fault in 13 popular systems. Our study reveals that the studied failures are catastrophic (e.g., lead to data loss), easily manifest, and are mainly due to design flaws. Our analysis identifies vulnerabilities in core systems mechanisms including scheduling, membership management, and ZooKeeper-based configuration management.

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

Finally, our findings motivate us to build Nifty, a transparent communication layer that masks partial network partitions. Nifty builds an overlay between nodes to detour packets around partial partitions. Nifty provides an approach for applications to optimize their operation during a partial partition. We demonstrate the benefit of this approach through integrating Nifty with VoltDB, HDFS, and Kafka.

CCS Concepts: • **Computer systems organization → Cloud computing**; **Reliability**; **Availability**; • **Networks → Network reliability**.

Additional Key Words and Phrases: network failures, fault tolerance, partial network partitions, distributed systems, reliability.

1 INTRODUCTION

Modern networks are complex. They use heterogeneous hardware and software [1], deploy diverse middleboxes (e.g., NAT, load balancers, and firewalls) [2–4], and span multiple data centers [2, 4]. Despite the high redundancy built into modern networks, catastrophic failures are common [1, 3, 5, 6]. Nevertheless, modern cloud systems

Authors' addresses: Basil Alkhatib, b2alkhatib@uwaterloo.ca, University of Waterloo, Canada; Sreeharsha Udayashankar, sreeharsha.udayashankar@uwaterloo.ca, University of Waterloo, Canada; Sara Qunaibi, squnaibi@uwaterloo.ca, University of Waterloo, Canada; Ahmed Alquraan, ahmed.alquraan@uwaterloo.ca, University of Waterloo, Canada; Mohammed Alfatafta, m.alfatafta@uwaterloo.ca, University of Waterloo, Canada; Wael Al-Manasrah, wael.al-manasrah@uwaterloo.ca, University of Waterloo, Waterloo, Canada; Alex Depoutovitch, alex.depoutovitch@huawei.com, Huawei Research Canada, Canada; Samer Al-Kiswany, University of Waterloo, Canada, salkiswany@uwaterloo.ca.

0734-2071/2022/12-ART $15.00
https://doi.org/10.1145/3576192

are expected to be highly available [7, 8] and to preserve stored data despite failures of nodes, networks, or even entire data centers [9–11].

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

In our previous work [12] we identified this fault and presented examples of how it leads to system failures. Other than our previous effort, we did not find any in-depth analysis of partial network partition failures and of their fault tolerance techniques. Nevertheless, we found 54 reports of failures caused by partial network partitioning faults[1] in the publicly accessible issue tracking systems of 13 production-quality systems (Section 4), numerous blog posts and discussions of this fault on developers' forums (Section 3), and eight popular systems with fault tolerance techniques specifically designed to tolerate this type of fault (Section 5).

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

It is important to understand that *partial* partitions are fundamentally different from *complete* partitions [12]. Complete partitions split a cluster into two completely disconnected sides and are well studied with known theoretical bounds (CAP theorem [13]) and numerous practical solutions [14–17]. On the contrary, a cluster experiencing a partial partition is still connected but not all-to-all connected. Consequently, the theoretical bounds of complete partitions do not apply to partial partitions, and fault tolerance techniques for complete partitions are not effective in handling partial partitions (Section 10).

An analysis of partial network partitioning failures. We conduct an in-depth study of 54 partial network partitioning failures from 13 cloud systems (Section 4). We select a diverse set of systems, including database systems (MongoDB and HBase), file systems (HDFS and MooseFS), an object storage system (Ceph), messaging systems (RabbitMQ, Kafka, and ActiveMQ), a data-processing system (MapReduce), a search engine (Elasticsearch), an in-memory data grid (Hazelcast), and resource managers (Mesos and DKron). For each considered failure, we carefully study the failure report, logs, discussions between users and developers, source code, and code patches.

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

Ease of manifestation. Unfortunately, these failures can easily occur. The majority of the failures are deterministic and require less than four events (e.g., read or write request) for the failure to occur. Even worse, all the studied

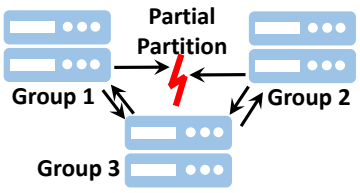


Fig. 1. Partial partition. Groups 1 and 2 are disconnected, while Group 3 can reach both sides of the partition.

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

failures can be triggered by partially partitioning a single node. The majority of failures do not require client access or can be triggered by clients only accessing one side of the partition.

Flaws in core system mechanisms. We find that the majority of failures are due to design flaws. We dissect the design of the studied systems and study the code patches. We identify flaws in five common distributed system mechanisms including leader election, scheduling, membership management, discovery service, and cluster management using a coordination service.

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

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

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

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

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

Nifty demonstrates two main insights. First, Nifty shows that tolerating partial partitioning does not require elaborate techniques such as the ones adopted by current systems (Section 5). It shows that it is possible to build a transparent fault tolerance technique by extending the current membership and connectivity monitoring mechanisms [18–20] with a simple rerouting capability. Second, unlike complete partitions that can not be masked from the client and impact the system semantics, Nifty shows that partial partitions can be masked transparently in an application agnostic way.

Nifty reroutes packet between end hosts to mask partial partitions. This approach increases the load on the intermediate nodes and can create a performance bottleneck. To reduce the load on intermediate nodes, system designers may optimize the data or process placement or employ a flow-control mechanism. Nifty provides an API that exposes the network state to the system running atop of it and facilitates building system-specific optimizations.

To demonstrate Nifty's effectiveness, we deploy it with seven systems: HDFS, Kafka, RabbitMQ, ActiveMQ, MongoDB, VoltDB and Redis Pubsub. We choose these systems because they are data intensive and popular systems. Furthermore, RabbitMQ and VoltDB implement generic techniques to tolerate partial partitions. Our

prototype evaluation with synthetic and real-world benchmarks shows that Nifty effectively masks partial partitions while adding negligible overhead.

To demonstrate the utility of the Nifty API, we integrate Nifty with HDFS, VoltDB, and Kafka and explore a number of optimizations. Our evaluation shows that system-specific optimizations can significantly reduce the traffic rerouting overhead during partial partitions.

This paper extends our previous work [21] on three fronts. First, our analysis revealed that the majority of studied failures are due to design flaws. In this paper, we study the implementation of nine popular systems to extract the design of their fault tolerance technique and identify the shortcomings of those techniques. Second, we demonstrate the utility of the Nifty API by extending HDFS and Kafka. We optimize the replication protocol and discovery service within these systems using the Nifty API. Our evaluation shows that the proposed optimizations can significantly improve system performance under partial network partitions. Finally, we extended our study by discussing recent service outages caused by partial partitions at Google, Amazon, CloudFlare, and Lyft; and studying failure reports in an one additional system (HazelCast).

2 DEFINITIONS

A *partial network partition* is a network fault that prevents at least one node (e.g., a node in Group 1 in Figure 1) from communicating with another node (Group 2) in the cluster, while a third node (Group 3) can communicate with both affected nodes. Nodes in a partially partitioned cluster are still connected but are not all-to-all connected (i.e., they do not form a complete graph [22]).

While multiple concurrent partial partitions are theoretically possible and may lead to complex failures, all the failures we study are caused by a single partial partition. Consequently, we focus the rest of our discussion on a single partial partition. A single partial partition divides a cluster into three groups: two sides and one bridge group. We identify a node as an *bridge* node if it can reach at least one node on each side of a partition. A single partial partition has two *sides*, all the nodes on one side of the partition cannot reach all the nodes on the other side of the partition.

We define a *single-node partial partition* (Figure 2) as a partial partition that has a single node on one side of the partition, while the rest of the cluster nodes are bridge nodes or are on the other side of the partition. For instance, a single-node partial partition can be caused by a firewall misconfiguration that prevents a node from communicating with some other nodes.

3 CAUSES OF PARTIAL NETWORK PARTITIONING FAULT

Recent reports indicate that network partitioning faults are common and happen at various scales. Connectivity loss between data centers [1] leads to network partitions in geo-replicated systems. Wide area network partitions happen as frequently as once every four days [6]. Switch failures can cause a network partition in a data center [5]. Switch failures caused 40 network partitions in two years at Google [3] and 70% of the downtime at Microsoft [5]. On a single node, NIC [23] or software failures can partition a node that may host multiple VMs. Finally, network

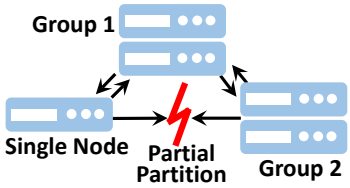


Fig. 2. Single-node partial partition. A partial partition with a single node on one side while the rest of the nodes are bridge nodes (Group 1) or are on the other side (Group 2).

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

While we did not find failure reports that detail partial partitioning faults, we found numerous discussions of their impact on production systems. Partial partitions were the cause of service outages at Cloudflare [24], Google [25], Lyft [26], and Amazon AWS [27]. A misbehaving switch caused the failure at Cloudflare. The switch data plane did not processes all packets, while the control plane protocols remained operational. This disrupted the communication between some nodes in the cluster and eventually caused a 6-hour outage of Cloudflare. AWS [27] also blames a misbehaving switch for a partial partitioning failure that affected applications that span multiple availability zones. A partial partition also affected Google Compute Engine (GCE) services. When a new VM is added, GCE uses two mechanisms to inform the other VMs: one to inform VMs in the same zone as the new VM, and another to inform VMs in other zones. When the processes responsible for informing the VMs in other zones failed [25], the newly added VM became unreachable from VMs outside its zone. This created a partial partition since some VMs (in the same zone as the new VM) could reach all VMs as they are updated through a separate mechanism. However, other VMs (from outside the zone) could not reach the new VM. Lyft reported cases of partial network partitions while running Kafka at scale on AWS [26]. Finally, an early version of Google's B4 control plane uses a primary master with a standby backup. A partial partition disconnected the primary master from the standby backup while both can reach the switches. When the standby backup could not reach the master it assumed that the master failed and started working as a master. This led to having two active masters in the infrastructure [4].

Furthermore, we found 54 failure reports detailing system failures due to partial network partitions, and numerous articles and online discussions discussing the fault [28–31]. Some of these reports and discussions mention the root cause of the partial partition. Partial partitions are caused by a connectivity loss between two data centers while both are reachable by a third center [1], the failure of additional links between racks [32, 33], network misconfiguration [34], firewall misconfiguration [34], network upgrades [35], and flaky links between switches [36].

4 ANALYSIS OF PARTIAL NETWORK-PARTITIONING FAILURES

We conduct an in-depth study of partial network partitioning failures reported in 13 popular systems (Table 1). We aim to understand the impact and characteristics of these failures and to identify opportunities for improving system resilience.

4.1 Methodology

We choose 13 diverse and widely used systems (Table 1), including two databases, a data analysis framework, two file systems, three messaging systems, a storage system, a search engine, an in-memory data grid, and two resource managers.

We selected the 54 failures in our study from publicly accessible issue-tracking systems. First, we used the search tools in the issue-tracking systems to find tickets related to partial network partitioning. Users did not classify network partitioning failures based on the partition type, so we had to search for all network partitioning failures and manually identified partial partitioning failures. We used the following keywords: "network partition," "partial network partition," "partial partition," "network failure," "switch failure," "isolation," "split-brain," and "asymmetric partition." Second, we considered tickets that were dated 2011 or later. Third, we excluded tickets marked as "Minor." For each ticket, we studied the failure description, system logs, developers' and users' comments, and code patches. For tickets that lacked enough details (e.g., missing output logs or did not have details about the affected mechanism), we manually reproduced them using NEAT [12]. Finally, during our evaluation, we found and reported bugs in Kafka and Elasticsearch. We included these failures in our study. Among the selected tickets

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

System	Category	Failures	
		Total	Catastrophic
Elasticsearch [37]	Search engine	17	17
MongoDB [38]	Database	9	5
RabbitMQ [18]	Messaging	5	3
MapReduce [39]	Data processing	4	2
HBase [40]	Database	3	2
Mesos [41]	Resource manager	2	1
Hazelcast [42]	In-memory data structures	2	2
Kafka [43]	Messaging	3	3
HDFS [39]	File system	3	1
Ceph [20]	Storage system	2	2
MooseFS [44]	File system	2	2
ActiveMQ [45]	Messaging	1	1
DKron [46]	Resource manager	1	1
Total	-	**54**	**42**

50 are classified as bugs and four as high priority improvements. Up to the date of the publication, forty two have been resolved. Figure 3 shows the distribution of the selected tickets over the years.

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

4.2 Limitations

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

(1) *Representativeness of the studied systems.* Although we study 13 diverse systems (Table 1), our results may not be generalizable to systems we did not study. The selected systems follow diverse designs from strongly consistent (MongoDB, HBase, and Ceph) to eventually consistent (Elasticsearch) designs and from systems

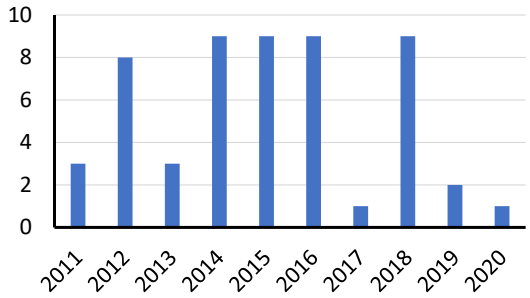


Fig. 3. Histogram of the tickets creation date.

persisting data on disks and replicating data in-memory across nodes (Hazelcast) to caching systems. They follow a primary-backup or peer-to-peer architecture and use synchronous or asynchronous replication. The selected systems are widely used: Kafka, ActiveMQ, and RabbitMQ are the most popular open-source messaging systems; MapReduce, HDFS, and HBase are the core of the Hadoop platform; Elasticsearch is a popular search system; and MongoDB is a popular database.

(2) *Priority bias.* We avoid tickets marked by the developers as low-priority. This sampling methodology may bias the results.
(3) *Limited number of tickets.* We study all 54 tickets that we found following our methodology. To increase confidence in our findings, we only report findings that apply to at least two-thirds of the studied failures. A third of our findings apply to all failures.
(4) *Observer error.* To reduce the chance for observer errors, two team members study every failure report using the same classification methodology. Then, we discuss the failure in a group meeting before reaching a verdict.

4.3 Findings

This section presents a summary of our findings [21]. Our study indicates that partial network partitioning leads to catastrophic failures that are easy to manifest. Luckily, our study identifies that code reviews and targeted testing can improve systems fault tolerance. We refer the reader to our previous paper for a detailed discussion of our findings [21].

Failure Impact. Overall, *we find that 76.4% of the studied failures lead to catastrophic effects.* A failure is said to be catastrophic if it leads to a system crash or violates the system's guarantees such as permanent data loss or corruption, system unavailability, and stale or dirty reads. The majority of non-catastrophic failures lead to reducing a system availability such as intermittent disruption of system operation [47].

Data loss and system unavailability are the two most common effects of partial partitions and are the result of 42.5% of failures. For instance, in HBase, region servers store their logs on HDFS. When a log reaches a certain size, the region server creates a new log and informs the master of the new log location. If a partial partition isolates a region server from the master while both can reach HDFS, the master assumes that the region server has failed and assigns its logs to a new region server. If at this time the old region server creates a new log, the master will not know about it, and the entries in the new log will be lost [48].

The majority of failures (81.5%) are silent, meaning the user is not informed about their occurrence. Some systems return a warning to the user when an operation fails due to partial network partitioning, but these warnings are ambiguous with no clear mechanisms for resolution. For example, in Elasticsearch, if a client sends a request to a replica that is partially isolated from the other replicas, the replica will return "a rejected execution" exception [49]. This confusing warning does not inform the user of the problem's actual cause nor the steps needed to resolve it. This is unsettling because a lack of error or warning notification delays failure detection.

Ease of manifestation. Unfortunately, the studied failures can easily occur:

- *All the studied failures, except one, are deterministic or have known time constraints,* such as the period before considering a node to have failed.
- *The majority of failures (66.6%) require three or fewer events (other than the partial partition) to manifest.* An event is a user request, a hardware or software fault, or a start of a background operation (e.g., leader election and data rebalancing). This is alarming because in real deployments, many users interact with the system, increasing the probability of failure.
- *Most failures (59.3%) do not require client access or require only that clients access one side of the partition.* To reduce the network partition's impact, some systems limit client access to one side of the partition [50–52]. This finding shows that this fault tolerance technique is not sufficient.

- *All the studied failures can be triggered by a single-node partial partition.* Arguably, single-node partial partitions (Section 2) are more likely than partitioning more than one node. These partitions could happen due to a malfunction of a single top-of-rack switch or a misconfiguration of a single firewall.

We further study which nodes need to be isolated for a failure to manifest. Of the failures, 33.3% manifest by partitioning any node in the system—regardless of its role. Among the failures that require partitioning a specific node, partitioning the leader replica is most common (44.4%). In real deployments, partitioning a leader is likely because almost every node in the cluster is a leader for some shard.

Failure Characteristics. Our study revealed two surprisings characteristics of these failures. First, *the majority of the fixed bugs (59.3%) are due to design flaws.* We consider a code patch to be fixing a design flaw if it significantly changes the implemented protocol or logic, such as changing the mechanism to select a master in Elasticsearch. Second, *partial partition faults affect a wide range of system mechanisms* including leader election, configuration change, replication protocol, request routing, scheduling, and data migration. Leader election, configuration change, and replication protocols are the most affected mechanisms (affected by 72.6% of failures).

Finally, these failures can be easily reproduced with small clusters of five or fewer nodes, and 75.9% require only three nodes. Furthermore, all the failures except one can be reproduced using a fault-injection framework that can inject partial partitioning faults such as NEAT [12].

4.4 Design Pitfalls

Our study reveals that the majority of failures are due to design flaws. For each design failure we study the code patches and the system design to understand the design flaw. We identify flaws in the following five common designs of core distributed system techniques. Revisiting the design of these techniques to tolerate partial network partitioning is a high impact research frontier that requires further investigation.

Leader election is the most vulnerable mechanism to partial partitions. The following are the most frequent flaws we found in the studied tickets.

- *Two leaders.* Partial network partitioning fault leads to having two active leaders in MongoDB [53] and RabbitMQ [54]. Having more than one leader results in data loss, dirty read, and stale read. This failure typically manifests when two nodes on different sides of the partition start the leader election process. If the bridge node votes for the two candidates, each candidate will get enough votes to become a leader. A common solution to avoid this double voting problem is to divide time into terms or epochs and each node has a single vote in a term [14, 55].
- *No leader in the system.* Some leader election policies may leave a cluster without a leader under partial network partitions. For instance, in an earlier version of ElasticSearch, a live node with the smallest id is the cluster leader. If a node can not reach the leader, it will ask the node with the second smallest id to become a leader. The node with the second smallest id will refuse to become a leader if it can reach the current leader. If a partial partition puts the current leader on one side of the partition and the node with the second smallest id is a bridge node, no node will be elected as a leader and the cluster pauses until the partition heals [56]. We discuss this failure in Section 5.2.
- *Leader election thrashing.* Partial partition faults may lead to continuous leader election thrashing if the two sides of the partition keep launching the leader election process. For instance, leader election in MongoDB is based on a majority vote, with an arbiter node included to break ties. Consider a shard that has two replicas (A and B), with A being the leader. If a partial partition disrupts the communication between A and B while both can reach an arbiter, B will detect that A is unreachable and calls for a leader election. Because there is only one candidate in the system, the arbiter votes for it, and B becomes the leader. The arbiter will inform A of the new leader, and A steps down. A will detect that the leader (B) is unreachable, call for a leader election, become a leader, and then B steps down. This leader-election thrashing continues until the

network partition heals [47]. The system is unavailable during leader election, so this failure significantly reduces system availability. CloudFlare reported a service outage due to a similar flaw in the leader election mechanism in etcd [24]. To fix this problem, MongoDB developers changed the leader election protocol to closely resemble the leader election protocol of Raft.

Leader election using a coordination service. A common approach for electing a leader in modern systems (e.g., Mesos, Kafka, ActiveMQ, HBase, and Neo4j) is to use a coordination service such ZooKeeper [57] to monitor the nodes and choose a new leader when a leader fails. As this is a common usage pattern for ZooKeeper, the ZooKeeper user guide has a "recipe" [58] for how to use ZooKeeper for leader election that is broadly followed. Unfortunately, this recipe is vulnerable to partial network partitions.

To elect a leader using ZooKeeper, each node creates a "sequence ephemeral" file in a specific shared directory at ZooKeeper. The file has a unique sequence number that is generated by ZooKeeper. The node with the smallest sequence number is the leader. If ZooKeeper misses heartbeats from a node, it deletes all the ephemeral files that are created by the unreachable node, and notifies the other nodes in the clusters. If the unreachable node was the leader, each node in the cluster will check the shared directory to see if its file has the smallest sequence number. The node with the smallest sequence number becomes the new leader.

If a partial partition isolates the leader from the rest of the cluster while all nodes are reachable from ZooKeeper, the nodes will typically pause their operations because they cannot reach the leader. Because ZooKeeper can reach the current leader, it will not delete its ephemeral file and no new leader will be elected. The cluster remains unavailable until the partial partition heals.

This failure manifested in ActiveMQ Classic [59] and Kafka [60]. Up to the date of this publication, these failures are not fixed. Interestingly, Kafka developers stopped using Zookeeper in Kafka v. 2.8 [61]. One of the main reasons stated for removing this dependency is the possibility of a divergent view of the cluster liveness between the cluster and ZooKeeper [62]. In the latest version, Kafka built its own metadata management mechanism using Raft.

Scheduling. Resource management and scheduling systems use heartbeating to monitor a cluster's health. If a scheduler misses heartbeats from a worker node, it will suspect that the node has failed and will typically reschedule all the tasks that were running on the failed node on other nodes in the cluster.

This fault tolerance technique is vulnerable to partial partitions. If a partial partition isolates the scheduler from one of the nodes, while the affected node can reach the rest of the cluster, the scheduler will reschedule the tasks running on the affected node on other cluster nodes. This leads to double, potentially concurrent, execution of those tasks. Double execution can corrupt shared state (e.g., data on HDFS) or confuse clients. For instance, in MapReduce, a partial partition leads to a double execution and data corruption of shared data [63]. Mesos [64], and ElasticSearch [65] suffered from a similar failure.

Membership management. Modern systems use membership lists (a.k.a. allow/block lists) to keep track of live nodes in the cluster, and avoid slow or unresponsive nodes. If a node detects that another node has failed or is slow, it notifies the metadata service. The metadata service updates the membership list or the block-list to avoid using that node in future operations. Our study shows that under partial partitions, these techniques could lead to a performance and availability degradation.

MapReduce uses block-listing to identify slow or unresponsive nodes. If a reducer cannot reach a mapper node, it will report it to the master node. The master will not assign new tasks to the node running that mapper. If a partial partition isolates a reducer from many nodes, while all nodes are still reachable by the master, the affected reducer will report and unnecessarily block-list many mappers, which leads to a significant drop in cluster performance [36].

RabbitMQ supports message replication for higher availability. RabbitMQ maintains a membership log that lists the current nodes in the cluster. If nodes have conflicting views on which nodes are part of the cluster, the

RabbitMQ cluster crashes. For instance, in a cluster with three nodes (A, B, and C), when a partial partition disconnects B and C, B assumes that C crashed and removes it from the membership log, and C assumes that B crashed and removes it from the membership log. This inconsistency in the cluster membership leads to a complete cluster crash [66].

Discovery service. Modern systems often use a metadata or discovery service to direct clients to a node hosting a queue in a messaging system, or to a leader replica in a storage system. If a partial partition isolates a client from some nodes in the cluster while the discovery service can reach all nodes, the discovery service may point a client to a node that the client can not reach. This problem often leads to system unavailability for some clients. For instance, in Kafka, a client asks the bootstrap service for a list of cluster nodes. If the client cannot reach a topic leader, while the bootstrap service indicates that the leader is alive, messages to that leader will be lost [67]. ElasticSearch had a similar failure [68].

In HDFS, consider a case when a partial network partition separates a client from, say, rack 0, while the NameNode can reach that rack. If the NameNode allocates replicas for a new data chunk on rack 0, then a client write operation will fail, and the client will ask for a different DataNode to place its replica. The NameNode, following its rack-aware data placement, will likely suggest another node from the same rack. The process repeats five times before the client gives up [69].

4.5 Insights

Surprisingly, partial network partitioning faults trigger silent failures that have catastrophic effects in production-quality systems. It is unsettling to realise how easy it is for these failures to manifest once a partial partitioning fault happens. Isolating a single node, with three or less events, with client access to one side of the partition, deterministically causes over two thirds of the failures.

Fortunately, we identify three approaches for improving system resilience to partial partitions. First, because these faults are deterministic and can be reproduced on a five-node cluster, improved testing can reveal the majority of the studied failures. Our analysis finds timing, client access, and partition characteristics that significantly reduce the number of sufficient test cases. Second, the fact that the majority of failures are due to design flaws, indicates that system designers overlook partial network partitioning failures in the design phase. We posit that design reviews focused on network partitioning could identify these vulnerabilities. Since a large number of failures are triggered without client access, our analysis highlights that system designers should consider the impacts of partial partitioning faults on all operations, including background operations.

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

Finally, we point out design flaws in core system mechanisms including leader election, scheduling, discovery service, and membership management (Section 4.4). *Most of the studied failures are caused by the underlying assumption that, if a node can reach a service, all nodes can reach that service, and if a node cannot reach a service then the service is down. Our analysis shows the danger of such assumptions; this leads to a confusing state, wherein some of the system's parts start executing a fault tolerance mechanism, while others presume the whole system is healthy and carry on with normal operations. The mix of these two operation modes is poorly understood and tested.*

5 DISSECTING MODERN FAULT TOLERANCE TECHNIQUES

We study the code patches related to the tickets included in our study. Seven of the systems in Table 1 (MongoDB, Elasticsearch, RabbitMQ, HBase, MapReduce, Hazelcast, and Mesos) changed the system design to incorporate a

fault tolerance technique specific to partial network partitioning faults. The rest of the systems either patched the code with an implementation-specific workaround or did not fix the reported bugs yet.

Furthermore, we find that two additional systems, VoltDB [19, 70] and LogCabin [71] (the original implementation of the Raft [14] consensus protocol), implement fault tolerance techniques for partial partitions. For these two systems, we do not find failure reports related to partial partitioning faults in their issue tracking systems, but VoltDB announced that their recent versions tolerates partial partitions [72]. We experimented with LogCabin to understand the impact partial partitions have on strongly consistent systems and found that LogCabin incorporates a technique to tolerate partial partitions. We include VoltDB and LogCabin in our study.

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

5.1 Identifying the Surviving Clique

Main idea. Upon a partial network partition, the system identifies the maximum clique of nodes [73], which is the largest subset of nodes that are completely connected. All nodes that are not part of the maximum clique are shut down. VoltDB and Hazelcast follow this approach.

VoltDB Implementation. VoltDB [19, 70] is a popular ACID, sharded, and replicated relational database. VoltDB follows a peer-to-peer approach to implement this technique. Every node in the system periodically sends a heartbeat to all nodes. If a node loses connectivity to any node, it suspects that a partial network partition occurred and starts the recovery procedure. The recovery procedure has two phases. In the first phase, the node that detects the failure broadcasts a list of nodes it can reach. When a node in the cluster receives this message, it broadcasts its list of reachable nodes to all nodes in the cluster. In phase two, every node independently combines the information from the other nodes into a graph representing the cluster connectivity. Each node analyzes this graph to detect the maximum completely connected clique of nodes. Every node that finds that it is not part of this "surviving" clique shuts itself down. Figure 4 shows an example in which a partial partition disrupts the communication between nodes 2, 3, and 4 on one side and nodes 5 and 6 on another. Nodes 5 and 6 are not part of the clique and will shut down.

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

Shortcomings. This fault tolerance approach has two severe shortcomings. First, it unnecessarily shuts down up to half of the cluster nodes, reducing the system's performance and fault tolerance. Second, this approach causes a complete cluster shutdown if the surviving clique is missing a single data shard. To understand how likely a cluster is to shut down, we conduct a probabilistic analysis (Appendix A). Figure 5 shows the probability of a complete cluster shutdown while varying the cluster size and the number of nodes that shut down (i.e., nodes that are not part of the surviving clique – the x-axis in Figure 5). Each shard has three replicas. Our analysis shows that isolating only 10% of the nodes leads to more than a 50% probability of shutting down the entire cluster, and isolating only 20% of the nodes leads to a staggering 90% chance of a complete cluster shutdown.

Hazelcast Implementation. Hazelcast [42] offers in-memory sharded and replicated data structures. Every node in the system periodically sends a heartbeat to all nodes. Hazelcast uses a master node to track the cluster membership, i.e., which nodes are part of the cluster. The master periodically sends a membership list to all nodes. A node will ignore membership updates coming from nodes that are not in the membership list.

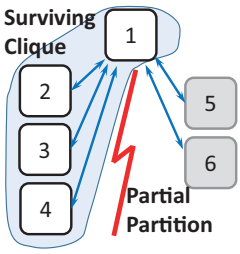


Fig. 4. VoltDB's surviving clique. Gray nodes shut down as they are not in the clique.

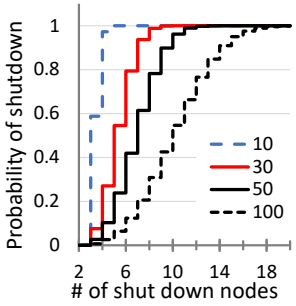


Fig. 5. The probability of a VoltDB cluster shutdown. Different lines represent different cluster sizes. The x-axis shows the number of nodes that are not in the clique.

Hazelcast escalates partial partitions to complete cluster partitions, such that the cluster is split into completely disconnected sub-clusters. When a partial partition occurs the master node collects connectivity information from all nodes. Nodes that are not reachable by the master are removed from the cluster membership list. The master then constructs a graph representing the cluster connectivity, runs the Bron–Kerbosch algorithm [74] to identify the largest fully connected sub-graph that includes the master node, removes all nodes that are not part of this sub-graph from the membership list, and broadcasts an updated membership list. For nodes that are removed from the membership list, Hazelcast supports two policies: pause or form a new cluster. Forming two clusters can lead to data inconsistency. When a partial partition heals, Hazelcast merges conflicting versions of the data using automated data consolidation policies (e.g., versions with latest access time win or versions on the majority side win). Unfortunately, these policies can lose data or keep an inconsistent version of the data [12].
Shortcomings. This fault tolerance approach offers two undesirable alternatives. The cluster may unnecessarily pause a large number of nodes reducing the system's performance and fault tolerance. Note that Hazelcast selects the largest subgraph that includes the master which may not include the majority of nodes. Alternatively, Hazelcast may form multiple clusters leading to data loss or inconsistency.

5.2 Checking Neighbors' Views

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

If S detects a partial network partition, S either disconnects from all nodes that can reach D, which effectively makes the partition a complete partition, or pauses its operation. RabbitMQ and Elasticsearch follow this approach.
RabbitMQ Implementation. RabbitMQ [18] is a popular messaging system that replicates message queues for reliability. In RabbitMQ, if a node detects that its communication with another node (e.g., node D) is affected by a partial partition, it applies one of the following policies depending on its configuration.

(1) *Escalate to a complete partition.* The node will drop its connection with any node that can reach node D. The goal of this policy is to create a complete partition in which both sides work independently. This configuration leads to data inconsistency and requires running a data consolidation mechanism after the partition heals.
(2) *Pause*: To avoid data inconsistency, once a node discovers a partial partition, it pauses its activities. It resumes its activities only when the partition heals. The result of this policy is that a subset of nodes will continue to operate. This subset will be completely connected and will run without sacrificing data consistency.

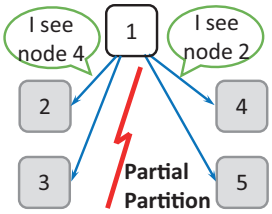


Fig. 6. A scenario for RabbitMQ's *pause* policy. Every non-bridge node pauses (gray nodes) as it detects that it cannot reach one node on the other side.

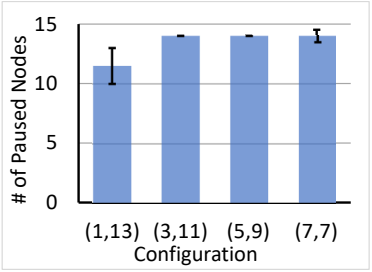


Fig. 7. The median number of paused nodes in a cluster of 15 nodes. In all runs, one node is unaffected by the partition. The notation (i, j) shows the number of nodes on each side of the partition.

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

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

Shortcomings. RabbitMQ's policies have serious shortcomings. Changing a partial partition to a complete partition (policies 1 and 3) may lead to multiple inconsistent copies of the data, whereas the *pause* policy (policy 2) may pause the entire system or the majority of the nodes. For instance, in Figure 6, if every node except node 1 detects that it cannot reach a node on the other side of the partition, it pauses, leading to a complete cluster pause.

In the case of the *pause* policy (policy 2), to determine how many nodes pause under different partial partition scenarios, we conduct an experiment in which we deploy a 15-node RabbitMQ cluster, introduce a partial partition, and observe how many nodes pause. In all experiments, we inject a partition such that one node remains unaffected and able to reach all nodes. Figure 7 shows the median number of paused nodes under various partition configurations. We run each configuration 30 times. Surprisingly, in all configurations almost all the cluster nodes pause because each node detects that it cannot reach at least one node on the other side of the partition. Even isolating a single node (configuration (1,13) in Figure 7) leads to pausing 12 nodes. We experimented with additional configurations with a larger number of bridge nodes and noticed a similar behaviour (Appendix B). Our investigation reveals that nodes declare another node unreachable after missing its heartbeats for a timeout period. In RabbitMQ, the default timeout period is 1 minute, which gives enough time for many nodes to detect the partition and pause. Using a shorter timeout period causes some nodes to prematurely declare that other nodes have failed, even without a partial partition.

Elasticsearch Implementation. Elasticsearch [37] is a popular search engine. Its master election protocol uses a fault tolerance technique based on checking neighbors' views. In Elasticsearch, the node with the lowest ID is the master. If a node (e.g., S) cannot reach the master, it contacts all nodes to check whether they can reach the master. If any node reports that it can reach the master, S pauses its operations. If none of the nodes can reach the master, the node with the lowest ID becomes the new master.

Shortcomings. First, this approach can affect cluster availability quite severely, as all nodes that cannot reach the master pause. In the worst case, it can cause complete cluster unavailability. For instance, in Figure 8, none of the nodes can reach the master except node 2, which refuses to become the new master because it can reach a node with a lower ID (node 1). Consequently, all the nodes in the cluster pause. Furthermore, because the master

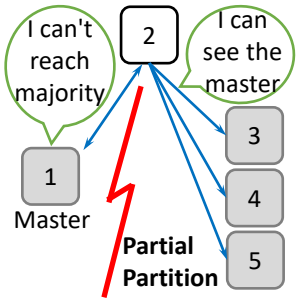


Fig. 8. Elasticsearch unavailability scenario. The master pauses because it cannot reach majority of nodes, and all nodes pause because they cannot reach the master.

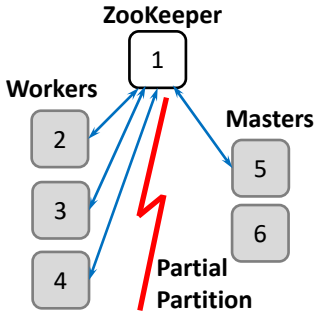


Fig. 9. A Mesos cluster becomes unavailable when a partial partition isolates the master node and its backups.

cannot reach a majority of nodes, it also pauses, which leads to system unavailability [56]. Second, Elasticsearch uses this approach to fortify only the master election protocol, which leaves the rest of the system vulnerable to partial partitions.

5.3 Failure Verification

Main idea. If a node (e.g., S) receives a notification from another node that a third node (D) has failed, node S first verifies that it cannot reach D before taking any fault tolerance steps. This approach is used in the leader election protocols of MongoDB [38], and LogCabin [71]. It was also used in an earlier version of Elasticsearch.

In MongoDB and LogCabin, if a leader is on one side of a partial partition but can still reach the majority of nodes, the nodes on the other side of the partition unnecessarily call for leader election. Section 4.4 discusses a scenario in which a partial partition leads to continuous leader election thrashing and to system unavailability [47]. To avoid unnecessary elections, when a node receives a call for election, it first verifies that the current leader is unreachable. A node participates in an election only if it cannot reach the current leader, else it will ignore the failure report.

Shortcomings. This approach has two major shortcomings. First, it leads to the unavailability of a large number of nodes. Second, it is mechanism specific. Designing a system-wide fault tolerance mechanism using this approach is tricky because one cannot ignore every failure notification. For instance, using this approach in an earlier version of Elasticsearch backfired [75]. During data migration from a primary replica of a shard to a secondary replica, if a partial partition isolates the primary replica from the secondary replica while both are reachable from the master node, the primary requests a new secondary replica. Because the master can reach the secondary replica, it ignores the failure report. This leads to the unavailability of the affected shard [75]. Broadly applying this fault tolerance technique is not feasible because designers have to revisit the design of every system mechanism, consider the consequences of ignoring failure reports, and examine the interaction of various mechanisms under partial partitions.

5.4 Neutralizing Partitioned Nodes

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

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

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

HBase Implementation. In HBase, data shards are managed by an HBase node but are stored on HDFS. If the HBase leader cannot reach one of the HBase nodes, it neutralizes that node by renaming the shard's directory in HDFS. Renaming a shard's directory effectively prohibits the old HBase node from making further changes to the shard [48]. The leader then assigns the shards of that node to a new HBase node.

MapReduce Implementation. In MapReduce, a manager node assigns tasks to AppMaster nodes. If the manager cannot reach an AppMaster, it reschedules the tasks assigned to that AppMaster to a new AppMaster. With partial network partitions, this approach may result in two AppMasters working on the same task, which leads to data corruption [63]. To fix this problem, when an AppMaster completes a task, it writes a completion record in a shared log on HDFS. Before an AppMaster executes a new task, it checks the shared log for a completion record. If it finds one, it does not re-execute the task.

Mesos Implementation. In Mesos, a master node assigns tasks to worker nodes. A Zookeeper instance selects the master node. The master sends periodic heartbeats to workers. If a partial partition isolates a worker node from the master, it pauses its operations. Figure 9 shows a worst-case scenario in which the partial partition isolates the master and its backup from all workers, which leads to a complete cluster unavailability. Finally, if a master detects that one of the workers is unavailable, it marks the tasks that were running on the unreachable worker as lost and reschedules them on new workers. This may lead to the double execution of a task [76].

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

5.5 Summary

Table 2 summarizes the shortcomings of the current fault tolerance techniques, none of which are adequate for modern cloud systems. All current techniques severely affect system availability, as they unnecessarily lose a significant number of nodes. Failure verification and neutralizing partitioned nodes are used to fortify *specific* mechanisms, rather than providing *system-wide* fault tolerance. Using mechanism-specific fault tolerance techniques requires the independent fortification of all system mechanisms and the analysis of the interactions between various mechanisms. This approach complicates system design, fault analysis, and debugging. An example of a system that uses multiple mechanism-specific techniques to tolerate partial partitions is Elasticsearch, which uses checking neighbors' view, failure verification [75], and neutralizing partitioned nodes [77] in different mechanisms. However, Elasticsearch has the highest number of reported failures due to partial partitions (Table 1).

Detecting the surviving clique and checking neighbors' views can be used to build a *system-wide* fault tolerance technique. However, as Table 2 shows, these techniques lead to a complete system shutdown or significant loss of

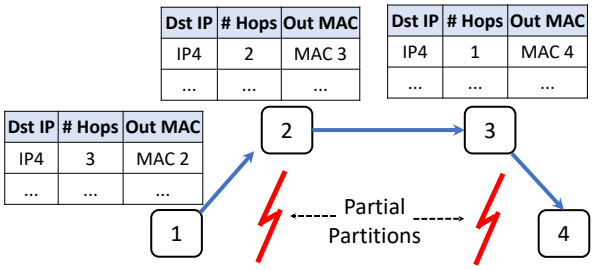


Fig. 10. A Nifty routing example. A partial network partition isolates node 1 from nodes 3 and 4, and another partial partition isolates node 4 from nodes 1 and 2. Communication between 1 and 4 is routed through nodes 2 and 3.

system capacity. This realization motivated us to build Nifty (Section 6), a system-wide fault tolerance technique that overcomes the aforementioned shortcomings.

6 NIFTY DESIGN

To overcome the limitations of current fault tolerance techniques, we design a simple, transparent network-partitioning fault-tolerant communication layer (Nifty). Nifty follows a peer-to-peer design in which every node in the cluster runs a Nifty process. These processes collaborate in monitoring cluster connectivity. When Nifty detects a partial partition, it reroutes the traffic around the partition through intermediate nodes. For instance, in Figure 10, if two partial partitions isolate node 1 from node 4, Nifty reroutes packets exchanged between nodes 1 and 4 through nodes 2 and 3. While our discussion here focuses on a single partial partition, the Nifty design can readily handle multiple concurrent partitions.

Although Nifty keeps the cluster connected, it may increase the load on the bridge nodes, leading to a lower system performance. System designers who use Nifty may optimize the data or process placement or employ a flow-control mechanism to reduce the load on bridge nodes. To facilitate system-specific optimization, Nifty provides an API to identify bridge nodes and nodes on different sides of a partition, and to help take action when a partial partition occurs or heals.

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

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

When a Nifty process detects a change in the cluster (e.g., a node becomes unreachable or reachable), it initiates the route discovery procedure to find new routes. In our prototype, we use the classic Bellman–Ford distance-vector protocol [78, 79] because it is easy to implement. We use hop count as the link weight. By hop, we mean a hop between end nodes. Using hop count naturally favors direct connections, when they exist, over rerouting through intermediate nodes.

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

Route deployment. Nifty uses OpenFlow [80] and Open vSwitch [81] to deploy the new routes. For instance, to reroute packets sent from node 1 to node 4 through nodes 2 and 3 in Figure 10, the Nifty process on node 1

installs rules on its local Open vSwitch to change the destination MAC address of any packet destined to node 4 to the MAC address of node 2. Whenever node 2 receives a packet with node 4 IP address as its destination, it changes the destination MAC address to node 3 MAC address and sends the packet out. Finally, when node 3 receives a packet with node 4 IP address, it changes the MAC address to node 4 MAC and sends the packet out.

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

To facilitate the implementation of these mechanisms, Nifty offers an API that informs a system running atop Nifty when a partial partition happens and identifies which nodes are on the same side of the network partition and which nodes serve as bridge nodes. Section 9 demonstrates how this information facilitates optimizations in HDFS, VoltDB, and Kafka.

7 IMPLEMENTATION

We implement Nifty in 575 lines of C++ code. A Nifty process runs as a background process on all cluster nodes. A configuration file lists the IP addresses of all cluster nodes. Each Nifty node heartbeats all the nodes listed in the configuration file. The heartbeat message is sent over UDP packets. The default heartbeat period is 200 ms. A node assumes it cannot reach another node if it misses three heartbeats from that node. We note that this is relatively aggressive heartbeating. The goal is for Nifty to discover the partial partition and create alternative routes before the system atop detects the partition with its own heartbeating mechanism. Raft has the shortest hearbeating periods of 250 ms, hence we choose to heartbeat every 200 ms. Nevertheless, the heartbeat period is configurable. Nifty uses the Bellman-Ford routing algorithm to find routes between end nodes. We piggyback the distance vector on every heartbeat message.

Rerouting using Open vSwitchs. An Open vSwitch can be controlled using the OpenFlow standard [80]. OpenFlow allows modifying (i.e., inserting or deleting) the forwarding rules of a switch. Each forwarding entry includes a matching rule and an action list. Matching uses wildcard matching rules on any field in the packet standard headers, including IP and MAC addresses, and protocol and port numbers. If a packet matches a rule, the switch performs the actions associated with that rule. The action list may contain multiple actions that are performed in order. Among the possible actions are packet forwarding to a specific switch port, packet dropping, and modifying fields in a packet such as the source/destination MAC/IP addresses.

After identifying the next hop using distance vector routing, our implementation uses the `ovs-ofctl` tool to manipulate Open vSwitch rules to deploy the routing paths between end nodes.

Limitations. Nifty's approach has two main limitations. First, Nifty relies on an aggressive all-to-all heart beating in order to detect a partial partition in a timely manner. Unfortunately, this approach does not scale to large clusters. We note that Nifty needs to be deployed on the system nodes, not on client nodes, making it an acceptable solution for a large number of systems. Section 8 shows that Nifty can support a cluster of 100 nodes while degrading the system throughput by only 3.5%.

Second, Nifty uses Open vSwitch and MAC address rewriting to create alternative paths between nodes. This limits Nifty's deployablity to a single data center. In our current research we are exploring the design of a fault tolerance technique that can support geographically distributed systems.

Nifty API. To facilitate building system specific optimizations, Nifty provides an API. The API mainly notifies the system when a partial partition happens and exposes the cluster connectivity graph to the system running atop Nifty. The current prototype offers a Java wrapper to simplify integrating Nifty with the systems we used in our evaluation.

Listing 1 shows the main functions in the Nifty API. The API has two groups of functions. The call back functions are triggered when the network state changes. Systems use the query APIs to find the network topology.

The user must override the abstract methods atPartialPartition(), atHealthyNetwork() and atCompletePartition(). Nifty calls these functions when the network state changes. To identify if a partition is complete or partial, Nifty uses depth-first search to traverse the connectivity graph. Depth first search fails to reach all the nodes in a complete partition.

The `getNetworkTopology()` API returns the current topology of the cluster. The `Topology` is represented with an adjacency matrix. The adjacency matrix can be analysed to find the bridge node, and which nodes are on which side of a partition. This information is sufficient to build optimizations that reduce the communication between nodes on different sides of the partition. We show in Section 9 that this basic topology information is sufficient to implement three optimizations in HDFS, VoltDB, and Kafka.

Listing 1. The Nifty API

```
// Call back functions
abstract void atPartialPartition();
abstract void atHealthyNetwork();
abstract void atCompletePartition();

// Query APIs
Topology getNetworkTopology();
NetState getNetworkState();
```

8 EVALUATION

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

Testbed. We conduct our experiments using 40 xl170 nodes at the Cloudlab Utah cluster. Each node has an Intel Xeon E5 10-core CPU, 64 GB of RAM, and a Mellanox ConnectX-4 25 Gbps NIC. To inject a network partition fault, we modify the Open vSwitch rules on the nodes to drop packets between the affected nodes. In all our experiments, we report the average for 30 runs. We note that the standard deviation in all our experiments is lower than 5%. The baseline for our experiments is the performance of the unmodified system without Nifty and without partial partitions.

8.1 Overhead Evaluation

To evaluate Nifty's overhead, we measure its impact on the performance of a synthetic benchmark using iperf [82] and seven data-centric systems (i.e., storage, database, and messaging systems). The systems we selected are:

- HDFS: We deploy HDFS (v3.3.0) on six nodes (one name node and five data nodes) and with a replication level of three. To avoid disk access, we configure data nodes to use tmpfs. We use the HDFS standard benchmark (TestDFSIO). The benchmark reads and writes 1 GB files.
- Kafka: We deploy Kafka (v2.6.0) on five nodes. We distribute the queues (a.k.a., topics) among nodes to balance the load. Each message is replicated on three nodes. We use Kafka's benchmarking tool to generate load on the system. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer.
- ActiveMQ: We deploy ActiveMQ Artemis (v2.15.0) on five nodes with each queue being replicated on two nodes. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer.

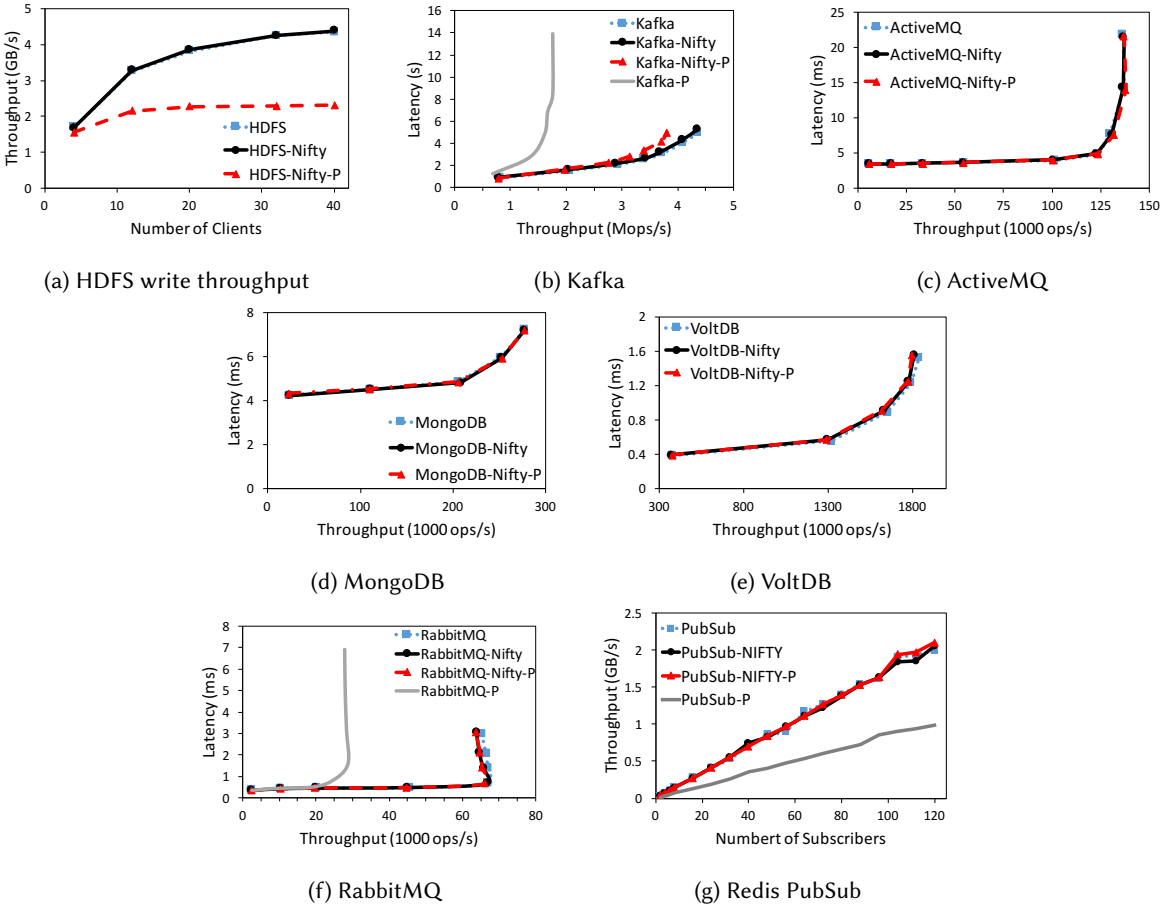


Fig. 11. Nifty's overhead. The average throughput for HDFS (a) and the average throughput vs. average latency for the rest of the systems. (-P) denotes the results with a partial partition.

- MongoDB: We deploy MongoDB (v4.4.1) on six nodes (one config server and five mongod nodes) and with a replication level of three. We discuss our results with the Yahoo benchmark workload B (95% reads and 5% writes) with a uniform distribution [83]. We use 10 million records. The rest of the Yahoo benchmark workloads show similar results.
- VoltDB: We deploy VoltDB (v9.0) on nine nodes, with data sharding enabled and a replication level of three. We use the Yahoo benchmark and the TCP-C benchmark. Figure 11.e shows the throughput-latency curve under Yahoo benchmark workload B (95% reads and 5% writes) with a uniform distribution. The results using the TPC-C benchmark and the Yahoo benchmark workloads A and C with uniform and skewed loads show similar low overhead.
- RabbitMQ: We deploy RabbitMQ (v3.8.2) on three nodes. We use the mirrored mode in which each queue has a leader replica and two backup replicas. We distribute the queue masters among brokers to distribute the load. The experiments use a set of producers and consumers. Each producer sends messages to a dedicated queue and each consumer reads messages from a dedicated queue.

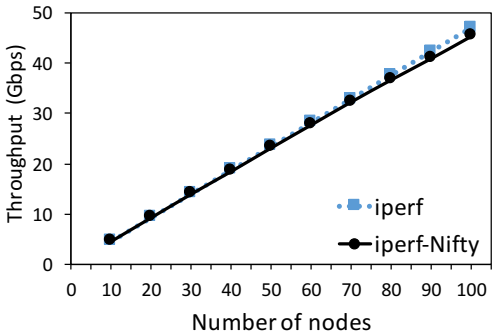


Fig. 12. Scalability evaluation. Average throughput while increasing the number of nodes.

- Redis PubSub: We deploy Redis (v6.2) on three nodes. One publisher connects to one node (i.e., root node) and continuously publishes 1 KB messages to one topic. With Redis PubSub, the root Redis node forwards the published messages to the other Redis nodes. The subscribers connect to the other two Redis nodes.

Results. The baseline for our experiments is the performance of the unmodified system without Nifty and without partial partitions. To evaluate the overhead of Nifty, we compare it to the baseline, the throughput, and average latency of each system with Nifty when there is no partial network partition. We evaluate Nifty with a partial partition in Section 8.2.

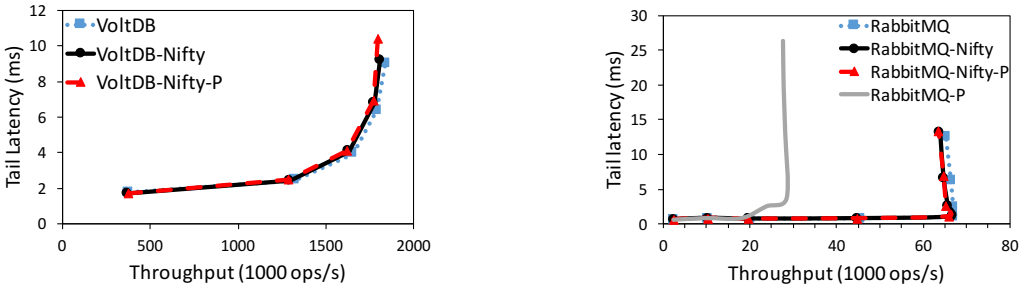
Figure 11 shows the write throughput of HDFS (Figure 11.a) and the throughput-latency curve for Kafka (Figure 11.b), ActiveMQ (Figure 11.c), MongoDB (Figure 11.d), VoltDB (Figure 11.e), RabbitMQ (Figure 11), and Redis PubSub (Figure 11.g). The results show that Nifty does not add noticeable overhead; for all systems, the curves almost completely overlap. This is because Nifty processes exchange a negligible number of packets. Each Nifty process sends a single UDP heartbeat packet every 200 ms to other nodes in the system. Consequently, in the largest deployment of nine nodes, each node sends only 40 packets every second.

Scalability evaluation. Nifty uses all-to-all heart beating to monitor a cluster's connectivity. Consequently, Nifty's overhead increases with the cluster size. To measure Nifty's scalablity, we evaluate its overhead on a 100 m510 nodes at the CloudLab Utah cluster. Each node has an ARMv8 (Atlas/A57) 8-core CPU, 64 GB of RAM, and a Mellanox ConnectX-3 10 Gbps NIC. For this experiment, we limit the throughput of each node to 1 Gbps, as CloudLab can not support a full 10 Gbps connectivity between the 100 nodes we managed to book. To generate network intensive load, we use iperf [82]. Half of the nodes run an iperf server, and the other half run an iperf client. Each client communicates with a single server. Figure 12 shows the aggregate throughput of the iperf servers when deployed with and without Nifty. The figure shows that Nifty's overhead is negligible. When using 100 nodes, Nifty degrades the aggregate throughput by only 3.5%. Nevertheless, this monitoring approach will not scale to clusters with thousands of nodes. We are currently exploring the design of a fault tolerance technique that can scale to larger clusters.

8.2 Handling Partial Partitions

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

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



(a) VoltDB tail latency.

(b) RabbitMQ tail latency.

Fig. 13. Tail latency evaluation. Average throughput vs. 99th percentile of latency.

Figure 11 shows the system performance when the cluster suffers from the partial partition. We notice that all the seven systems are severely effected by the partial partition. ActiveMQ, MongoDB, and VoltDB suffer a complete cluster pause or shutdown when deployed without Nifty. HDFS fails almost all write operations. The VoltDB cluster shuts down because, after detecting the surviving clique, the system misses at least one shard. This confirms our analysis in Section 5.1.

RabbitMQ uses the checking neighbor's views fault tolerance approach. In our deployment, each queue is mirrored on a backup replica. Due to the strong consistency requirement, we configure RabbitMQ to pause in case of partial partition. We deploy RabbitMQ on three nodes. Unfortunately, we could not use a larger RabbitMQ cluster because partial partitions often lead to the pause of the entire RabbitMQ cluster when Nifty is not used (Figure 6). Even with three nodes, partial partitions sometimes lead to pausing two out of three nodes. We discard those results and only include results in which one node pauses. Consequently, our results show the best possible performance of RabbitMQ under partial partitions. Pausing a broker in RabbitMQ leads to more than 50% reduction in throughput (RabbitMQ-P in (Figure 11.f)).

In Redis PubSub, if a partial partition isolates a node that receives new messages from another Redis node, Redis will fail to deliver the message to subscribes connected to the isolated node , leading to 50% reduction in throughput (Figure 11.g).

Kafka uses Zookeeper to monitor cluster nodes. If a partial partition isolates a queue leader from the majority of replicas while Zookeeper runs on a bridge node, Zookeeper will not select a new leader and the entire cluster pauses (Finding 1 in Section 4). To mitigate this, we make sure that Zookeeper falls on one side of the partition. In this case, all the nodes on the other side of the partition that cannot reach Zookeeper are removed from the cluster. In our experiment, the partial partition causes two nodes to pause, which leads to almost a 50% reduction in system throughput (Figure 11.b).

Figure 11 shows that Nifty effectively masks the partial partition, so none of the nodes shut down or pause. Figure 11.a shows the write operation throughput for HDFS. With a replication level of three, each file has replicas on both sides of a partial partition. Consequently, for every 1 GB of data written, up to 2 GB of data are rerouted through the bridge node. This reduces the system throughput by up to 45%. We note that having a partial partition result in a performance degradation is better than a complete system unavailability when HDFS is deployed without Nifty. We present an optimization for HDFS that alleviates this problem in Section 9. For the rest of the systems, during the partial partition, almost 50% of client requests and responses are rerouted through the bridge node. Even so, the system throughput only decreases by 2-6.7% and latency only increases by 3-7.8%. This shows that Nifty can effectively mask partial partitions and is able to utilize remaining connections to reduce the performance impact.

Figure 13 shows the tail latency for VoltDB and RabbitMQ for the same experiments presented in Figure 11. The figure shows the average throughput and the 99th percentile of latency while increasing the load on the system. The figure shows that Nifty increases the 99th percentile latency by up to 6.8% without a partial partition and by 15% under a partial partition failure.

9 EVALUATING THE BENEFITS OF THE NIFTY API

In this section we demonstrate the utility of Nifty's API in improving the performance of three systems. A system using Nifty can be optimized to reduce the amount of data forwarded through bridge nodes. To enable this optimization, systems need to know which nodes are on the same side of the partition and which nodes are bridge nodes. This basic topological information is provided by the Nifty API (Section 7). Systems can use this information to optimize their operations to reduce the communication between nodes on different sides of the partition. The approach to do so is system-specific.

To demonstrate the benefit of using Nifty's API we modify the implementation of three mechanisms. We modify the data placement protocol in HDFS, the processing of multi-shard operations in VoltDB, and the discovery and replication service in Kafka.

9.1 HDFS

HDFS uses chain replication to replicate write operations. Chain replication arranges replicas in a chain where each node passes the write operations to its successor. When selecting three data nodes for a new data chunk, the name node tries to select nodes located on more than one rack as well as balance the number of blocks across nodes. Under partial partitions, large volumes of data can be rerouted through the bridge node. In the worst case, with three-way replication, the same data may traverse bridge nodes twice. Figure 11.a shows that system throughput degrades by up to 45% under partial partitions. During this experiment clients write 48 GB of data, and the bridge node rerouted 39.2 GB of data. Our probabilistic analysis shows that with replication level of 3 the total volume of rerouted traffic through the bridge nodes will be equivalent to 85% of the data writen by the clients.

To improve the system performance under partial partitions we used the Nifty's API to implement three optimizations. The optimizations are implemented by changing 124 lines of code.

- *Optimized chain ordering (Opt.-Chain).* In the worst case, when the system is configured with a replication level of three, a newly written data block may be rerouted twice through the bridge nodes (Figure 14). Our probabilistic analysis shows that in our experimental setup of seven data nodes with one node being a bridge, there is a 17% chance to reroute a block twice through the bridge node.
  We modify the HDFS data placement algorithm to avoid the situation in which data is routed twice through bridge nodes (Figure 14). After the data placement mechanism picks three replicas, we modify the code to query Nifty to get the network topology. We use the network topology to reorder the replicas in a chain such that data is forwarded through bridge nodes at most once.
- *Two replicas (2-Replicas).* Another approach to avoid rerouting the same data twice through bridge nodes is to temporally reduce the replication level to 2 during network partitions. Once the partial partition heals the NameNode can create additional replicas of the affected data chunks.
- *Optimized data placement (One-side).* In this alternative we modified the HDFS data placement algorithm to query Nifty to identify the cluster topology under partial partitions then for any new data chunk allocate three data nodes on the same side of the partition or bridge nodes. This effectively eliminates any data rerouting through the bridge nodes.

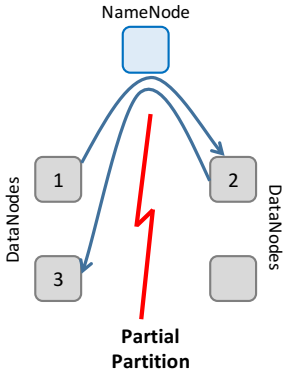


Fig. 14. HDFS worst case rerouting. NameNode choosing the replicas to be on 1,2, and 3 will case the data to move across the partition twice.

We note that these policies affect data written during a partial partition. When the partition heals, Nifty returns to the original replication factor or placement policy. For the 2-Replicas policy, the system will create an additional replica after the partition heals.

Results. We deploy HDFS on eight nodes: one name node, and seven data nodes on the same cluster detailed in Section 8. The partial partition is injected to put three data nodes on each side of the partition and keep one bridge node. Clients run on dedicated machines and use the TestDFSIO benchmark to write to and read 1 GB files. We use the default replication factor of three. We show the average of 30 runs. The maximum standard deviation of these experiments is 4.7%.

Figure 15 shows the system throughput while varying the number of clients. We use the performance of HDFS without a partial partition as a baseline (Baseline in Figure 15). The figure shows that when the partial partition is injected, Nifty, without using any optimization, achieves up to 41% of the baseline throughput. This is mainly because 85% of the client data is rerouted through the bridge node which creates a bottleneck. In the worst case, a client data will be forwarded twice through the bridge node during the replication step. This scenario accounted for 34% of the forwarded data. The Opt.-Chain optimization guarantees that each write operation is rerouted at most once through the bridge node. This optimization reduces the amount of data rerouted through the bridge node to 68% of the client data and achieves up to 59% of the baseline throughput. The 2-replicas optimization further reduces the replication overhead, but also reduces the durability grantees for data written during a partial partition. This optimization achieves 81% of the baseline throughput. Figure 15 shows that the one-side optimization achieves a throughput comparable to the baseline under the partial partition. This is because this optimization avoids rerouting any client data through the bridge nodes.

9.2 VoltDB

In VoltDB, a single server (a.k.a, multi-data-partition initiator or MPI) processes all multi-shard operations. The MPI divides a multi-shard query (e.g., a join) to sub-queries, such that each sub-query targets a single shard. The MPI forwards each sub-query to its shard leader, gathers the intermediate results, performs final query processing, and sends the result to the client.

When deploying VoltDB atop Nifty, if the MPI node is on one side of the partition, a potentially significant volume of intermediate data passes through the bridge node. In our setup, when the MPI is on one side of the

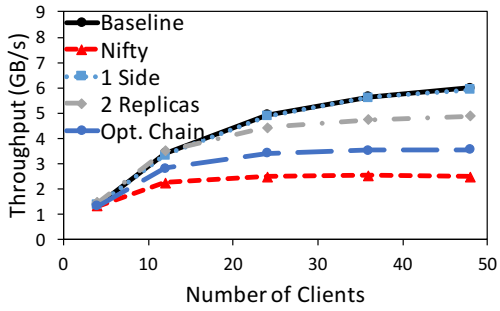


Fig. 15. HDFS write throughput with different optimizations

partition, 50% of the intermediate results are rerouted through the bridge node. This increases operation latency and the load on bridge nodes.

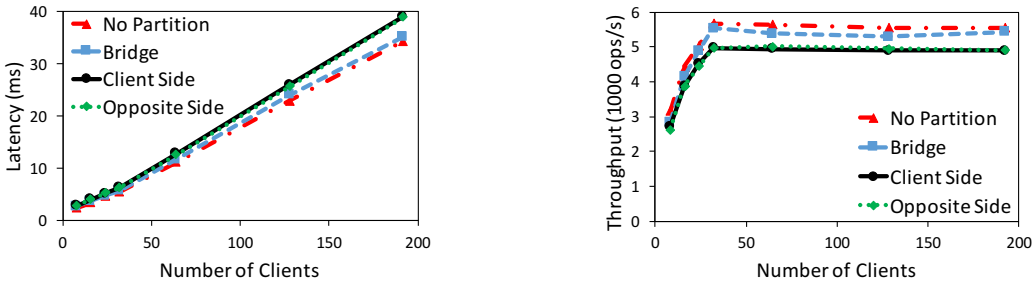
To improve the performance of multi-shard operations, the MPI process can be migrated to a bridge node. This effectively eliminates the need to reroute any traffic for multi-shard queries. We modify VoltDB to use Nifty's API to identify bridge nodes and migrate the MPI to a bridge node. This optimization is implemented in 57 lines of code.

To evaluate this optimization's effectiveness, we evaluate the effect of the MPI's location on system performance. We restrict clients to contacting VoltDB nodes on one side of the partition and compare the system performance of three MPI placements: on clients side of the partition (client side in Figure 16), on the bridge node (bridge), and on the side opposite to the clients (opposite side). Bridge placement represents our optimization.

Setup and Workload. We use the same VoltDB configuration and partial partition setup detailed in the previous sections. Unfortunately, VoltDB has limited support for join queries, so it cannot run standard benchmarks such as TPC-H [84]. In our experiments, we use a simple synthetic benchmark that joins two tables. The benchmark has two sharded tables of 20 fields each. Each field is 50 bytes, leading to approximately 1 KB rows. To use multiple shards, clients issue a range query that joins the two tables on the primary key. The client issues a query with a range that includes four primary keys. Consequently, the query result size is limited to four rows, with a total size of almost 8 KB. We populate the database with 20 GB of data before running the experiments. We report the average and standard deviation for 30 runs. Our results do not include the time taken to migrate the MPI process.

Results. Figure 16 shows the system throughput (a) and the average latency (b) for the three possible MPI placements. During a partial partition fault, placing the MPI on a bridge node decreases the latency by up to 11% and improves throughput by 11% compared to client and opposite side placements. Placing the MPI on a bridge node reduces the number of hops the join query must make before the MPI accumulates all the results and sends the query reply. Furthermore, bridge placement achieves throughput and latency within 4% of VoltDB's performance when there is no partition ("no partition" in Figure 16).

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



(a) Latency (b) Throughput

Fig. 16. The impact of MPI placement on VoltDB's performance. Figure shows the average latency (a) and average throughput (b). Standard deviation was less than 2%.

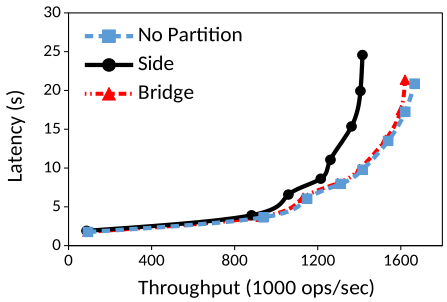


Fig. 17. The impact of the primary replica placement on Kafka's performance. The figure shows the latency-throughput graph. The standard deviation was 3%

9.3 Kafka

Kafka is a replicated event streaming system. Clients follow a discovery protocol to find which replica is the primary replica. Clients send their messages to the primary replica which replicates the message to the other replicas. When deploying Kafka atop Nifty, replication traffic can overwhelm bridge nodes. Using Nifty's API, we modify the discovery and replication protocols to place the primary replica on a bridge node, effectively avoiding any rerouting during message replication. This optimization is implemented in 60 lines of code.

Setup and Workload. We deploy Kafka on five nodes with a replication factor of two. All clients produce and consume messages on a single topic. We create a partial partition that leaves one bridge node and two nodes on each side of the partition.

Results. To evaluate the effectiveness of our optimization we compare the system performance when the primary is on a bridge node to when it is on one side of the partition. We use Kafka's performance when there is no partitions as a base line. Figure 17 shows the throughput-latency graph of the three configurations. The results show that placing the primary on a bridge node decreases the latency by up to 15% and improves throughput by 13% compared to primary being on a side of a partition. This is because placing the primary on the bridge avoids rerouting any data. Furthermore, bridge placement achieves throughput and latency within 3% of Kafka's baseline performance.

10 RELATED WORK

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

Failure studies. A number of previous efforts analyzed failures in distributed systems, including characterizing specific component failures [5, 6, 85–88] and characterizing failures in a specific domain such as HPC [89–91], IaaS clouds [92], data-mining services [93], hosting services [8, 94], data-intensive systems [95–97], and cloud systems [98]. Our work complements these efforts by focusing on failures triggered by partial network partitions.

Yuan et al. [98] conducted a study on 198 general user-reported failures from six distributed systems. Yuan et al. [98] reports 24% of the their failures to be catastrophic failures, while our work shows a much higher percentage in partial network partition failures (76%). Our work also shows that less than 2% of the failures we found are nondeterministic, compared to 26% of general failures they found. This clearly shows that partial network partition failures have more severe effects on the systems than general failures. This also shows that testers should be able to catch partial network partition failures with well written tests since almost all of them are deterministic.

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

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

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

Overlay Networks. Previous efforts explored building scalable and reliable overlay networks. The Resilient Overlay Network (RON) [99] explored an overlay design that recover from path outages in the internet. Unlike RON, Nifty focuses on partial partitions in data center networks, works in the MAC layer, and targets millisecond convergence times. Peer-to-peer systems like Chord [100] and Pastry [101] use overlay networks to route requests in peer-to-peer distributed storage systems. BDS [102] uses overlay network routing for optimizing inter-datacenter data replication. Other works [103] investigate using overlay networks for better performance in content delivery. Nifty differs from these systems in its purpose, which is to keep a cluster fully connected in the case of partial partitions.

SDN for overlay networks Software-defined networking capabilities have been used to engineer traffic and optimize system operations. Google Andromeda [104] uses SDN for Google Cloud Platform's network virtualization stack.

Other efforts focused on using SDN to optimize certain systems. SDN was used for performing different network measurement tasks like QoS measurements or anomaly detection [105, 106]. Some works have used SDN to implementing an in-network stateful firewalls [107]. Other works [108, 109] show the use of SDN to create load balancers to optimize key-value stores and distributed Memcached deployments. SDN was also used for key-value-based routing [110, 111]. Nifty is similar in spirit to these systems, as we use Open vSwitch capabilities to implement an overlay. Our goal however is different: to improve systems fault-tolerance by masking partial network partitions.

11 CONCLUSION AND FUTURE WORK

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

This is the first work to focus on partial network partitioning faults and present an in-depth analysis of system failures triggered by these faults. We identify characteristics that can facilitate better test design. Our findings highlight that focused design reviews can identify vulnerabilities early in the design process. We identified flaws in five common designs of core system techniques. Investigating alternative designs that tolerate partial partitions is a high impact research area that needs further exploration.

We dissect the implementation of nine popular systems and study their fault tolerance techniques. In doing so, we identify four modern approaches for tolerating partial partitions. Unfortunately, all implemented fault tolerance techniques have severe shortcomings.

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

In our future work we will focus on two directions. The first direction is to investigate the design of a scalable fault tolerance technique. We will explore techniques to identify partial partitions and find alternative routes without continuously performing all-to-all heart-beating. The second direction is to reduce the reliance of Nifty on OpenvSwitch by exploring the design of an alternate fault tolerance technique, which does not require modifying OpenvSwitch rules. We plan to build a communication library that masks partial partitions.

ACKNOWLEDGMENT

We thank Bernard Wong, Trevor Brown, Omid Abari, Ali Mashtizadeh, and Khuzaima Daudjee for their insightful feedback. We thank Joslin Goh for her feedback on our probabilistic analysis of VoltDB's failure probability. This research was supported by an NSERC Discovery grant, Canada Foundation for Innovation (CFI) grant, NSERC Collaborative Research and Development (CRD) grant, and a Waterloo-Huawei Joint Innovation Lab grant. Ahmed is supported by an IBM PhD fellowship.

REFERENCES

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

[2] Data Center: Load Balancing Data Center, Solutions Reference Nework Design. Technical report, Cisco Systems, Inc., 2004. ([n. d.]).

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

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

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

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

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

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

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

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

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

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

[13] Seth Gilbert and Nancy Lynch. 2002. Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services. *SIGACT News* 33, 2 (June 2002), 51–59. DOI: http://dx.doi.org/10.1145/564585.564601

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

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

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

[17] Barbara Liskov and James Cowling. 2012. *Viewstamped Replication Revisited*. Technical Report MIT-CSAIL-TR-2012-021. MIT.

[18] RabbitMQ message broker. https://www.rabbitmq.com. ([n. d.]). Accessed: June 2021.

[19] VoltDB In-Memory Database Platform. https://www.voltdb.com/. ([n. d.]). Accessed: June 2021.

[20] The Ceph object store. https://ceph.io/. ([n. d.]). Accessed: June 2021.

[21] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. 2020. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 351–368. https://www.usenix.org/conference/osdi20/presentation/alfatafta

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

[23] bnx2 Cards Intermittantly Going Offline. https://www.spinics.net/lists/netdev/msg152880.html. ([n. d.]). Accessed: June 2021.

[24] CloudFlare Blog: A Byzantine failure in the real world. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/. ([n. d.]). Accessed: June 2021.

[25] Google Cloud Networking Incident #18003. https://status.cloud.google.com/incident/cloud-networking/18003. ([n. d.]). Accessed: June 2021.

[26] Lyft Engineering: Operating Apache Kafka Clusters 24/7 Without A Global Ops Team. https://eng.lyft.com/operating-apache-kafka-clusters-24-7-without-a-global-ops-team-417813a5ce70. ([n. d.]). Accessed: June 2021.

[27] Datadog: Learning from AWS failure. https://www.datadoghq.com/blog/gray-aws-failures/. ([n. d.]). Accessed: June 2021.

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

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

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

[31] Observability in Paxos clusters. https://davecturner.github.io/2017/08/18/observability-in-paxos.html. ([n. d.]). Accessed: June 2021.

[32] Partial network partitions and obstacles to innovation. https://rachelbythebay.com/w/2012/02/16/partition/. ([n. d.]). Accessed:June 2021.

[33] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. ([n. d.]). Accessed: June 2021.

[34] Healthchecking is Not Transitive. https://www.robustperception.io/healthchecking-is-not-transitive. ([n. d.]). Accessed: June 2021.

[35] Cluster broken after switches upgrade. https://github.com/elastic/elasticsearch/issues/9495. ([n. d.]). Accessed: June 2021.

[36] Using map output fetch failures to blacklist nodes is problematic. https://issues.apache.org/jira/browse/MAPREDUCE-1800. ([n. d.]). Accessed: June 2021.

[37] Elasticsearch: Distributed search & Analytics. https://www.elastic.co/products/elasticsearch. ([n. d.]). Accessed: June 2021.

[38] MongoDB: The database for modern applications. https://www.mongodb.com/. ([n. d.]). Accessed: June 2021.
[39] The Apache Hadoop project. http://hadoop.apache.org/. ([n. d.]). Accessed: June 2021.
[40] Apache HBase. https://hbase.apache.org/. ([n. d.]). Accessed: June 2021.
[41] Apache Mesos. http://mesos.apache.org/. ([n. d.]). Accessed: June 2021.
[42] Hazelcast | The Leading In-Memory Computing Platform. https://hazelcast.com/. ([n. d.]). Accessed: June 2021.
[43] Kafka: A distributed streaming platform. https://kafka.apache.org/. ([n. d.]). Accessed: June 2021.
[44] MooseFS: Distributed file system. https://moosefs.com/. ([n. d.]). Accessed: June 2021.
[45] ActiveMQ: Flexible & Powerful Open Source Multi-Protocol Messaging. http://activemq.apache.org/. ([n. d.]). Accessed: June 2021.
[46] Dkron: A distributed Cron service. https://dkron.io/. ([n. d.]). Accessed: June 2021.
[47] 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. ([n. d.]). Accessed: June 2021.
[48] Possible data loss when RS goes into GC pause while rolling HLog. https://issues.apache.org/jira/browse/HBASE-2312. ([n. d.]). Accessed: June 2021.
[49] Partial network partition and retries. https://github.com/elastic/elasticsearch/issues/6105. ([n. d.]). Accessed: June 2021.
[50] Hazelcast: the Leading In-Memory Data Grid. https://hazelcast.com/. ([n. d.]). Accessed: June 2021.
[51] Redis: in-memory data structure store. https://redis.io/. ([n. d.]). Accessed: June 2021.
[52] A. Herr. 2016. *Veritas Cluster Server 6.2 I/O Fencing Deployment Considerations.* Technical Report. Veritas Technologies.
[53] Two primaries with network partitioned replica set (non-transient). https://jira.mongodb.org/browse/SERVER-2544. ([n. d.]). Accessed: June 2021.
[54] Synchronisation causes crash in duplicated master #1006. https://github.com/rabbitmq/rabbitmq-server/issues/1006. ([n. d.]). Accessed: June 2021.
[55] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*. Association for Computing Machinery, New York, NY, USA, 8–17. DOI: http://dx.doi.org/10.1145/62546.62549
[56] Partial network partitioning leads to cluster unavailability. https://github.com/elastic/elasticsearch/issues/43183. ([n. d.]). Accessed: June 2021.
[57] Apache Zookeeper. https://zookeeper.apache.org/. ([n. d.]). Accessed: June 2021.
[58] ZooKeeper Recipes and Solutions. https://zookeeper.apache.org/doc/current/recipes.html. ([n. d.]). Accessed: June 2021.
[59] ActiveMQ cluster blocks indefinitely in the presence of partial network partition. https://issues.apache.org/jira/browse/AMQ-7064. ([n. d.]). Accessed: June 2021.
[60] Kafka leader election doesn't happen when leader broker port is partitioned off the network. https://issues.apache.org/jira/browse/KAFKA-8702. ([n. d.]). Accessed: June 2021.
[61] Giorgos Myrianthous. Kafka No Longer Requires ZooKeeper. https://towardsdatascience.com/kafka-no-longer-requires-zookeeper-ebfbf3862104. ([n. d.]). Accessed: June 2021.
[62] Colin McCabe. Apache Kafka Needs No Keeper: Removing the Apache ZooKeeper Dependency. https://www.confluent.io/blog/removing-zookeeper-dependency-in-kafka/. ([n. d.]). Accessed: June 2021.
[63] MapReduce Ticket 4832. https://issues.apache.org/jira/browse/MAPREDUCE-4832. ([n. d.]). Accessed: June 2021.
[64] Mesos-1529: Handle a network partition between Master and Slave. https://issues.apache.org/jira/browse/MESOS-1529. ([n. d.]). Accessed: June 2021.
[65] Disconnect between coordinating node and shards can cause duplicate updates or wrong status code #9967. https://github.com/elastic/elasticsearch/issues/9967. ([n. d.]). Accessed: June 2021.
[66] Mirrored queue crash with out of sync ACKs. https://github.com/rabbitmq/rabbitmq-server/issues/749. ([n. d.]). Accessed: June 2021.
[67] KAFKA-3686: Kafka producer is not fault tolerant. https://issues.apache.org/jira/browse/KAFKA-3686. ([n. d.]). Accessed: June 2021.
[68] Partial network partition and retries #6105. https://github.com/elastic/elasticsearch/issues/6105. ([n. d.]). Accessed: June 2021.
[69] HDFS-1384: NameNode should give client the first node in the pipeline from different rack other than that of excludedNodes list in the same rack. https://issues.apache.org/jira/browse/HDFS-1384. ([n. d.]). Accessed: June 2021.
[70] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.
[71] LogCabin. https://github.com/logcabin/logcabin. ([n. d.]). Accessed: June 2021.
[72] How does VoltDB handle partial network partitions? https://www.voltdb.com/resources/transaction-consistency-faq#net. ([n. d.]). Accessed: June 2021.
[73] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms.* MIT press.
[74] Coen Bron and Joep Kerbosch. 1973. Algorithm 457: Finding All Cliques of an Undirected Graph. *Commun. ACM* 16, 9 (Sept. 1973), 575–577. DOI: http://dx.doi.org/10.1145/362342.362367
[75] Faulty recovery caused by partial network partitions. https://github.com/elastic/elasticsearch/pull/8720. ([n. d.]). Accessed: June 2021.

[76] Designing Highly Available Mesos Frameworks. http://mesos.apache.org/documentation/latest/high-availability-framework-guide/. ([n. d.]). Accessed: June 2021.
[77] Wait on shard failures. https://github.com/elastic/elasticsearch/issues/14252. ([n. d.]). Accessed: June 2021.
[78] Deep Medhi and Karthik Ramasamy. 2017. *Network routing: algorithms, protocols, and architectures*. Morgan Kaufmann.
[79] Dimitri P Bertsekas, Robert G Gallager, and Pierre Humblet. 1992. *Data networks*. Vol. 2. Prentice-Hall International New Jersey.
[80] OpenFlow Switch Specification, Version 1.5.1 (ONF TS-025). Open Networking Foundation, 2015. ([n. d.]).
[81] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. 117–130.
[82] iPerf: The ultimate speed test tool for TCP, UDP and SCTP. https://iperf.fr/. ([n. d.]). Accessed: June 2021.
[83] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. DOI: http://dx.doi.org/10.1145/1807128.1807152
[84] 2018. TPC-H BENCHMARK (Decision Support) Standard Specification. Transaction Processing Performance Council. Revision 2.18.0.
[85] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 193–204.
[86] Robert Birke, Ioana Giurgiu, Lydia Y Chen, Dorothea Wiesmann, and Ton Engbersen. 2014. Failure analysis of virtual and physical machines: patterns, causes and characteristics. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 1–12.
[87] Daniel Ford, François Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. (2010).
[88] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. 2008. Are disks the dominant contributor for storage failures?: A comprehensive study of storage subsystem failure characteristics. *ACM Transactions on Storage (TOS)* 4, 3 (2008), 7.
[89] Nosayba El-Sayed and Bianca Schroeder. 2013. 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)*. IEEE, 1–12.
[90] Yinglung Liang, Yanyong Zhang, Anand Sivasubramaniam, Morris Jette, and Ramendra Sahoo. 2006. Bluegene/l failure analysis and prediction models. In *International Conference on Dependable Systems and Networks (DSN'06)*. IEEE, 425–434.
[91] Bianca Schroeder and Garth Gibson. 2009. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing* 7, 4 (2009), 337–350.
[92] Theophilus Benson, Sambit Sahu, Aditya Akella, and Anees Shaikh. 2010. A First Look at Problems in the Cloud. *HotCloud* 10 (2010), 15.
[93] Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Haibo Lin, Haoxiang Lin, and Tingting Qin. 2015. An empirical study on quality issues of production big data platform. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*. IEEE Press, 17–26.
[94] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. 2016. Why does the cloud stop computing?: Lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. ACM, 1–16.
[95] Ariel Rabkin and Randy Howard Katz. 2012. How hadoop clusters break. *IEEE software* 30, 4 (2012), 88–94.
[96] 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. 2014. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–14.
[97] Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. 2013. A characteristic study on failures of production distributed data-parallel programs. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 963–972.
[98] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U Jain, and Michael Stumm. 2014. 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)*. 249–265.
[99] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. 2001. Resilient Overlay Networks. 35, 5 (Oct. 2001), 131–145. DOI: http://dx.doi.org/10.1145/502059.502048
[100] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM Comput. Commun. Rev.* 31, 4 (Aug. 2001), 149–160. DOI: http://dx.doi.org/10.1145/964723.383071
[101] Antony Rowstron and Peter Druschel. 2001. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware 2001*, Rachid Guerraoui (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 329–350.
[102] Yuchao Zhang, Junchen Jiang, Ke Xu, Xiaohui Nie, Martin J. Reed, Haiyang Wang, Guang Yao, Miao Zhang, and Kai Chen. 2018. BDS: A Centralized near-Optimal Overlay Network for Inter-Datacenter Data Replication. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 10, 14 pages. DOI: http://dx.doi.org/10.1145/3190508.3190519

[103] John Byers, Jeffrey Considine, Michael Mitzenmacher, and Stanislav Rost. 2002. Informed Content Delivery across Adaptive Overlay Networks *(SIGCOMM '02)*. Association for Computing Machinery, New York, NY, USA, 47–60. DOI: http://dx.doi.org/10.1145/633025.633031

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

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

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

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

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

[109] Alex FR Trajano and Marcial P Fernandez. 2015. Two-phase load balancing of In-Memory Key-Value Storages through NFV and SDN. In *2015 IEEE Symposium on Computers and Communication (ISCC)*. IEEE, 409–414.

[110] I. Kettaneh, A. Alquraan, H. Takruri, S. Yang, A. S. Dusseau, R. Arpaci-Dusseau, and S. Al-Kiswany. 2019. The Network-Integrated Storage System. *IEEE Transactions on Parallel and Distributed Systems* (2019). DOI: http://dx.doi.org/10.1109/TPDS.2019.2938158

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

APPENDIX A: THE PROBABILITY OF VOLTDB CLUSTER SHUTDOWN

We consider a VoltDB cluster with N nodes. The cluster stores S shards with a replication factor of R. When a partial network partition happens, VoltDB identifies the surviving clique and all the nodes that are not part of the clique shutdown. We denote the number of nodes that shutdown due to a partial partition as F (Since F is not in the surviving clique then $F < \frac{N}{2}$), leaving the system with $(N - F)$ surviving nodes.

Assumptions. We assume that:

(1) The system selects R nodes to hold the replicas of a given shard using a uniform random distribution.
(2) Shard placement is independent of other shards locations.
(3) Each node has enough capacity to store all the shards.

VoltDB will shut down if the surviving clique does not have all the shards, i.e., if the F failed nodes contain all the R replicas of any of the shards, then VoltDB shuts down. In other terms, the VoltDB cluster will survive a partial partition if every shard has at least one replica in the surviving clique.

Step I. Single Shard Probability. Consider the case of a system with a single shard. The system will survive in all cases in which the surviving clique has at least a single replica of the shard. To compute the probability a system will survive a partial partition, we will compute the number of possible replica placements in the cluster, then compute how many of those placements would fail when losing F nodes. Finally, we will use these two numbers to compute the probability a system survives a partial partitioning fault.

Number of possible combinations to place a shard. The system selects R nodes to hold the replicas for a shard. The selection is without replacement since no two copies of the shard can be placed on the same node, and order of selected nodes is not important.

Number of possible combinations for placing a shard is:

$$C(N, R) = \frac{N!}{(N - R)! \times R!} \tag{1}$$

If F nodes fail, the number of combinations in which all the replicas of the shard are on the F failed nodes is (this is again without replacement and ordering is not important)

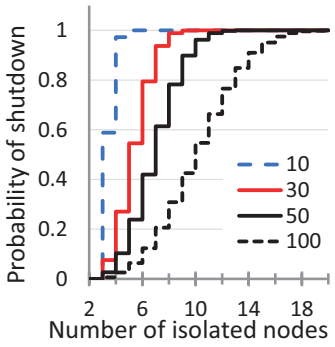


Fig. 18. Probability of a VoltDB system shut down.

$$C(F,R) = \frac{F!}{(F-R)! \times R!} \tag{2}$$

The probability the system shutdown when F nodes fail is

$$\begin{aligned} P(single\_shard\_shutdown) &= \frac{C(N,R)}{C(F,R)} \\ &= \frac{\frac{N!}{(N-R)!\times R!}}{\frac{F!}{(F-R)!\times R!}} = \frac{F!(N-R)!}{N!(F-R)!} \end{aligned} \tag{3}$$

And the probability of a system surviving a partial partition that shuts down F nodes is

$$P(single\_shard\_surviving) = 1 - \frac{F!(N-R)!}{N!(F-R)!} \tag{4}$$

Step II. Multi-shard probability. Assuming that shards are placed independently, the probability of a system with S shard surviving a partial partition with F nodes shutting down is:

$$P(system\_surviving) = \left(1 - \frac{F!(N-R)!}{N!(F-R)!}\right)^S \tag{5}$$

The probability the system shuts down is

$$P(system\_shutdown) = 1 - \left(1 - \frac{F!(N-R)!}{N!(F-R)!}\right)^S \tag{6}$$

Example. We used this formula to compute the probability of failure of VoltDB on different cluster sizes. The number of shards VoltDB allocates to nodes is equal to the number of cores. Figure 18 shows the probability of VoltDB shutting down for different cluster sizes, with replication level of 3, and assuming nodes with 32 CPU cores. The figure shows that isolating only 10% of the nodes leads to over 50% probability of shutting down the entire cluster, and isolating only 20% of the nodes leads to a staggering 90% chance for a complete cluster shutdown.

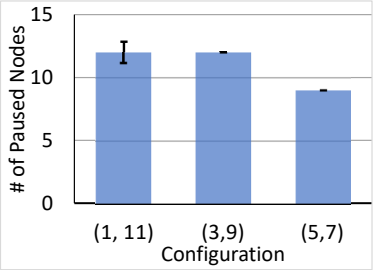


Fig. 19. The median number of paused nodes in a cluster of 15 nodes. In all runs, 3 node are unaffected by the partition. The notation (i, j) shows the number of nodes on each side of the partition.

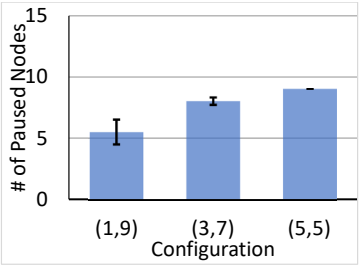


Fig. 20. The median number of paused nodes in a cluster of 15 nodes. In all runs, 5 node are unaffected by the partition. The notation (i, j) shows the number of nodes on each side of the partition.

APPENDIX B: THE IMPACT OF PARTIAL PARTITIONS ON RABBITMQ

RabbitMQ's has two main policies for handling partial partitions. The first policy changes a partial partition to a complete partition which may lead to multiple inconsistent copies of the data. The second is the *pause* policy which preserves data consistency but may lead to pausing the entire system or the majority of its nodes.

To determine how many nodes pause when using the pause policy, we conduct an experiment in which we deploy a 15-node RabbitMQ cluster, introduce a partial partition, and observe how many nodes pause. In all experiments, we inject a partition such that one node remains unaffected and able to reach all nodes. Figures 7, 19, and 20 show the median number of paused nodes under various partition configurations. We run each configuration 30 times. Note that the maximum number of nodes that can pause is 14, 12, and 10 in Figures 7, 19, and 20, respectively, because the rest of the nodes are bridge nodes that can reach all the nodes in the cluster. Surprisingly, under all partial partition scenario a significant number of the affected nodes pause. Our investigation of this failure scenario reveals that nodes declare another node unreachable after missing its heartbeats for a *timeout* period. In RabbitMQ, the default timeout period is 1 minute, which gives enough time for many nodes to detect the partition and pause. We experimented with significantly shorter timeout periods, but that caused some nodes to prematurely declare that other nodes had failed, even without a partial partition.