

A Fault Injection Tool for Testing Distributed System with Network Faults

by

Seba Tayser Khaleel

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

Waterloo, Ontario, Canada, 2023

© Seba Tayser Khaleel 2023

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners. I understand that my thesis may be made electronically available to the public.

Abstract

Modern systems are complex, they include hundreds of components that implement complex protocols such as scheduling, replication, membership, resource management, client access, and security. These systems are expected to offer high availability and to preserve data stored in them despite environment faults. Testing is the primary approach for improving system reliability. Testing against environment faults such as hardware failures, memory corruption, and network problems is complicated since environment faults can happen at any time in the system lifetime, at any component, and at any step in a complex protocol.

In this work, we focus on testing for network partitioning faults. We build PPATT, a fault injection testing tool that injects network partition faults between components. To reduce the number of test scenarios that need to be considered, we implement two techniques to focus testing on components that communicate during an operation. We verify the tool through reproducing four catastrophic failures from two widely popular systems: Spark and Kafka. To demonstrate the benefits of our system, we test three systems using PPATT: Flink, Hazelcast, and ActiveMQ Artemis. Our testing discovers three failures in these systems. All these failures are due to design flaws.

Acknowledgements

I would like to thank my beloved family, who have stayed with me through my journey, who have been there for me and supported me from the very beginning, there are not enough words to describe how grateful and sincerely thankful I am for every moment you have been there for me, and you will always remain in my heart.

I would like to thank my supervisor Professor Samer Al-Kiswany, who have been keen and passionate to teach me, to lead me into a brighter future and to enlighten my path. I sincerely thank you from the depths of my heart and I am honored to receive such knowledge and guidance from you. Your guidance and enlightenment have taken me through a path full of knowledge, information, and experience which defined what I became, and I believe your influence will keep me moving towards great goals and achievements.

I would like to thank my dear friends and colleagues, who we have ridden along and shared every joyful and tearful moment, I am grateful to have such company in my life and I will never forget your willingness to help me, to encourage me and to believe in me.

Dedication

This is dedicated to the one I love.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	viii
1 Introduction	1
2 Background and Related Work	5
2.1 Network Partition Faults	5
2.2 Network Partition Failures	6
2.3 Related Work	7
2.3.1 Memory Testing	7
2.3.2 Security Testing	7
2.3.3 Network Testing	8
2.3.4 Fault Injection Testing	8

3	Design	11
3.1	System Under Test module	12
3.2	Capture Connections Module	13
3.2.1	Capturing Communication in a Cluster	14
3.2.2	Capturing SUT Communication	14
3.3	Fault Injection Module	15
3.4	Putting It All Together	16
3.5	PPATT Extensibility	16
3.6	Implementation	17
4	Tool Verification	18
4.1	Failure of Spark Standalone Cluster Manager	18
4.2	Failure of Mesos Cluster Manager	20
4.3	Failure of ZooKeeper Coordination Mechanism	22
5	Demonstration of the Tool Capabilities	24
5.1	Hazelcast	24
5.1.1	Map Read and Write Operation	25
5.1.2	Count down latch and Fenced locks Operation	28
5.2	Apache Flink	29
5.3	ActiveMQ Artemis	31
6	Conclusion and Future Work	33
	References	34

List of Figures

1.1	Partial Network Partition	2
2.1	Network partitioning types	6
3.1	PPATT Architecture	12
4.1	Spark Standalone Architecture	19
4.2	Spark with Mesos Architecture	21
4.3	Kafka Architecture	23
5.1	Hazelcast Architecture	27
5.2	Flink Architecture	30
5.3	ActiveMQ Artemis Architecture	32

Chapter 1

Introduction

Modern distributed systems [42, 26, 49, 45] should be reliable, highly available [27, 19] and durable [47, 16, 18]. Building a reliable distributed system is challenging due to failures that impact the system hardware or software [48]. Among the most complex failures that are challenging to tolerate are failures caused by faults that are external to the system code. Examples of such faults are memory corruptions, disk corruption, and network failures. Testing and debugging the system tolerance to external faults using standard testing frameworks is complicated. These faults can occur at any point during the system operation. Furthermore, these faults can impact any component or stored data, for instance, network faults can impact communication between any two subgroups of nodes in the cluster, and disk corruption can impact data or metadata objects. The impact of a fault depends on which component and data experience the fault, and when.

To improve systems resilience to these faults, developers resort to fault injection testing. These testing framework includes a fault injection mechanism to mimic external failures by injecting them during system tests. One example is memory fault injection [14], in which corrupted data is injected into the system memory to test its ability to detect and handle such fault. One area that has not received significant attention for fault injection testing is network failures, and more precisely, network partitioning failures [44].

Network partitioning have detrimental consequences, as they can lead to catastrophic system failures. For instance, network partitions led to service outages at Cloudflare [17], Google [24], Lyft [11], and Amazon AWS [20].

Alquraan et al. [10] study network partitioning failures from 25 diverse production systems and report that network partitions can lead to data loss, data corruption, stale

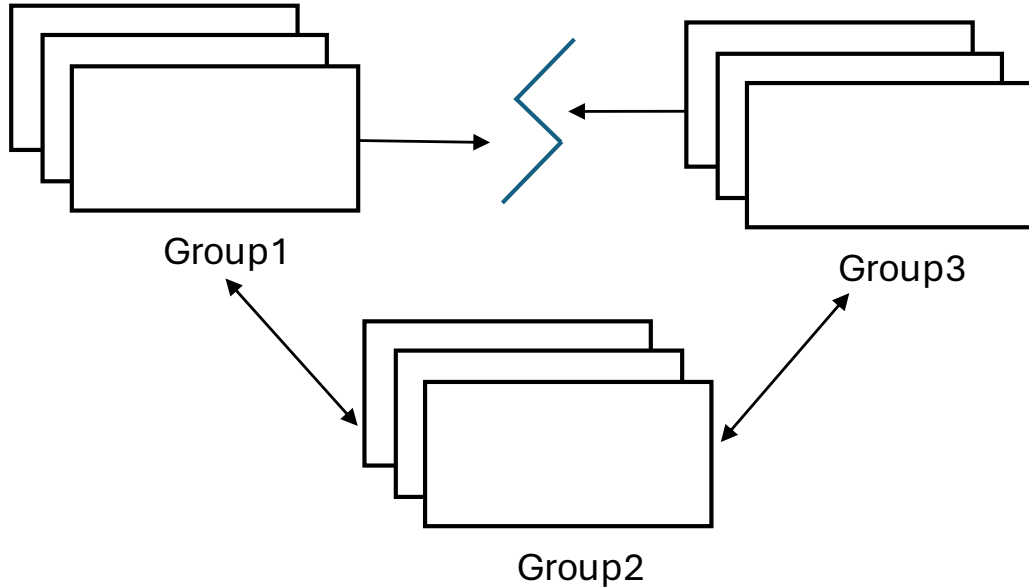


Figure 1.1: Partial Network Partition

reads, double locking, and system crashes. Alquraan et al. identify a peculiar type of network partitions: partial partitions. Partial partitions disrupt the communication between a subset of nodes in the cluster. For instance, figure 1.1 shows a cluster that is impacted by a partial network partition that leads to the division of nodes into three groups (Group1, Group2, and Group3) such that two groups Group1 and Group3 are disconnected while Group2 can communicate with both Group1 and Group3. In such partitions the system will reach a confusing state in which some nodes (e.g., Group1 and Group3 in figure 1.1) run fault tolerance mechanisms while others (e.g., Group2) do not run any fault tolerance mechanism. This confusing state is poorly understood and tested.

Recently, Alfatafta et al. [8] conduct a study of partial partitions in production systems. Partial partitions lead to catastrophic failures such as data loss, data corruption and system crashes. The study shows that these failures are easy to manifest and are caused by design flaws. The two previous studies [10, 8] discuss ways to improve system tolerance to such faults and highlight that better testing with a framework that can inject network partitions is key to improve systems' fault tolerance. Unfortunately, there is no testing tool that can readily test against this type of network partitioning fault.

In this work, I present PPATT-Partial Partition Auto Testing Tool-, a fault injection testing tool for network partitioning failures. PPATT is able to inject complete as well as partial network partition faults. This tool runs in the background of current system's tests and injects network partitions to verify the reliability of a system.

Designing such a tool is challenging because the number of possible partitions and the points of time when they occur during a system operation significantly increases the number of test cases that must be tested. However, testing against all possible fault scenarios as part of the development cycle is not possible. To limit the number of test cases, PPATT does the following two steps. First, it eliminates any communication happening outside the system under test. Second, it limits testing to the system components that communicate during a unit test. This is achieved through building a monitoring tool that captures the system interactions. The tool only tests the system while injecting network partitions between the system components that interact during system operation. These two steps significantly reduce the number of test cases.

I implement a prototype of the tool and verify its operation through reproducing network partitioning related failures on two systems, Spark and Kafka. Spark is a data analytics tool. I reproduce three complex failures related to its scheduling mechanism. I test Spark with two schedulers, Standalone and Mesos, with a word count application. The tool reproduces catastrophic failures that cause the system to hang after injecting a partition. Kafka is a message queuing system, I reproduce one failure related to cluster management logic using ZooKeeper. This failure leads to a system pause until the partition is healed. PPATT successfully reproduces these four failures and confirms the reported failures and their impact.

I demonstrate the benefit of PPATT by testing three diverse systems: Apache Flink; a streaming system, Hazelcast; a distributed data storage system, and ActiveMQ Artemis; a message queuing system. I test their operation while using my tool to inject network partitioning failures.

I test Flink's standalone resource manager while running a word count application. I configure the system to use its fault tolerance and recovery mechanism to detect failures and restart operations. The tool detects one failure related to Flink's resource management logic. The failure causes Flink resource manager to unsuccessfully retry allocating resources multiple times before giving up and throwing an exception.

I test Hazelcast with three data structures: maps, locks, and count down latch. Hazelcast includes fault tolerance mechanisms for complete and partial partitions. I configure the system to use these fault tolerance mechanisms. The tool detects one failure in the system in the presence of partial partitions. This failure leads to failing operations for a

subset of clients.

I test ActiveMQ Artemis and configure it to replicate the data on three nodes. The tool detects one failure that puts the system in an erroneous state in which two nodes act as active replicas. These two replicas do not sync their data leading to data inconsistency which persists even after fixing the partition.

PPATT finds three bugs in these three systems and I report these bugs to the developing community of these systems [3, 2, 1]. The detected failures are catastrophic as they cause systems' unavailability, data inconsistency, failing operations, and violating systems' guarantees. They are also easy to reproduce in a small cluster of four nodes. All failures are often due to design flaws and are deterministic.

The rest of this thesis is organized as follows. We present an overview of network partitioning faults and failures, and testing tools in chapter 2. We present PPATT design and implementation in chapter 3, our verification in chapter 4 and a demonstration of our tool capabilities in chapter 5 . We conclude in chapter 6.

Chapter 2

Background and Related Work

In this section I present a background related to network partition faults and failures.

2.1 Network Partition Faults

Network partitioning is a network fault that disrupts the communication between nodes in a system. Previous study [10] identifies three types of network partitions. Figure 2.1 shows these three partitions. In a Complete partition as shown in figure 2.1.a, a system is divided into two disconnected groups. Partial partition affects some, but not all, nodes in the system. Figure 2.1.b shows an example of a partial partition in which Group 1 and 3 of nodes can not communicate with each other. While Group 2 does not notice a disruption in the cluster. Simplex partition as shown in figure 2.1.c, traffic flows in one direction but not in the another. Alquraan et al. [10] report that complete and partial partitions are common in production systems while simplex partitions are rare.

Network partitions are a common occurrence in distributed systems. Turner et al. [44] find that network partitions happen approximately every four days within the California-wide CENIC network, Google reported 40 network partitions in two years [25] and according to Microsoft's findings, network partitions account for 70% of its reported downtime [22].

These network partitions happen because of hardware failures [22], software issues [33], network congestion, or temporary disruptions [44]. Each scenario presents unique challenges for system reliability, data consistency, and fault tolerance. Addressing network partitions involves implementing resilient networking protocols and strategies to minimize

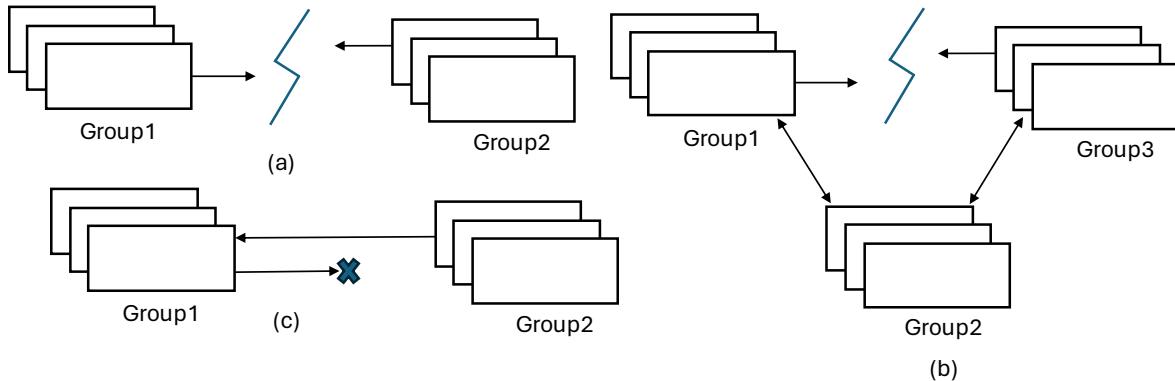


Figure 2.1: Network partitioning types

their impact on system integrity, ensuring continuous connectivity, and efficient data transmission even in the presence of these disruptive events. Understanding and mitigating the effects of network partitions remain crucial in maintaining the stability and reliability of complex networked systems.

2.2 Network Partition Failures

Alquraan et al. [10] study the impact of network partitions on distributed systems. They find that complete network partitions cause catastrophic failures (e.g., data loss, data corruption, data unavailability, and broken locks), with data loss being the most popular among them. The majority of failures that happen due to network partitions are silent and can cause a permanent damage in the system that remains after the partition is healed. Interestingly, these failures are easy to manifest; they need few events to happen and are deterministic. One interesting finding of this study [10] is that better testing can alleviate this problem, where the majority of the failures can be reproduced and tested using a small cluster of three nodes and with a testing framework that can inject network failures.

Alfatafta et al. [8] study the impact of partial network partitions on cloud systems. They show that partial network partitioning faults result in silent failures and lead to severe consequences such as data loss and corruption. The study shows that partial partitions affects core system functionalities like leader election and replication. Surprisingly, Alfatafta et al. [8] find that the majority of failures are because of design flaws, and highlight that

system developers do not expect networks to have these faults. The study suggests three methods to enhance system resiliency to partial network partition faults, which are better testing, concentrated design reviews and constructing a comprehensive fault-tolerant communication layer.

Previous studies [10, 8] indicate that better testing is key for improving systems resiliency to network partitions. Unfortunately, there is no testing framework that can readily test systems and inject network partitioning faults. This work aims to address this gap.

2.3 Related Work

Testing tools play a fundamental role in the software development life cycle, ensuring the quality, reliability, and functionality of software applications. These tools are designed to automate the process of identifying logical and development bugs. Through automated test scripts, testing tools systematically execute test cases, enabling fast and repetitive assessment of software applications. Additionally, testing tools provide detailed analysis features, enabling developers and quality analysis teams to gain valuable insights into the application's performance metrics, code coverage, and overall stability.

2.3.1 Memory Testing

Memory testing is the process of examining how applications utilize computer memory, in order to detect and resolve memory leaks, which can lead to performance degradation and system crashes. One of the most famous and widely used memory testing tools for software applications is Valgrind [35]. Valgrind is an open-source tool that is compatible with various programming languages and platforms. It allows developers to analyze memory usage, detect memory leaks, and examine the execution of software applications. It detects memory-related programming errors such as reading uninitialized memory and accessing out-of-bounds memory which helps to ensure the reliability and robustness of software applications.

2.3.2 Security Testing

Security testing is a fundamental aspect of the software development process and plays a crucial role in protecting sensitive data and ensuring user privacy. It focuses on evaluating

the resilience of applications and systems against potential security threats and vulnerabilities. The methodology of software security testing involves a wide range of techniques including, vulnerability assessment, security code reviews and penetration testing. One example of a software security testing tool is Burp Suite [39]. Burp Suite is a cybersecurity testing platform widely used for web application security assessments. It is capable of intercepting and manipulating HTTP/HTTPS traffic, enabling developers to discover and fix security flaws of web applications. The tool has diverse functionalities including cross-site scripting (XSS), and insecure direct object references. It also has a user-friendly interface that makes it easy to use and integrate for developers who aim to discover and address security faults, ensuring web applications are resilient against common cyber threats.

2.3.3 Network Testing

The purpose behind network testing is ensuring the robustness and reliability of computer networks. It involves a systematic evaluation of network components, protocols, and configurations to assess their performance, security, and overall functionality. Network testing plays a critical role in optimizing network performance, enhancing security measures, and ensuring seamless communication and data exchange among connected devices. A widely used network testing tool is Wireshark [46]. Wireshark is capable of capturing and analyzing network traffic in real time. It supports a wide range of protocols and provides detailed insights into the data packets traveling across a network, allowing users to examine the contents of these packets, identify potential issues, and troubleshoot network problems.

2.3.4 Fault Injection Testing

Fault injection testing is a vital methodology in software engineering aimed at assessing a system's robustness and reliability under adverse conditions. This testing technique involves introducing faults into a software application or system, simulating scenarios that could potentially compromise its functionality. By injecting these faults, developers and quality assurance teams can evaluate how the system responds to these faults, identifying vulnerabilities and weaknesses that might go unnoticed under normal operating conditions. Fault injection testing provides valuable insights into the system's error handling mechanisms, fault tolerance, and recovery procedures. It allows for a comprehensive evaluation of the software's resilience, ensuring that it can withstand unexpected failures and disruptions. Fault Injection testing can be used to test software and hardware. We mainly focus on software testing using the fault injection testing technique.

Storage Fault Injection Testing. Storage failures are complex and can go unnoticed, stored data can become inaccessible or silently corrupted [30]. With storage fault injection testing, faults are introduced into storage devices such as hard disk drivers, flash devices and network attached storage. Injected faults include disk read or write errors, file corruption, network interruptions, or storage device failures. Alagppan et al. [7] implement a fault-injection framework that injects storage faults. The framework injects two different storage faults, random corrupted blocks of data and crashed nodes at different instances in time to simulate lagging nodes while inserting values. After injecting faults, the framework initiates read requests from clients to check the system under test availability and safety. They test LogCabin and ZooKeeper. They find that for block corruptions, the original LogCabin and ZooKeeper are unsafe or unavailable for 30% of the testcases. Whereas for block error, both systems kill the infected node which causes unavailability in 50% of the cases. For lagging nodes tested scenario, in most of the test cases, the read data were either, unavailable, unsafe or incorrect.

Memory Fault Injection Testing. Memory Fault Injection testing is an approach that injects memory faults, such as data corruption, address errors, or memory leaks in a software system to test memory error handling and recovery mechanisms. Pattabiraman et al. [14] implement an error injection model that injects temporary faults in memory and registers. They replace the contents of a register or a memory location or the program counter by the symbol `err` at a certain points during execution, when the program reaches a specific location in the code, in order to detect its effect on the execution of the program. They use it to test TCAS -Traffic Collision Avoidance System- and find hardware transient errors that can lead to catastrophic consequences for the TCAS system.

Network Fault Injection Testing. Network fault injection testing is a testing technique in which network faults are intentionally introduced into the system under test connections, these faults can be network partitions, network congestion or device failure. Alquraan et al. [10] introduce NEAT framework, a testing framework with network-partitioning fault injection, it supports the three types of network partitions, complete, partial and simplex partitions. Their goal is to test if a system is resilient to network partitions. Different distributed systems were tested using NEAT, including message queuing systems, key-value store systems, scheduling systems, file systems, distributed data structures, and object storage systems. It detects catastrophic failures caused by complete and partial partitions between system components. Qunaibi et al. [41] test modern schedulers resiliency to partial network partitions and their effect on scheduling by manually injecting failures specifically partial network partitions on Apache Spark that deploys Mesos and Kubernetes. The effect of injected partitions varies depending on when it is injected, but it almost always lead to significant performance degradation or complete halting of the

application, which further proved that partial network partitions can have a significant impact on system performance.

PPATT is the first fault injection testing tool that automatically injects network partitions and verifies their effect on the system being tested. PPATT is the first tool to capture the connections established on the tested distributed system, injects partitions between captured connections and verifies the impact of the partition on the operation of the system. It is capable of testing the resiliency to network partition in modern distributed system.

Chapter 3

Design

We present PPATT, an application-agnostic tool to facilitate testing a distributed system’s resilience to network partitioning faults. The tool makes one assumption on the system under test (SUT): the SUT operation follows a deterministic protocol, i.e., it generates the same pattern of communication when repeated with the same input.

Using PPATT, developers can run a test operation (e.g., a put operation in a replicated key-value store), use PPATT to inject complete and partial network partitions before the operation runs, then verify the correctness of the system response and status.

Modern systems use tens of nodes with tens of processes. Therefore, following a brute force approach to inject network faults between all pairs of processes in the system leads to a large number of faults that need to be tested. For every operation, to test with a network partition between every pair of nodes is not tractable.

PPATT tries to limit testing only to nodes that communicate during a test. PPATT runs in two phases. First, it runs the SUT with the test operation and monitors the network to detect which nodes communicate during this operation. This phase will produce a list of node pairs that communicate during a test. In the second phase, it runs the same operation and only injects faults between nodes detected in the first phase, one pair at a time.

Figure 3.1 shows the architecture of the tool. The tool uses two types of nodes: a Coordinator node, and Daemon Processes running on every node in the cluster. The Coordinator runs the Central Coordinator processes that controls PPATT operations.

The Central Coordinator uses the System Under Test module to deploy SUT and to run an operation test. While an SUT operation is running, Central Coordinator can use Capture Connections module to run phase one of the testing processes and capture which

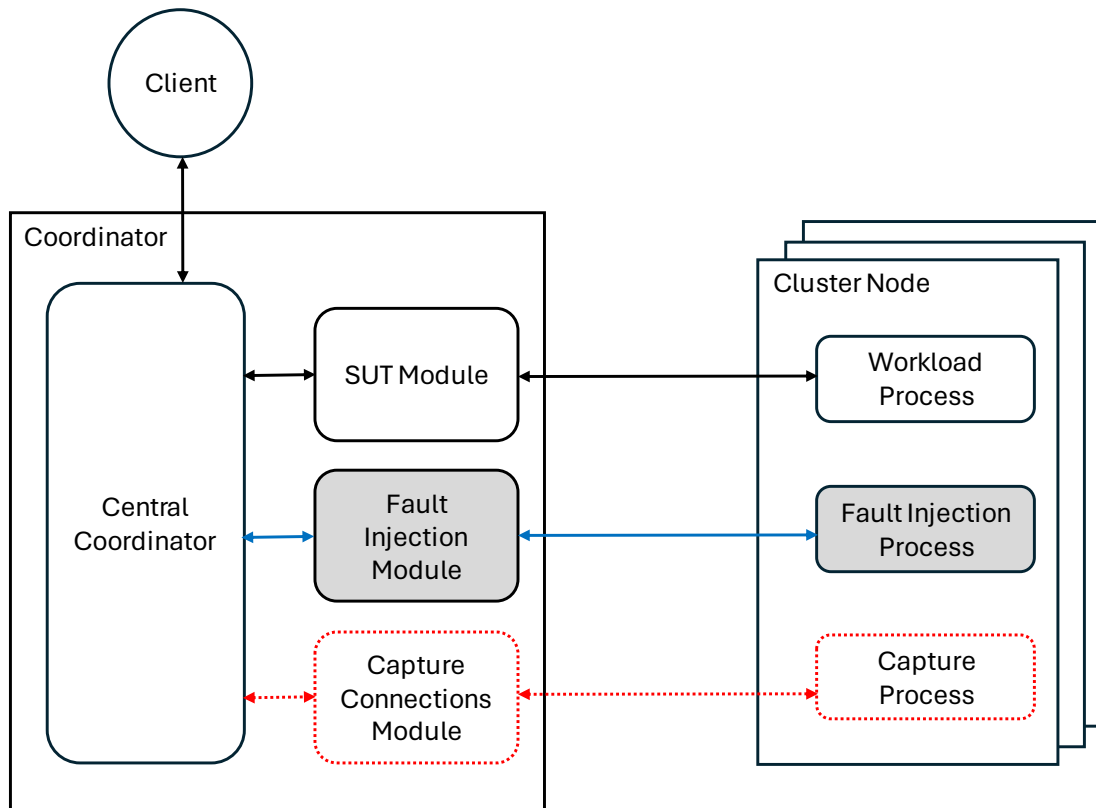


Figure 3.1: PPATT Architecture

nodes communicate during the test. Then Central Coordinator uses Fault Injection module to inject faults during an operation run. The System Under Test module, Fault Injection module, and Capture Connections module run daemon processes on every node in the cluster to, run the target system, inject failures, and capture communication, respectively. In the following subsections I detail these modules.

3.1 System Under Test module

The System Under Test module offers an abstraction to allow the Central Coordinator to control the SUT. The System Under Test module offers four main APIs: deployment,

operation test, system verification, and log collection. In the following text, we discuss the functionality of these APIs. At the end of this chapter, we discuss how a user can use PPATT to test a new system.

SUT deployment. This API is implemented by the developer and it completes two tasks: installing the SUT on the cluster nodes and deploying the SUT. The installation is typically done once. System Under Test module deploys the SUT once per operation.

Operation test. This API implements the test operation. For instance, this can be a script to put a key in a key-value store, or to submit a query to a database. There can be multiple tests in a test suit.

SUT status verification. For each operation test, this API verifies the operation output and the system status to verify the correctness of the operation. For instance, the verification script for a key-value put operation can read the put key and verify its value, examine the return code of the put operation, verify the number of replicas of the new key in the system, and verify that none of the involved nodes report an error. The verification script is application and system specific.

Log collection. This API collects SUT logs from all the nodes in a cluster and collects the `stdout` and `stderr` output from all the SUT processes. The logs are stored in a single directory at the coordinator node.

3.2 Capture Connections Module

Distributed systems typically use tens to hundreds of nodes. The goal of the Capture Connections module is to identify the nodes in a distributed system that communicate during its operation. This information helps with significantly reducing the test cases we need to consider in PPATT. In the Fault Injection module, the tool will only inject faults between nodes that communicate during a system operation.

Modern cloud platforms run complex software stack. In addition to the communication happening between the processes of the SUT, there is significant communication happening between system processes and services deployed on the same cluster, such as time protocols, accounting software, storage, and monitoring logic. If PPATT blindly captures the communication happening in a cluster during a test, it will capture a lot of communication that is not related to SUT. This complicates testing. To address this challenge, PPATT filters out any communication that is not happening between the SUT processes. In the following section, I detail how capturing communication in a cluster works, followed by how to filter out unrelated communication.

3.2.1 Capturing Communication in a Cluster

The dashed modules in Figure 3.1 show modules for the communication capture logic. To detect which nodes communicate in a cluster, we first capture any network activity, then analyze the captured log to detect which nodes communicate. The Central Coordinator controls when the Capture Connections module starts and stops the capture process.

Capturing Network Activity. Before starting the capture process, the tool synchronizes the clocks of cluster nodes using the NTP protocol. This clock synchronization makes it easier to detect connections at the end of the capture phase. To capture the network activity, the tool deploys a daemon on all cluster nodes. Each daemon runs a packet capture tool such as Wireshark [46] to capture all communication. To reduce the amount of captured packets, for TCP connections, the daemons only capture the header of the SYN and FIN packets and ignore the rest of the flow. The daemon captures all UDP packet headers. All captured headers are stored in a local log.

Communication Log Analysis. After capturing the network activity, the Capture Connections module collects captured logs from cluster nodes. The Capture Connections first analyzes each log separately to identify the following fields for each flow: protocol, source IP, source port, destination IP, destination port, start timestamp, and end timestamp. For TCP connections, the connection information and timestamps are retrieved from the SYN and FIN packets. For UDP communication, the start timestamp is the time of the first UDP packet captured between the source and destination ports, and the end timestamp is the timestamp of the last packet captured in the log. At the end of this step, Capture Connections module has a list of connections for each node.

Capture Connections module merges the list of connections for all the nodes in a cluster and eliminates duplicates. The Capture Connections module produces a list of node pairs that communicate during the capture process.

3.2.2 Capturing SUT Communication

To capture the communication of a SUT, the Capture Connections module captures background communication not related to SUT operation and captures the communication during an SUT operation run, then identifies which communication is related to the SUT operation. It follows the following steps:

1. Capture communication before any SUT test operation. Central Coordinator deploys the SUT, then deploys the Capture Connections module that uses the communication

capture and analysis mechanisms discussed above (Section 3.2.1) without running the SUT operation to be tested. The goal is to identify any communication happening in the cluster before running the SUT test operation, including any SUT services or heartbeating mechanism that are orthogonal to the test operation.

2. Capture the communication during an SUT operation test. The Central Coordinator starts the capture mechanism above, then it starts an SUT operation test. When the operation completes, the Central Coordinator stops the capture processes. Analyzing this communication identifies all communication happening in a cluster during a test operation. This includes communication related to SUT and background system communication.
3. Filter out communication not related to SUT. The Capture Connections module takes the output of step 2 above and filters out communication identified in step 1 above. This step eliminates platform related communication that is not related to the SUT.

I note some platform communication that is not related to SUT may start during the SUT operation. This may add a few test cases to test for fault injection but does not add significant overhead. If this becomes a concern, given that SUT operations are deterministic, one can run the previous three steps two times and select the output with the shorter list of connections.

The Capture Connections module produces a list of pair of nodes that communicate during a given SUT operation. This list guides fault injection during the testing phase.

3.3 Fault Injection Module

The shaded modules in Figure 3.1 show modules for the network partitioning injection logic. The process for fault injection is simple. The Central Coordinator sends a request to the Fault Injection module to break the communication between two nodes. The Fault Injection module sends the command to the fault injection daemons on the target nodes. The fault injection daemons use `iptables` commands to drop packets coming from the identified pair node. The Central Coordinator can break the communication between multiple nodes to create a complete or partial partition.

To fix a partition, the Central Coordinator sends a request to the Fault Injection module to restore the communication between two nodes. The Fault Injection module sends the

command to the fault injection daemons on the target nodes. The fault injection daemons use `iptables` commands to restore the communication between the two nodes.

3.4 Putting It All Together

To test an SUT operation (e.g., a put operation in a replicated key-value store), the Central Coordinator uses the System Under Test module and the Capture Connections module to capture the communicating processes during the tested operation. This step provides a communication list that has pairs of nodes that communicate during the tested operation. The Central Coordinator also asks the System Under Test module to run and measure the time taken to run the test without any faults. The operation time is reported to the Central Coordinator. I call this the fault-free execution time.

For each pair of nodes in the communication list, the Central Coordinator uses the Fault Injection module to inject a network partition between the two nodes, uses the System Under Test module to run the operation test, then uses the System Under Test module verification mechanism to verify the correctness of operation response and system status. If the verification fails, the Central Coordinator reports a test failure.

In some cases, the SUT operation causes the system to hang indefinitely, i.e., the operation never completes. In this case, the Central Coordinator will wait for a configurable amount of time before declaring that an application hanged. In our implementation we wait for a generous period equivalent to $5\times$ the fault-free execution time. If an operation does not complete during this time period, Central Coordinator reports a failure under this specific network partition scenario. The reason to wait for a period five times longer than the fault-free execution time is to give enough time for the SUT to handle the network failure, including identifying the problem, employing a fault tolerance technique, and completing executing the operation.

For each failed test scenario the Central Coordinator records the network partition details, and uses the System Under Test module to collect the SUT logs from all the nodes and record all `stdout` of all the SUT processes. This information is stored for further inspection.

3.5 PPATT Extensibility

PPATT is an SUT agnostic tool. PPATT design makes it easy to extend the tool to test new systems. Outside the System Under Test module, the tool is generic. For a developer

to use the tool to test a new system, they have to implement the four APIs in System Under Test module for SUT deployment, operation test, verification, and log collection. While the tool offers an interface for these four APIs, the developer has complete flexibility in implementing those scripts and customize it to the target system.

3.6 Implementation

We implement the tool in approximately 550 lines of python code. We use python 3.7. We use `chrony` to synchronize the system clocks across the nodes of the tested network. Chrony is an open-source network time synchronization software. In our implementation all nodes are synchronized with the clock of the Coordinator node as a reference.

Capturing Connections. We use Pyshark, the python wrapper of Wireshark [46], the open-source network protocol analyzer. It allows users to capture and analyze the traffic on a network in real time. With Pyshark, users can filter packets based on specific criteria for example, specific port number or network protocol. In our implementation we capture IPv4, TCP and UDP packets while excluding the TCP and UDP open ports.

Our implementation optimizes the capture approach. In our design of the communication capture mechanism, we capture the background communication, capture communication during a test, then filter out the background communication from the communication log. In our implementation, we optimize these processes. After capturing the background communication, we analyze the log to identify which IP addresses communicate and on which ports. Then we add filters to Pyshark to avoid capturing these connections. This significantly reduces the overhead of the capture mechanism and allows scaling the operation to larger systems. We also stop the capture processes once the execution of the SUT test operation is completed, as Wireshark keeps continuously capturing the flow of packets unless the user stops it or the user-defined capture duration is reached. We stop capturing using a special TCP packet sent to all cluster nodes on the completion of SUT test operation. When the capture process sees the stop packet, it terminates.

Injecting Partitions. For this phase, we use `iptables` commands to break the connection between two nodes in the network then heal it back again. It is a command-line firewall utility for Linux operating systems. To cut the communication between two nodes we use the `DROP` option and to fix it we use the `ACCEPT` option.

Chapter 4

Tool Verification

To verify the functionality of our tool, we reproduce four failures that are reported in two production systems: Spark and Kafka. We select these four failures because they manifest in a complex setup that involves multiple subsystems and have catastrophic effects. We test Spark with two schedulers Standalone and Mesos, and we test Kafka with Zookeeper. PPATT successfully detects the reported failures, and our analysis of the returned logs confirms that the reason for the failure matches the reason in the failure reports.

4.1 Failure of Spark Standalone Cluster Manager

We reproduce a failure of Spark Standalone cluster’s manager scheduling mechanism reported by Qunaibi et al. [41]. The paper reports a failure that leads to a system hanging until the network partition is fixed.

Apache Spark is a popular data analytics system. Spark system architecture consists of three main components: the application driver, cluster manager, and worker nodes. Spark comes with a basic resource manager, called Spark Standalone cluster manager. Spark supports the integration of other resource managers including Mesos and Kubernetes. When using the Spark Standalone cluster manager, the cluster manager and the application driver run on the same node.

The general workflow of a workload application is as follows. When a client submits a job to the cluster manager, the manager starts a driver program to orchestrate the job execution. The driver works with the cluster manager to allocate executors on worker

nodes. Worker nodes host the executor process which runs the application program on that node.

The driver launches the tasks at the executors. The executors run the tasks and may exchange intermediate results among themselves. The driver monitors the tasks progress, collects the final results, and returns the results to the client.

Quanibi et al. [41] report a failure when there is a partial partition between two executor nodes that need to exchange intermediate results, i.e., during the shuffle stage. If a partial partition breaks the communication between two executors, data transfer between executors during shuffle operation fails and the destination executor waits indefinitely for the intermediate results. Executors do not report this issue to the driver program nor the cluster manager. The application freezes until the partial partition is healed, because neither the driver nor the cluster manager detects the problem.

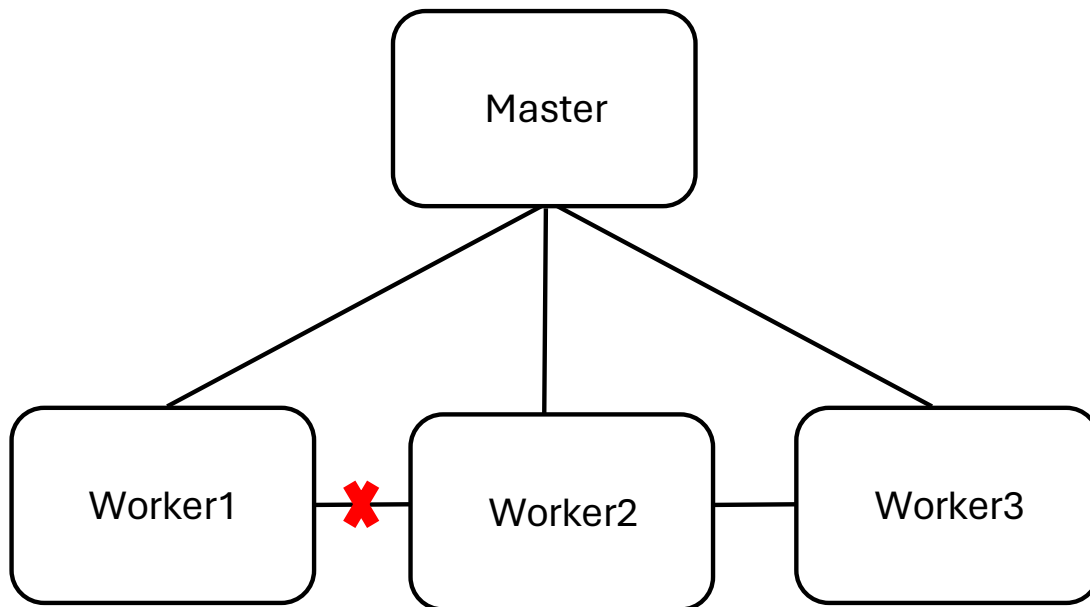


Figure 4.1: Spark Standalone Architecture

We use PPATT to test Spark with the Standalone cluster manager. Figure 4.1 shows

the components of our cluster. Master node hosts the driver program and the cluster manager, and three worker nodes host the executors instances. Lines between components in Figure 4.1 represent communication between nodes during job execution. We run the WordCount application that was used in the failure report and reproduce the reported failure.

PPATT reports a failure when there is a partition between two workers as shown in red in Figure 4.1. PPATT reports the same effect in the original report. The partition between workers prevent the exchange of their intermediate results between executors and this causes the application to pause. The investigation of the collected logs from the cluster nodes shows that the reason behind this halt is that workers keep waiting indefinitely for the intermediate results that are required for the completion of their execution from other workers.

4.2 Failure of Mesos Cluster Manager

We reproduce a failure of Mesos scheduling mechanism reported by Qunaibi et al. [41]. The reported failure leads to a failure to start a Spark application.

Apache Mesos is a cluster manager that handles resource management and allocation in large distributed environments. Spark can use Mesos as a cluster manager. Mesos runs daemons (a.k.a. Mesos agents) on cluster nodes. The agents monitor nodes' CPU and memory resources and inform the Mesos manager of available resources. The cluster manager aggregates information about available resources and offers those resources to an application driver. The driver can decline or accept all or some of the offered resources. The driver then starts executors on allocated nodes, then assigns tasks to executors.

The Mesos master continuously offers all available resources to the driver. When a client submits an application to the driver, the driver selects the nodes for the application from the list of available resources it receives from Mesos. The driver runs the executors on the allocated worker nodes, then the application is executed.

Qunabi et al. [41] report two failures. The first failure happens when there is a partial partition between two worker nodes that execute the WordCount application. The partial partition breaks the communication between the two executors and cuts the data transfer between executors during shuffle stage, which causes the destination executor to wait indefinitely for the intermediate results. Executors do not report this issue to the driver program nor the cluster manager. The application halts until the partition is healed.

The second failure happens when there is a partial partition between the driver and Mesos master before the application starts. As the driver does not receive any resource offers from the cluster manager, the application does not start.

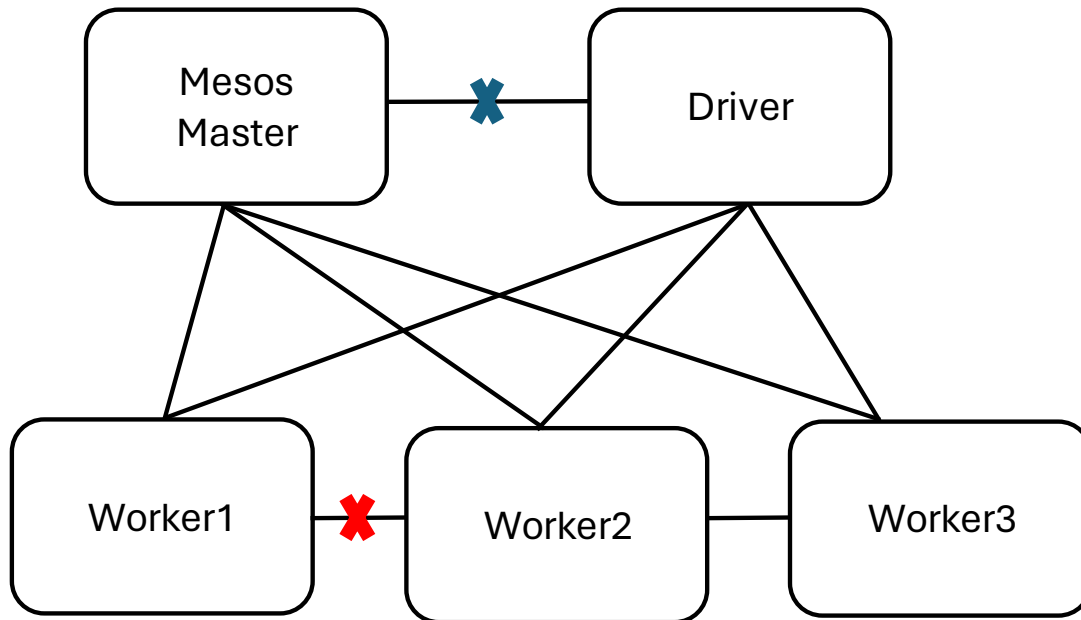


Figure 4.2: Spark with Mesos Architecture

We use PPATT to test Spark with Mesos. We deploy Spark with Mesos v(1.11.0). Figure 4.2 shows the components of our cluster; Driver node, Mesos Master node, and three worker nodes that host executors instances. The driver runs on a separate node from the Mesos cluster manager (Mesos Master). Lines between components in the Figure 4.2 represent communication between nodes during job execution. We run WordCount application as used in the original report of reported failure.

PPATT reports two failures. PPATT reports a failure during a partition between two workers and a partition between the driver and the cluster manager. For the first failure, PPATT injects a partition between two workers as shown in red in Figure 4.2. We get the same effect of such partition on the system operation, similar to the original report:

the partition between workers causes a complete halt of the system operation as workers keep waiting indefinitely for the intermediate results they need in order to complete their execution. We examined the returned system logs and verify that the application hangs because of the loss of communication between workers.

PPATT reports a second failure when it injects a partition between Mesos Master and the Driver as shown in blue in Figure 4.2. The system never starts running the application as the driver does not receive any resources from the cluster manager.

4.3 Failure of ZooKeeper Coordination Mechanism

We reproduce ticket #8702 [4], which reports a failure in ZooKeeper coordination mechanism when deployed with Kafka.

Kafka is a message queuing system. Data in Kafka is organized into topics, which act as a logical category for messages. Each topic is divided into partitions. Kafka operates in a distributed cluster that consists of brokers, producers, and consumers.

Producers publish messages to Kafka topics, they write messages to specific topics, and Kafka handles the distribution of these messages across partitions within the topic. Brokers are responsible for handling partitions and storing data. Consumers subscribe to Kafka topics to retrieve records and consume messages, they can read records from specific partitions or from multiple partitions, allowing for parallel processing.

Kafka maintains fault tolerance by replicating partitions across multiple brokers. Each partition has one leader and multiple replicas, ensuring that if a broker fails, the data remains available.

Kafka traditionally relied on Apache ZooKeeper for cluster coordination, metadata management, and leader election. ZooKeeper is a distributed coordination service, it monitors the health and availability of nodes within a distributed system and helps a cluster to select a new master if the current one fails.

The ticket #8702 [4] reports a failure within a cluster that deploys Kafka v(2.3.0) and integrates ZooKeeper for coordination purposes. They report a failure when there is a partial partition between the master broker from all other brokers. As all brokers are reachable from ZooKeeper, the nodes pause their operations because they cannot reach the master and a new master is not selected because Zookeeper can reach the old master.

We use PPATT to test Kafka on our cluster. Figure 4.3 shows our deployment. We deploy Kafka v(2.3.0) with three replicated brokers, each message is replicated on three

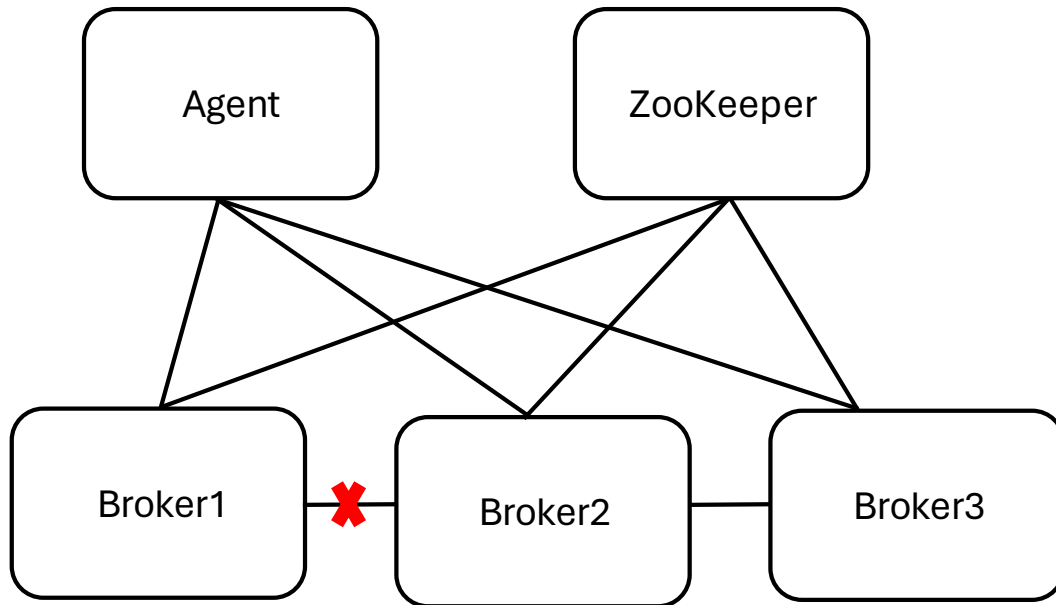


Figure 4.3: Kafka Architecture

nodes. We use Kafka’s benchmarking tool to generate load on the system. We use a set of producers and consumers. Each producer sends messages to a dedicated queue and each queue has one consumer. We deploy ZooKeeper within our cluster to monitor the cluster nodes. The lines between components in the Figure 4.3 represent communication between nodes during producing, replicating, and consuming messages.

PPATT reports a failure when it introduces a network partition between the master broker and all other brokers, as shown in red in Figure 4.3. During this partition, all test operations fail. The cluster remains unavailable until the partition heals, because ZooKeeper can reach the current master, it neither detects the problem nor helps selecting a new master, the entire cluster pauses. We examined the returned logs and verified the cause of this behavior.

Chapter 5

Demonstration of the Tool Capabilities

In order to assess the design, implementation and effectiveness of the tool, we use it to test new versions of distributed systems, to produce new failures if they exist. We start by selecting a diverse set of systems to test and end up targeting three distributed systems from different categories of large data processing systems. We decide to test a streaming system, a message queuing system and a distributed data store system. We were able to detect three catastrophic failures in these systems.

5.1 Hazelcast

Hazelcast is an in-memory distributed computation and storage system. Hazelcast system architecture consists of members and clients. A member is the computational and storage unit of the Hazelcast cluster. Clients are used to communicate with the cluster members. The first member to run is designated as the master member. Hazelcast provides a set of data structures such as map, queue, and locks. In our tests, we run experiments with map and lock operations.

Consistency Guarantees. Hazelcast offers different guarantees for different operations. Synchronization operations such as locks and count down latches are always linearizable and are implemented using a Raft [37] linearizable replication protocol. Data related operations and data structures, such as maps and queues offer a lighter weight replication

mechanism that can be configured to be strongly or eventually consistent. We experiment with these two groups of APIs and their different implementation of the replication protocol.

Partial Network Partition Tolerance. Hazelcast design includes a mechanism for tolerating partial network partitioning. To tolerate a partial partition, cluster members perform all-to-all heartbeating in which each member heartbeats all cluster members. If a member detects that another member is not reachable, it reports this connection failure to the master member. The master member collects this data, builds a graph to represent the connectivity in a cluster, then analyzes the graph to identify the largest connected subset of the cluster members (a.k.a, clique). The system divides the cluster into two sub-clusters, one including all the nodes in the clique, and one including all nodes outside the clique. This effectively turns a partial partition into a complete partition. Furthermore, Hazelcast includes a mechanism to handle complete partitions called split-brain protection technique. Hazelcast's split-brain protection feature is used when consistency is a primary concern for the user. This feature enables the user to specify the minimum cluster size required for operations to occur. If the cluster size is below this minimum value, the operations are rejected and the rejected operations return a `SplitBrainProtectionException`. The split-brain protection detects when there is a complete partition and pauses nodes in the minority partition of the network.

5.1.1 Map Read and Write Operation

Maps in Hazelcast are partitioned and replicated among members. Each partition stores a part of the map. Each partition can have multiple replicas, one acts as a primary and the others are backups. Clients send read and write requests for a particular object to the primary replica of the partition holding the object. If the primary replica fails, one of the backup replicas takes the primary role. By default, there are 271 partitions that are evenly distributed among members. Each may be a primary for some partitions and a backup for others.

Hazelcast has two types of backup replicas: synchronous and asynchronous backups. In synchronous backups, each write operation is synchronously replicated to backup nodes before confirming the write operation as successful. Asynchronous backups involve a delay between the write operation and the replication of data to backup nodes. A write operation can be confirmed to the client before it is replicated to backups.

Hazelcast uses partition tables within each cluster member in order for all members in the cluster to keep track of where the data is. The Partition table stores the partition

IDs and the addresses of cluster members that are the primary for that partition. The partition table is created by the master member. The master member periodically, every 15 seconds by default, sends the partition table to all members.

Hazelcast uses keys to distribute map entries across partitions. When an entry is added to a map, Hazelcast assigns that entry to a specific partition based on a hash of the entry key. Hazelcast hashes the key using a hashing algorithm, then calculates the $\text{mod}(\text{hash result}, \text{partition count}(271))$, the result is the partition ID in which the key value is stored. For all members in the cluster, the partition ID for a given key is always the same.

If a client wants to write a value to a specific key, Hazelcast hashes the key and calculates the partition ID in which the data will be stored. The client randomly connects to one of the members listed in its configurations. The client gets the partition table of the member and finds the member that is hosting the specific partition for the new key. If a synchronous replication is used, a client blocks until backups receive the new write operation before an acknowledgment is sent back. If the client wants to read a value of a specific key, it follows the same steps and reads the value of the key from the primary replica.

Test Setup

We use PPATT to test map read and write operations. We deploy Hazelcast v(5.3.1) on a four nodes cluster, three nodes are members and one node is a client. We configure the cluster with synchronous replication and with a replication level of 3. The map is configured with a size of 999. The client writes 999 objects with each object having the size of 1 KB. This load uses the three members in our cluster as this load touches most of the partitions. We enable the partial partition fault tolerance mechanism, and the split-brain protection mechanism with `minimum-cluster-size` of two.

Figure 5.1 shows the cluster architecture, lines between components in the figure show the captured communications happening during the execution of map read and write operations.

Failures Discussion

Our test discovers a failure in the client access protocol in which a client is not able to access the system despite the availability of members to serve its request.

The discovered failure happens when a partial partition impacts the communication between two members of the cluster. Figure 5.1 shows such a partial partition in red. In

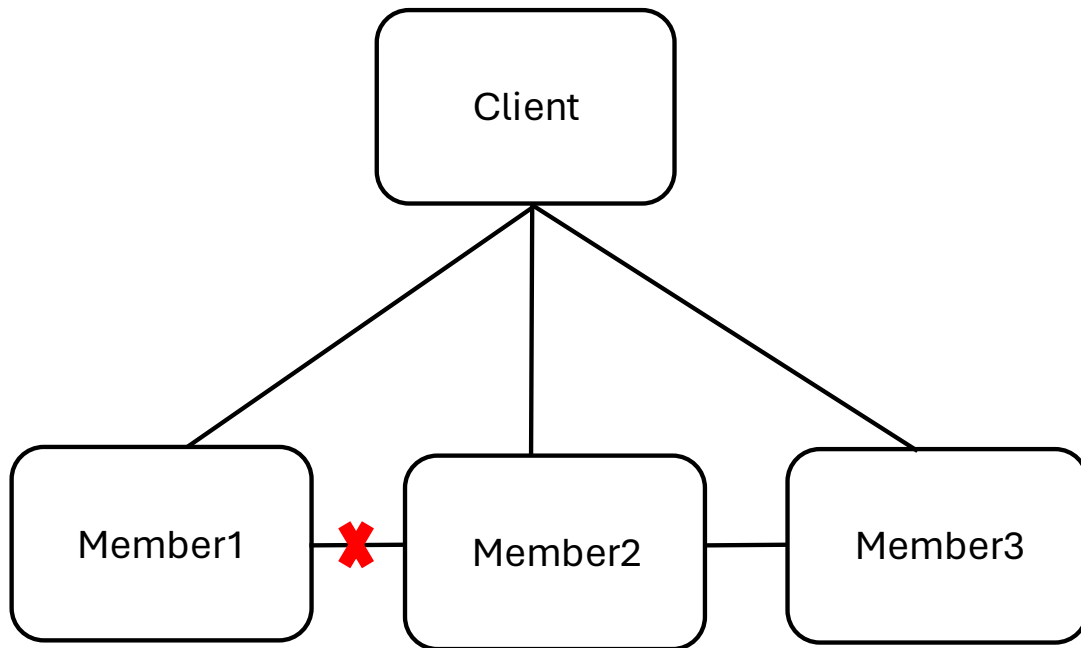


Figure 5.1: Hazelcast Architecture

this scenario, the master member detects this partition and follows the partial partition tolerance mechanism. The mechanism splits the members into two sub clusters, with one side containing the majority of nodes that are connected. As we use the split-brain protection mechanism, nodes on the minority side pause their operation and do not process client requests. Clients select which member to send their requests to in a random fashion. If a client is not impacted by the partial partition and can reach all members and it sends its request to a member on the minority side, the request will fail. The client gets the `SplitBrainProtectionException` exception, which indicates that the operation failed because it is sent to the minority side. Hazelcast fails these operations unnecessarily as the operation fails although the client is connected to the majority side and can send its request to the active side.

5.1.2 Count down latch and Fenced locks Operation

Count down latch and Fenced locks uses Raft algorithm [37] for its replication. In a normal operation, members are divided into a leader and followers. Cluster members select their own Raft leader, which runs the Raft consensus algorithm. All other members become followers. The leader is responsible for handling incoming requests from clients and replicating those requests to follower members. Internal heartbeats between members maintain the authority of the Raft leader.

If a client requests to lock or unlock a lock or initialize a count down latch, it randomly connects to one of the members listed in its configurations. This member leads the client to the leader that processes the client request. Operations are committed and executed only after they are successfully replicated to a majority of members.

Test Setup

We use PPATT to test lock, unlock, and count down operations. We run two separate tests for Fenced locks and the Count Down Latch. In both tests we deploy Hazelcast v(5.3.1) on a four nodes cluster, three nodes configured as members and one node is the client. Figure 5.1 shows the cluster architecture. Lines between components in the figure show the captured communications happening during the execution of client requests.

Failures Discussion

Our test discovers a failure in the client access protocol. This failure is similar to the failure we reported with map operations. A client is not able to access the system despite the fact that it can reach the current cluster leader.

If a complete partition isolates the leader from the cluster. The old leader pauses its operation, and the nodes on the majority side elect a new leader. A client randomly selects a member to send its requests to. If it connects to the old leader, the client operation will fail and return an exception indicating that this member has left the cluster. Hazelcast fails these operations unnecessarily. If a client connects to one of the members on the majority side that has the new leader, the client operations succeed.

5.2 Apache Flink

Flink is a data streaming processing system. Figure 5.2 shows the Flink system architecture. The architecture includes a JobManager and TaskManagers. The JobManager receives jobs from clients, tracks job progress, and manages resources. A TaskManager runs on every worker node in the cluster. A TaskManager executes tasks.

Flink quantizes resources as slots. A slot is the basic unit of resource scheduling and expressing resource requirement for a task. TaskManagers resources are defined by configuration and can vary between different TaskManagers within the cluster. To allocate resources, a default strategy is a greedy strategy in which the JobManager traverses the TaskManagers slots and selects the first one that has enough resources to fulfil the request. The list of TaskManagers is provided by the admin and is always traversed in order. TaskManagers offer their free slots to the JobManager, the JobManager stores these slots in a slot pool. When a job is submitted to the JobManager e.g. WordCount application, the Scheduler within the JobManager queries the slot pool. If the slot resources are sufficient, resources are assigned to the job. The JobManager uses the assigned resources to schedule tasks. Once the task execution is finished, the slots are freed and their resources are returned to the available resources of the TaskManager. TaskManagers may communicate during the execution of an application to exchange intermediate results during the shuffle stage.

Flink has a built-in fault tolerance mechanism based on checkpointing. With checkpointing, Flink periodically takes snapshots of the state of every operator in the cluster. In the event of a failure, Flink restores a complete state of the cluster and resumes processing.

Test Setup

We use PPATT to test the execution of a WordCount application on Flink. We deploy Flink v(1.17.1) on a cluster of six nodes, the cluster consists of a JobManager and five TaskManagers. Figure 5.2 shows the cluster architecture, lines between components in the figure show the captured communications happening during the execution of the WordCount application. We set the number of slots per TaskManager to one, and submit the WordCount application with a level of parallelism equal to three. This configuration forces the JobManager to run the tasks on three different TaskManagers. Shaded TaskManagers in Figure 5.2 represent the assigned TaskManagers to execute the submitted job. We enable Flink checkpointing. We configure the checkpointing mechanism to attempt a restart three times, and the time between attempts to 10 seconds.

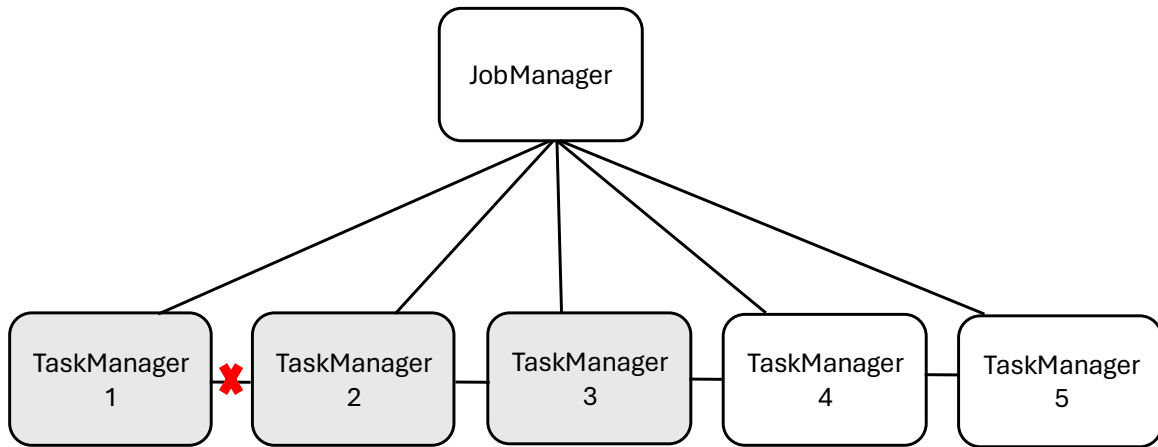


Figure 5.2: Flink Architecture

Failure Discussion

Our test discovers a failure in Flink’s resource manager under a partial network partition. The failure leads to a termination of an application despite the availability of resources to execute the application.

In the WordCount application, TaskManagers exchange data during the shuffle stage. If a partial partition between TaskManagers prevents the data exchange, it causes the job execution to fail and the allocated slots to be released. As checkpointing is enabled, the JobManager restarts the execution of the job. The JobManager checks the slot pool to find resources for the rerun of a job. The greedy scheduling strategy selects the first slots available for a task. Given that the JobManager is not aware of the partial partition between the TaskManagers, the resources at the nodes impacted by the partial partition are at the top of the list of available slots. Consequently, the JobManager selects the same TaskManagers chosen earlier as it always selects the free slots of the first TaskManagers in its static list. Because of the partial network partition between the TaskManager, the rerun of the job fails. Flink tries to run the job 3 times before giving up and raising an exception. Interestingly, the raised exception is the `PartitionConnectionException` failed partition requests due to connection failure with unreachable producer. Although Flink terminated the job due to a communication problem, it did not realize that the

communication problem only impacts two nodes and that it could avoid the problem by scheduling the task on nodes that are not impacted by the partial partition. We reported this failure to the Flink issue tracking system.

5.3 ActiveMQ Artemis

ActiveMQ Artemis is a message queuing system. Figure 5.3 shows the system architecture. ActiveMQ Artemis consists of producers, brokers, and consumers. Producers produce messages, brokers are the messaging middleware, and consumer consumes messages. ActiveMQ Artemis supports replication for higher availability. In the replicated configuration, Artemis uses one live broker to serve all producers and consumers requests. The system can have one or more backup brokers. Backup brokers are not active. One of the backup replicas is randomly selected to be in a passive mode. The passive backup connects to the majority of backups in the cluster, if not, the backup waits and tries reconnecting to the live broker. The passive broker announces its status as a passive broker and starts monitoring the live broker heartbeats. Synchronization occurs in parallel with current network traffic and does not cause any blocking on current client requests. All messages are replicated to this passive backup. This replication is asynchronous and does not cause any blocking on client requests.

The live broker keeps heartbeating all nodes in a cluster. If the passive broker misses the heartbeats from the live broker, it assumes the role of the live broker. The new live broker starts to heartbeat all backup brokers and serve producers and consumers. Producers produce messages to specific names addresses (e.g., "news" and "stock changes"). Messages are stored in a queue before being served to consumers.

Test Setup

We use PPATT to test the produce and consume messages operations on ActiveMQ Artemis. We deploy ActiveMQ Artemis v(2.30.0) on a five-node cluster. Figure 5.3 shows the cluster architecture. Lines between components in the figure show the captured communications happening during the execution of produce and consume messages requests.

Failure Discussion

Our test discovers a failure in the replication protocol of ActiveMQ Artemis. The failure leads to having two live brokers in a cluster, leading to data inconsistency.

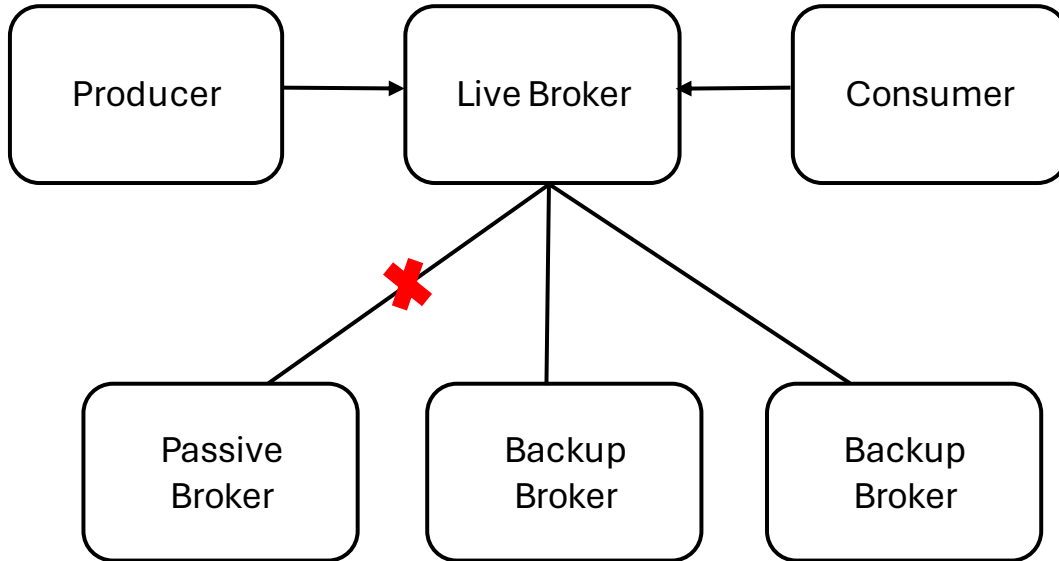


Figure 5.3: ActiveMQ Artemis Architecture

If a partial partition cuts the communication between the live and passive broker as shown in red in Figure 5.3, the passive broker will miss the live broker heartbeats and will assume that the live broker crashed. The passive broker will start serving as a live broker, serve producer and consumer requests, and heartbeating backups. All the messages are replicated to the announced backup. With this failure, the system has two live brokers serving the same address. The two live brokers do not synchronize their data. Consequently, consumers do not get all messages sent to an address. Furthermore, consumers subscribed to the same address may get different sets of messages. These scenarios violate the system guarantees. While there are two live brokers, no passive broker is elected and data replication is disabled. This leads to lower system reliability. Interestingly, the problem persists even after the network partition is fixed, the system continues to have two live brokers. We reported this failure to the ActiveMQ issue tracking system.

Chapter 6

Conclusion and Future Work

We built PPATT, an automatic fault injection testing tool, that tests software's resiliency to partial network partitions by injecting partitions between connections captured by the tool. The tool is capable of reproducing failures and detecting new failures in a wide range of distributed systems.

The current design of the PPATT inserts partitions before launching the application on the system under test. One future plan is injecting partitions at certain points of time while running the application.

Also, the current design distinguishes between connections depending on their source and destination IP. A future work is taking port numbers into consideration, i.e., considering duplicated connections as connections that have the same source and destination IP addresses and port numbers.

Overall, our tool automates and eases the testing process for distributed systems' resiliency and fault tolerance techniques for partial network partitions. It is easy to use and integrates with a wide range of applications, which makes it the right choice for developers and quality insurance specialists to adopt for testing their software systems and developing their designs and techniques to avoid catastrophic failures that can happen due to partial network partitions.

References

- [1] Artemis-4555: Activemq artemis can have two live brokers in one cluster at one time. <https://issues.apache.org/jira/projects/ARTEMIS/issues/ARTEMIS-4555?filter=allissues>. Accessed: 5-Jan-2024.
- [2] Flink-34006: Flink terminates the execution of an application when there is a network problem between taskmanagers. <https://issues.apache.org/jira/browse/FLINK-34006>. Accessed: 5-Jan-2024.
- [3] Hazelcast-26208: Hazelcast fails some client requests when there is a split in the cluster. <https://github.com/hazelcast/hazelcast/issues/26208>. Accessed: 5-Jan-2024.
- [4] Kafka-8702: Kafka leader election doesn't happen when leader broker port is partitioned off the network. <https://issues.apache.org/jira/browse/KAFKA-8702>. Accessed: April 2023.
- [5] Mapreduce ticket 4832. <https://issues.apache.org/jira/browse/MAPREDUCE-4832>. Accessed: April 2023.
- [6] Mesos-1529: Handle a network partition between master and slave. <https://issues.apache.org/jira/browse/MESOS-1529>. Accessed: April 2023.
- [7] Ramnatthan Alagappan, Aishwarya Ganesan, Eric Lee, Aws Albarghouthi, Vijay Chidambaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Protocol-aware recovery for consensus-based distributed storage. *ACM Trans. Storage*, 14(3), oct 2018.
- [8] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a generic fault tolerance technique for partial network partitioning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 351–368. USENIX Association, November 2020.

- [9] Basil Alkhatib, Sreeharsha Udayashankar, Sara Qunaibi, Ahmed Alquraan, Mohammed Alfatafta, Wael Al-Manasrah, Alex Depoutovitch, and Samer Al-Kiswany. Partial network partitioning. *ACM Trans. Comput. Syst.*, dec 2022. Just Accepted.
- [10] Ahmed Alquraan, Hatem Takruri, Mohammed Alfatafta, and Samer Al-Kiswany. An analysis of network-partitioning failures in cloud systems. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 51–68, USA, 2018. USENIX Association.
- [11] Andrey Falko. 2019. 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>, Sep 2019. [Online; accessed 19-Dec-2022].
- [12] ActiveMQ Artemis Documentation. <https://activemq.apache.org/components/artemis/documentation/>, 2023. [Online; accessed 1-Sep-2023].
- [13] Apache ZooKeeper. <https://zookeeper.apache.org/>. [Online; accessed 20-May-2023].
- [14] IEEE Staff Corporate Author. Symplified: Symbolic program-level fault injection and error detection framework. In *2008 IEEE International Conference on Dependable Systems and Networks*, 2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN), [Place of publication not identified], 2008-06. I E E E.
- [15] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12(7):20–32, jul 2014.
- [16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. Tao: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, page 49–60, USA, 2013. USENIX Association.
- [17] CloudFlare Blog. 2020. A Byzantine failure in the real world. <https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/>, Nov 2020. [Online; accessed 19-Dec-2022].

- [18] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s Globally-Distributed database. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 261–264, Hollywood, CA, October 2012. USENIX Association.
- [19] Brendan Cully, Geoffrey Lefebvre, Dutch Meyer, Mike Feeley, Norm Hutchinson, and Andrew Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI’08, page 161–174, USA, 2008. USENIX Association.
- [20] Datadog. 2013. Learning from AWS Failure. <https://www.datadoghq.com/blog/gray-aws-failures/>, Oct 2013. [Online; accessed 19-Dec-2022].
- [21] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.
- [22] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. *SIGCOMM Comput. Commun. Rev.*, 41(4):350–361, aug 2011.
- [23] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM ’11, page 350–361, New York, NY, USA, 2011. Association for Computing Machinery.
- [24] Google Cloud. 2019. Google Cloud Networking Incident #18003. <https://status.cloud.google.com/incident/cloud-networking/18003>, Feb 2019. [Online; accessed 19-Dec-2022].
- [25] Ramesh Govindan, Ina Minei, Mahesh Kallahalla, Bikash Koley, and Amin Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM ’16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.
- [26] Hazelcast Documentation. <https://docs.hazelcast.com/hazelcast/5.3/>, 2023. [Online; accessed 15-July-2023].

- [27] HDFS High Availability. <https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html>, 2023. [Online; accessed 1-Sep-2023].
- [28] Roger Ignazio. *Mesos fundamentals*, page 58–62. Manning, 2018.
- [29] iptables(8) - Linux man page. <https://linux.die.net/man/8/iptables>. [Online; accessed 1-Feb-2023].
- [30] Weihang Jiang, Chongfeng Hu, Yuanyuan Zhou, and Arkady Kanevsky. Are disks the dominant contributor for storage failures? a comprehensive study of storage subsystem failure characteristics. *ACM Trans. Storage*, 4(3), nov 2008.
- [31] Kubernetes Documentation. <https://kubernetes.io/docs/home/>, 2023. [Online; accessed 24-Mar-2023].
- [32] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proc. ACM Program. Lang.*, 2(POPL), dec 2017.
- [33] Tony Mills. Bnx2 cards intermittantly going offline.
- [34] Andrew Montalenti. Kafka apocalypse: A postmortem on our service outage, Mar 2015.
- [35] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '07*, page 89–100, New York, NY, USA, 2007. Association for Computing Machinery.
- [36] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.
- [37] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference, USENIX ATC'14*, page 305–320, USA, 2014. USENIX Association.
- [38] Partial network partitions and obstacles to innovation. <https://rachelbythebay.com/w/2012/02/16/partition/>, Feb 2012. [Online; accessed 14-Dec-2023].
- [39] PortSwigger. Burp suite - application security testing software.
- [40] Pyshark Documentation. <https://github.com/KimiNewt/pyshark/>, 2023. [Online; accessed 15-Dec-2022].

- [41] Sara Qunaibi, Sreeharsha Udayashankar, and Samer Al-Kiswany. Caspr: Connectivity-aware scheduling for partition resilience. *SRDS 2023*, Sep 2023.
- [42] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.
- [43] D. Turner, K. Levchenko, J.C. Mogul, S. Savage, and A.C. Snoeren. On failure in managed enterprise networks. *HP Laboratories Technical Report*, 01 2012.
- [44] Daniel Turner, Kirill Levchenko, Alex C. Snoeren, and Stefan Savage. California fault lines: Understanding the causes and impact of network failures. *SIGCOMM Comput. Commun. Rev.*, 40(4):315–326, aug 2010.
- [45] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI '06*, page 307–320, USA, 2006. USENIX Association.
- [46] Wireshark User’s Guide. https://www.wireshark.org/docs/wsug_html_chunked/. [Online; accessed 15-Dec-2022].
- [47] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13*, page 292–308, New York, NY, USA, 2013. Association for Computing Machinery.
- [48] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. Simple testing can prevent most critical failures: An analysis of production failures in distributed Data-Intensive systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 249–265, Broomfield, CO, October 2014. USENIX Association.
- [49] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 15–28, 2012.