Slicify: Fault Injection Testing for Network Partitions

Seba Khaleel*
University of Waterloo
Waterloo, Canada
stayser@uwaterloo.ca

Sreeharsha Udayashankar*
University of Waterloo
Waterloo, Canada
s2udayas@uwaterloo.ca

Samer Al-Kiswany
University of Waterloo
Waterloo, Canada
alkiswany@uwaterloo.ca

***Abstract*—Modern distributed systems are complex. They include hundreds of components that implement complex protocols such as scheduling, replication, and access control. These systems are expected to offer high availability and preserve their data even in the face of external environmental faults. Testing is the primary approach for improving system reliability. Testing against environmental faults such as hardware failures, memory corruption, and network problems is complicated since they can happen at any step in the protocol and affect any component.**

We present Slicify, a generic framework to test the network partition resilience of distributed systems. Slicify injects network partitions during unit tests to analyze system behavior in their presence. Slicify reduces the test space in an application-agnostic fashion with its novel connection tracking mechanism. We verify Slicify's capabilities by reproducing previously documented failures in two production systems. In addition, we demonstrate its effectiveness by uncovering new failures in three popular distributed systems.

***Index Terms*—distributed computing, fault tolerance**

I. Introduction

Modern distributed systems should be reliable, highly available, and durable. Building a reliable distributed system is challenging due to failures that impact the system hardware or software [1]. Among the most complex failures to tolerate are those caused by environmental faults external to the distributed system's code. Examples of such faults include memory corruption, disk corruption, and network failures.

Testing and debugging the system's tolerance to external faults using standard testing frameworks is complicated as these faults can occur at any point during system operation. Furthermore, these faults can impact any component or stored data. For instance, network faults can impact communication between two subgroups of cluster nodes, and disk corruption can impact data or metadata objects. The impact of a fault depends on which components and data experience the fault as well as when it occurs.

To improve the resilience of systems to external faults, developers resort to testing them with fault injection i.e. the testing framework includes a mechanism to mimic external failures by injecting them during system tests. One example is memory fault injection [2], in which corrupted data is injected into system memory to analyze its ability to detect and handle the fault. Despite significant attention being devoted to testing via fault injection [2], [3], it has remained largely unused when testing for network failures such as network partitions.

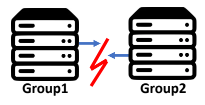
Network partitions [4] have detrimental consequences and can lead to catastrophic system failures. For instance, network partitions led to service outages at Cloudflare [5], Google [6], Lyft [7], and Amazon AWS [8]. NEAT [9] studies network partitioning failures from 25 diverse production systems and report that they lead to data loss, data corruption, stale reads, double locking, and system crashes. In addition, NEAT identifies a specific kind of network partition: partial partitions.

Partial partitions disrupt the communication between a subset of nodes in the cluster while other nodes remain unaffected. Figure 1c shows a cluster that is impacted by a partial partition, leading to the division of nodes into three distinct groups. Nodes in Group 1 and Group 3 are disconnected while Group 2 nodes can communicate with those in both other groups. Such partitions cause the system to enter a state of confusion. Some nodes (Groups 1 and 3) run fault tolerance mechanisms while others (Group 2) do not, as they see that all cluster nodes are alive and healthy. This state of confusion is poorly understood and tested. Partial partitions can have a significant impact. NIFTY [10], [11] studied partial partitions and reported that they lead to catastrophic failures such as data loss, data corruption, and crashes in production systems. Furthermore, the study shows that these failures are easy to manifest and are primarily caused by system design flaws.

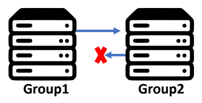
NEAT [9] and NIFTY [10] both discuss ways to improve the resilience of systems to such faults and highlight that comprehensive partition-based testing is the key to improving system reliability. Unfortunately, no existing testing tool can readily test against this type of network failure. Designing a testing framework for network partitions is challenging because of the sheer number of possible partitions and the points of time during which they can occur. This significantly increases the number of test cases, making it impossible to test every possible scenario.

In this paper, we present Slicify, a fault injection framework to test system behavior under network partitions. Slicify facilitates the testing of distributed systems for partition resilience while itself remaining application agnostic. Slicify limits the test space using a novel connection tracking mechanism to make testing feasible. It uses a connection tracking module to identify the system components communicating during fault-

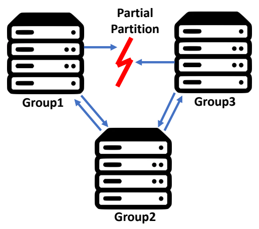
* Denotes equal contribution



(a) Complete Partition



(b) Simplex Partition



(c) Partial Partition

Fig. 1: Types of Network Partitions

free scenarios and uses this knowledge during partition testing.

We implement Slicify and verify its capabilities by using it to reproduce previously documented failures in Apache Spark, Mesos [12] and Kafka [13] (§V). In addition, we demonstrate the effectiveness of Slicify by using it to test three popular systems (§VI - §VIII) against network partitions: Hazelcast [14], Apache Flink [15], and ActiveMQ Artemis [16]. Slicify finds four previously undocumented bugs with catastrophic impact across these systems, including data inconsistency and system unavailability. We have reported these bugs via failure reports [17]–[19] to the respective system development communities. All the failures arise from system design flaws and were discovered by Slicify without knowledge of the system design. We have made our code publicly available [20].

II. Background and Motivation

A. Network Partitions

Network partitions are failures that disrupt the communication between nodes in a system. Turner et al. [4] report that network partitions happen approximately once every four days within the California-wide CENIC network, Google reported 40 network partitions in two years [21] and according to Microsoft's findings, network partitions account for 70% of their reported downtime [22]. Network partitions occur due to numerous underlying factors such as hardware failures [22], software issues [23], network congestion, or temporary disruptions [4]. Each scenario presents unique challenges for system reliability, data consistency, and fault tolerance.

Network partitions can be categorized based on their topological impact into complete, partial, and simplex partitions [9]. Figure 1 shows examples of each of these. A complete partition (Figure 1a) divides a cluster into two completely disconnected groups. In a simplex partition (Figure 1b), traffic flows only in one direction. Finally, a partial partition (Figure 1c) affects some but not all nodes in the system.

B. Impact of Network Partitions

NEAT [9] and NIFTY [10], [11] analyze the impact of network partitions on distributed systems. They report that network partitions cause catastrophic failures such as data loss, corruption, and unavailability. The majority of failures that happen due to network partitions are silent, causing permanent damage which persists in the system even when the partition is healed. The failures are also easy to manifest; they need only a few operations to occur. One of their key insights is that the majority of these failures can be avoided with better testing. They find that *these failures can be reproduced using a small cluster of three nodes* by using a testing framework capable of injecting network partitions.

C. Testing System Behavior under Network Partitions

Network partition testing today is cumbersome, requires detailed knowledge of system architecture and fails to cover all possible test cases. Network partitions come in three flavors (§II-A), can occur at any time before or during test operations [9] and can affect any combination of system components, resulting in a huge number of possible test scenarios. It is not feasible to run every unit test under all of these scenarios due to the computational cost involved.

Hence, the majority of partition-related testing today is done with human intervention. For instance, NIFTY [10] and CASPR [12] both analyze the impact of network partitions on production systems. To make testing feasible, they analyze each system's design in detail before manually deciding partition placement and timing. Similarly, NEAT [9] offers a framework for network partition injection but leaves it up to the developer to decide when, how, and between which system components to place these partitions. Due to the sheer size of the test space and the increasing complexity of modern applications, such manual testing often results in poor coverage and untested behavior. Slicify instead provides an automated and application-agnostic way for developers to test their systems, reducing the effort required for partition resilience testing while simultaneously improving coverage.

III. Design

Slicify is an application-agnostic tool to facilitate testing distributed systems for network partition resilience. Figure 2 shows the design of Slicify, consisting of a command node and cluster nodes. The command node houses the `Coordinator`, which is responsible for coordinating tests and inserting network partitions. The distributed system to be tested is deployed on the cluster nodes.

The coordinator contains various modules to support its operations. To allow for testing distributed systems in an application-agnostic fashion, it uses a `SUT Control` module. This module exposes abstract APIs relevant to system management, such as deployment and test execution, to the developers. To limit the test space and identify intra-system communication, it uses a `Connection Tracker` module. Finally, network partitions are inserted using the `Partition Injection` module. As shown in Figure 2, Slicify also runs a lightweight `Daemon` process on each cluster node to communicate with the coordinator and execute local commands. By default, Slicify uses NEAT's [9] and NIFTY's [10] insights

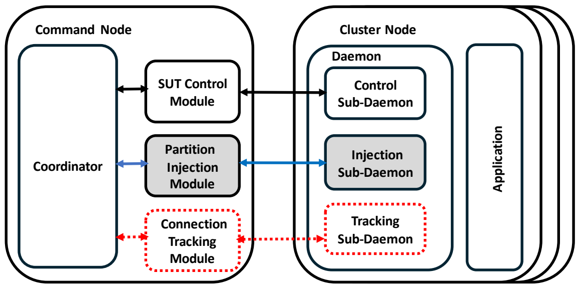


Fig. 2: Slicify's Design

to test for only partitions involving up to 3 nodes, as the majority of failures can be reproduced with such partitions. However, depending on the available computational resources, this can be extended to cover all possible partitions.

A. SUT Control Module

The `SUT Control Module` allows the coordinator to control the System Under Test (SUT) i.e. the distributed system to be tested. The module exposes the following APIs to be implemented by the system developers:

- *Installation API:* This API is used to install the SUT and its required dependencies onto all cluster nodes.
- *Deployment API:* This API is used to launch or terminate SUT instances on cluster nodes.
- *Test Execution API:* This API allows unit tests to be run by the coordinator. Unit tests consists of multiple simple operations such as database queries [24] or PUT operations within a key-value store [25].
- *Result Verification API:* For each unit test, this API is used to analyze the logs to verify execution correctness. For instance, the verification for a PUT operation can be to GET back the key, verify its value and ensure that none of the involved nodes report an error.

B. Connection Tracking Module

Distributed systems typically use tens to hundreds of nodes. As discussed in §II-C, running each unit test with every possible network partition is not feasible. The goal of the connection tracking module is to identify the nodes hosting system components involved in network communication during each unit test. This information helps Slicify significantly reduce the number of test cases. Disrupting communication between non-communicating nodes would not affect the unit test and thus, such nodes are not considered during partition placement.

Filtering out noise. Modern cloud platforms run complex software stacks. In addition to the communication happening between SUT components, there is significant background traffic between cluster nodes. This may involve communication between system processes or other platform-centric applications such as monitoring tools and health checkers. To eliminate this noise, Slicify filters out communication external to the SUT by obtaining a list of previously open ports on each cluster node before SUT deployment and ignoring their traffic.

Clock Synchronization. Before starting the capture process, the system clocks of all cluster nodes are synchronized (using a protocol such as NTP [26]) to the command node's clock. Clock synchronization makes it easier to detect the direction of traffic flow during the capture phase.

Distributed Packet Capture. Slicify captures network activity in a distributed fashion using all its daemons and then processes the packet logs on the command node. Each daemon runs a packet capture utility such as Wireshark [27] to capture all packets sent and received by its node during the test.

However, this can still result in a large number of packets captured per node. To reduce this, the daemons only capture the headers of SYN and FIN packets for TCP connections and ignore the rest of the flow. For UDP connections, they capture all packet headers. All captured data is stored locally.

Communication Log Analysis. After the unit test is complete, the captured packet logs from all cluster nodes are sent to the command node. The connection tracker first analyzes each log separately to identify the following fields for each flow: Protocol, Source IP, Source Port, Destination IP, Destination Port and Start and End Timestamps. For TCP connections, the connection information and timestamps are retrieved from the SYN and FIN packets. For UDP communication, the start timestamp is retrieved from the first UDP packet captured between the source and destination ports while the end timestamp is retrieved from the last captured packet. The connection tracker module then merges the lists of connections from all cluster nodes and eliminates duplicates. This allows it to produce a list of all node pairs that communicate during the capture process.

We note that some background communication unrelated to the SUT may start after noise filtering or during the unit test. This may add a few additional test cases for fault injection but does not incur a significant overhead. If this becomes a concern, Slicify can be configured to run the previous steps multiple times and select the output with the shortest list of connections.

C. Partition Injection Module

Figure 2 shows the partition injection module. The `Coordinator` uses the partition injection module to disrupt all communication between the target nodes. When a partition is desired between two nodes, the partition injection module notifies the corresponding sub-daemons on the target nodes. The sub-daemons configure the operating system to drop packets from the other node, simulating a network partition. Healing a partition involves reversing these changes.

D. Putting It Together - Slicify's Operational Procedure

The `Coordinator` first installs the SUT and required dependencies onto cluster nodes using the `SUT Control Module`'s installation API. Unit tests are run via the Test Execution API from the `SUT Control Module` while the `Connection Tracking` module obtains the list of node pairs communicating during the test. The `Coordinator` also records the time taken to run the test without any partitions, called the fault-free execution time.

Following this, for each node pair in the list, the `Coordinator` uses the `Partition Injection` module to inject a network partition between the nodes. It then uses the `SUT Control Module` to rerun the unit test and verify its output. Slicify also repeats this procedure for all triplets of communicating nodes i.e. if Nodes A and B communicate with Node C, a partition is inserted such that A and B are both isolated from C. Finally, Slicify isolates one node at a time from the rest of the cluster. These cover the majority of cases because, as shown by NEAT [9] and NIFTY [10], almost all issues can be reproduced by 3 nodes or less experiencing partitions. However, if computational resources are available, Slicify can be configured to test all combinations of partitions involving more nodes. If the verification fails, the `Coordinator` reports a test failure. Note that while Slicify supports inserting partitions at any time during unit tests, our evaluation only checks for the impact of *pre-existing partitions* i.e. partitions that exist when a unit test is begun.

In some cases, the partition causes the system to hang indefinitely, i.e., the unit test never completes. In this case, the `Coordinator` will wait for 5$\times$ the fault-free execution time before reporting a failure. This is to allow sufficient time for SUT fault tolerance techniques to detect and handle the problem.

For each failed test scenario the `Coordinator` records the network partition details and collects SUT logs from all cluster nodes. This information is stored for further inspection by system developers.

E. Slicify's Extensibility

Slicify is an application-agnostic tool. Slicify's design makes it easy to extend the tool to test new systems. Outside of the `SUT Control Module`, the tool is generic and follows the same operational procedure for all distributed applications. For a developer to use the tool to test a new system, they have to implement the `SUT Control Module` APIs for their system. While the tool exposes an interface for these APIs, developers have complete flexibility in tailoring the implementation to their respective applications.

IV. Implementation

We implement Slicify in $\sim$ 750 lines of Python code. We use Python 3.7 for our implementation. We have made our code publicly available [20].

SUT Control Module. To estimate the amount of effort needed to develop the SUT control module, we have released a Slicify-compatible client-server application with our code [20]. We implemented the SUT control module for this application using $\sim$ 50 lines of Python and Linux shell code.

Connection Tracking Module. We use `chrony` [28], an open-source time synchronization tool, to synchronize the system clocks across cluster nodes before capturing network communication. In our implementation, all nodes are synchronized using the command node's clock as a reference. We use `PyShark` [29], to capture packets on each cluster node. We capture IPv4, TCP, and UDP packets during our analysis.

Partition Injection Module. We use `iptables` [30] to disrupt/restore the connections between nodes in the network.

Testbed. For all our experiments, we use c220g1 [31] nodes from CloudLab [32]. The machines consist of two Intel Haswell E5-2630 CPUs with 8 cores each, 128GB of RAM, and a 10GBps network connection. We mention the cluster size for each experiment in its description.

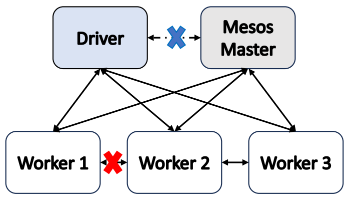


Fig. 3: Spark Failures with Mesos

V. Reproducing Documented Failures

To verify the functionality of our tool, we reproduce four failures previously reported in Apache Spark [33] and Apache Kafka [34]. We select these failures because they manifest in a complex setup involving multiple subsystems and have catastrophic effects. We test Spark with both its standalone scheduler [33] and Mesos [35]. We test Kafka paired with Zookeeper [36]. Slicify successfully detects the reported failures, and our analysis of the returned logs confirms that the reason for the failure matches the reason previously reported in the failure reports. Table I summarizes our verification efforts.

A. Failures in Spark

Apache Spark. Apache Spark is a popular data analytics system. Spark's system architecture consists of three main components: the application driver, cluster manager, and worker nodes. While Spark comes bundled with a cluster manager (Spark Standalone), it supports the integration of other resource managers such as Mesos [35].

The general workflow of a Spark application is as follows: the client submits a job to the cluster manager after which the manager starts a driver program. The driver works with the cluster manager to allocate executors on worker nodes and launch executors. The executors run tasks and may exchange intermediate results among themselves. The driver monitors task progress, collects results, and reports them.

Spark with Mesos. Apache Mesos is a cluster manager that handles resource management and allocation in large distributed environments. Mesos runs daemons (a.k.a. Mesos agents) on cluster nodes to monitor node resources. The Mesos master aggregates information about these available resources and offers them to Spark applications (drivers).

CASPR [12] reports two failures when running Spark with Mesos. The first occurs when there is a partial partition

System	Partition	Impact	Reported By	Detected by Slicify?
Spark	Worker - Worker	Halt	CASPR [12]	✓
Spark + Mesos	Worker - Worker	Halt	CASPR [12]	✓
	Driver - Mesos Master	Halt	CASPR [12]	✓
Kafka + ZK	ZK - Replicas	Halt	Kafka-8702 [13]	✓
Hazelcast - Maps	Member - Member	Operation Failure	-	✓
Hazelcast - Locks	Member - Member	Operation Failure	-	✓
Flink	Task Manager - Task Manager	Operation Failure	-	✓
ActiveMQ	Live Broker - Passive Broker	Inconsistent State	-	✓

TABLE I: Summary of partitions detected by Slicify on production systems

between two worker nodes, leading to a system pause. This occurs because the executors on these nodes cannot communicate and exchange data during the shuffle phase. The second failure occurs 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 Mesos master, the application does not start.

We use Slicify to test Spark with Mesos v1.11.0. Figure 3 shows the components of our cluster: Driver, Mesos master, and three worker nodes that host executor instances. We run the WordCount application used by CASPR [12].

Slicify detects both failures reported by CASPR [12]. The first failure occurs when Slicify injects a partition between two workers (shown in Figure 3 with a red marker). We get the same effect of complete system halt as reported by CASPR [12] and we verified the reason for the halt using the application logs. Slicify reports the second failure when it injects a partition between Mesos master and the driver (shown in Figure 3 with a blue marker). The driver never starts running as it does not receive any resources from the cluster manager.

Spark Standalone. In addition to Spark with Mesos, we used Slicify to test Spark with the standalone cluster manager. CASPR [12] reports that the failure between two workers affects Spark Standalone as well, causing a complete system halt. Slicify also manages to detect and report this failure.

B. Failures in Apache Kafka and Zookeeper

Apache Kafka is an open-source distributed message queuing system [34]. Kafka relies on Apache ZooKeeper [36] for coordination, metadata management, and leader election.

Kafka Architecture. Data in Kafka is organized into topics, consisting of multiple shards. Each shard can hold multiple messages. Kafka operates in a distributed cluster that consists of brokers, producers, and consumers. Producers publish messages to Kafka topics, and Kafka handles the distribution of these messages across shards within the topic. Brokers are responsible for storing data. Consumers subscribe to Kafka topics to retrieve records and consume messages.

Kafka maintains fault tolerance by replicating shards across multiple brokers. Each shard has one leader and multiple replicas, ensuring that the data remains available if a broker fails. If a leader fails, Kafka relies on ZooKeeper to detect the failure and elect a new leader for the shard.

Failure Details. Ticket #8702 [13] reports a failure within a cluster that deploys Kafka with ZooKeeper. They report a failure when a partial partition occurs between the Kafka shard leader and all replicas. In such cases, replicas pause operations as they can no longer contact the leader. The leader pauses its operations as it can no longer contact the majority of replicas. On the other hand, ZooKeeper does not detect a problem as all nodes appear healthy and thus does not elect a new leader. As a result, the system halts until the partition is healed.

Reproducing the failure. We use Slicify to test Kafka. Figure 4 shows our deployment. We deploy Kafka v2.3.0 with three replicated brokers i.e. each message is replicated on three nodes. We use Kafka's benchmarking tool to generate load onto the system. We use five topics distributed across cluster nodes and a set of producers and consumers. Each producer sends messages to a dedicated topic and each topic has one consumer. We deploy ZooKeeper within our cluster to monitor the cluster nodes.

Slicify reports a failure when it introduces a network partition between the shard leader and all other replicas (Figure 4). During this partition, all operations fail. The cluster remains unavailable until the partition heals because no new leader is elected. As ZooKeeper can reach the current leader, it assumes all operations are normal and does not elect a new leader. We examined the Kafka logs and verified this.

VI. Network Partitions with Hazelcast

In §VI - §VIII, we demonstrate Slicify's effectiveness by using it to test three popular distributed systems, uncovering new failures. Within this section, we examine Hazelcast [14], an in-memory distributed computation and storage system. Hazelcast provides distributed data structures such as maps, queues, and locks. Table I summarizes our experiments with

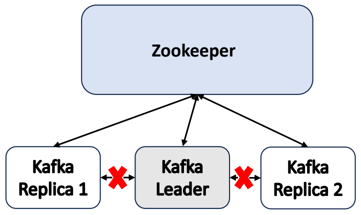


Fig. 4: Kafka Failure

Hazelcast. Slicify discovers two undocumented bugs, which we report by filing failure tickets [19].

A. Hazelcast's Design

Hazelcast's system architecture consists of members and clients (Figure 5). A member is the computational and storage unit of the Hazelcast cluster. Clients communicate with the cluster members.

Consistency Guarantees. Hazelcast offers different guarantees for different operations. Synchronization operations such as locks and countdown latches are always linearizable and are implemented using a Raft [37] replication protocol. Data structures such as maps and queues offer a lighter weight replication mechanism that can be configured to be strongly or eventually consistent.

Network Partition Tolerance. Hazelcast's design includes mechanisms for tolerating partial and complete network partitions [38], [39].To tolerate a partial partition, cluster members perform all-to-all heartbeating i.e. each member heartbeats all other 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 and builds a graph to represent cluster connectivity. It then analyzes the graph to identify the largest connected subset of cluster members (a.k.a maximum clique). In the event of a partial partition, all the members outside this clique are shut down. This effectively turns a partial partition into a complete partition. If the master member is outside the clique, it shuts down and the member with the next lowest ID takes its place.

B. Hazelcast's Partition Tolerance

Maps in Hazelcast are sharded and replicated among members [40]. Each shard stores a part of the map. Each shard can have multiple replicas; one acts as the primary replica while the others are backups. Note that the primary replica can be different from the master member.

Clients send read and write requests for an object to the primary replica of the shard holding the object. If the primary replica fails, one of the backup replicas takes over the primary role. Each member may be the primary replica for some shards and a backup for others. Hazelcast uses shard tables to help cluster members keep track of the primary replica for each shard. The shard table is created by the master member who periodically sends it to all members. Hazelcast can use either synchronous or asynchronous replication.

Hazelcast uses keys to distribute map entries across shards. When an entry is added to the map, Hazelcast assigns that entry to a specific shard based on the key's hash value. If a client wants to read / write a specific key, Hazelcast hashes the key and calculates the shard ID in which the data will be stored. The client randomly connects to one of the members listed in its configurations. The client finds the primary replica for the shard hosting the key within this member's shard table and sends the request to it.

Test Setup. We use Slicify to test maps and fenced locks. We deploy Hazelcast v5.3.1 on a four node cluster. Three

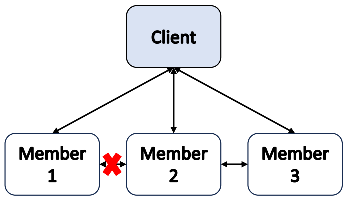


Fig. 5: Hazelcast Failures

nodes are members and one node is a client. We configure the cluster with synchronous replication and a replication level of 3. The client writes 1000 value objects of size 1 KB to the map. The chosen keys result in all shards being accessed. We enable the partial partition and the split-brain protection fault tolerance mechanisms. The minimum cluster size is set to 2.

Failure Discussion. Our tests discover a failure in the client access protocol, where a client is unable to access the system despite member availability to serve its request. The failure occurs when a partial partition impacts the communication between two cluster members. Figure 5 shows such a partial partition using a red marker.

In this scenario, the master member detects this partition. Following the partial partition tolerance mechanism, the master member splits the cluster into two sub clusters, with one side containing the majority of nodes that are fully connected. Nodes on the minority side pause operations and do not process client requests.

Clients select which member to send their requests to in random fashion. If the client is not impacted by the partition and can reach all members, it can send its request to a member on the minority side. This request will fail with an exception indicating that the failure occurred as the operation was sent to the minority side. Hazelcast fails these operations unnecessarily although the client may be connected to the majority side and can send its request to a member there. We discovered a similar failure when using fenced locks as well.

VII. Network Partitions with Flink

Apache Flink [15] is a popular stream processing engine. Table I summarizes our efforts to test Flink with Slicify. We uncover one undocumented bug which causes jobs to fail despite resources being available to execute them. We report this bug by filing a failure ticket [18].

A. Flink's Design

Figure 6 shows the Flink system architecture. Flink consists of a single JobManager and various TaskManagers. The JobManager receives jobs from clients, tracks job progress, and manages resources. A TaskManager runs on every worker node in the cluster and executes tasks. Flink quantizes resources as slots. A slot is the basic unit of resource scheduling and expressing resource requirements for tasks. The list of TaskManagers is provided during cluster deployment and is always traversed in the same order.

TaskManagers register their slots with the JobManager upon initialization. The JobManager stores these slots in a pool.

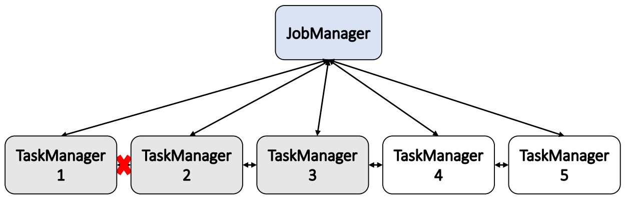


Fig. 6: Flink Failure

When a job is submitted, the JobManager queries the pool. If the pool has sufficient resources, they are assigned to the job, even if they are located across multiple TaskManagers. Once execution is complete, slots are released back to the pool. TaskManagers may communicate to exchange intermediate results during the shuffle stage.

B. Flink's Partition Tolerance

We use Slicify to test the execution of a WordCount application on Flink. We deploy Flink v1.17.1 on a cluster of six nodes; as shown in Figure 6. We enable checkpointing with a restart failover strategy, configured to attempt a restart three times with 10 seconds between attempts.

Each TaskManager has one slot, and the WordCount application submission has a level of parallelism of three. This forces the JobManager to run the tasks on three different TaskManagers. Shaded TaskManagers in Figure 6 represent the TaskManagers assigned to execute the job.

Failure Discussion. In the WordCount application, TaskManagers exchange data during the shuffle stage. If a partial partition between TaskManagers prevents the data exchange, this causes the job execution to fail and the allocated slots to be released. As restart failover is enabled, the JobManager restarts the job.

The JobManager checks the slot pool to find resources for the rerun and, due to the greedy scheduling strategy, it selects the first available slots. As the JobManager is unaware of the partition between TaskManagers, the slots impacted by the partial partition are still at the top of the list. Thus, the JobManager selects the same TaskManagers chosen earlier. The job rerun fails as well. Flink tries to run the job 3 times before giving up and raising an exception, terminating the application despite having available resources.

VIII. Network Partitions with ActiveMQ Artemis

ActiveMQ Artemis [16] is a message queuing system. Table I summarizes our efforts to test ActiveMQ Artemis with Slicify. We uncover one failure in Artemis' replication protocol which causes data inconsistency. We have reported the failure using the Artemis issue tracking system [17].

A. Artemis' Design

Figure 7 shows the ActiveMQ Artemis system architecture. Artemis consists of producers, brokers, and consumers. Brokers are the middleware, facilitating communication between message producers and consumers. Producers send messages to named addresses, which are stored in a queue before being served to consumers.

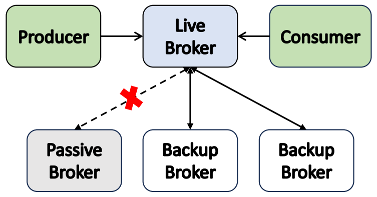


Fig. 7: ActiveMQ Artemis Failure

Replication. 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.

Fault Tolerance. One of the backup brokers is selected to execute in "passive mode". All messages are replicated to this passive backup. This replication is asynchronous and does not cause any blocking on client requests. To successfully become a passive broker, a broker needs to be connected to the majority of brokers in the cluster. Once a passive broker is successfully chosen, it begins monitoring the live broker via heartbeats. The live broker heartbeats all brokers in the cluster. If the passive broker misses a series of heartbeats from the live, it assumes the role of the live.

B. Artemis' Partition Tolerance

We use Slicify to test the produce and consume operations on ActiveMQ Artemis. We deploy Artemis v2.30.0 on a five node cluster. Figure 7 shows the cluster architecture.

Failure Discussion. If a partition cuts the communication between the live and passive brokers, the passive broker will miss the live's heartbeats and assume that it has crashed. It will then start acting as a live broker, serving requests. Thus, the system has two live brokers serving the same addresses. The two live brokers do not synchronize their data as they are unaware of each other. Consequently, consumers may not receive all messages sent to an address or may receive different sets of messages when subscribed to the same address. The issue persists even after the network partition is healed.

IX. Related Work

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. While other areas have received significant attention [2], [3], fault injection in network testing is more limited. Some fault injection efforts target networks [41] and network protocols [42], but do not support network partition testing. NEAT [9] supports network partition testing but requires manual intervention. Loki [43] and CoFi [44] are not application agnostic and do not test for partial partitions.

Network Tools. Modern networking tools fall into various categories such as traffic generation [45], packet capture [27], and network monitoring [46]. Our previous work, CASPR

[12] focuses on building a partition-aware scheduler. These are orthogonal to Slicify.

X. Conclusion

We present Slicify, an automated and application-agnostic tool to test the network partition resilience of distributed systems, Slicify tests distributed systems by injecting partitions between system components communicating during unit tests. We verify Slicify's capabilities by reproducing failures in production systems previously detected only using human intervention. In addition, we demonstrate Slicify's effectiveness by detecting new failures in three production systems.

References

[1] 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.

[2] Karthik Pattabiraman, Nithin Nakka, and Ravishankar Iyer. Symplfied: Symbolic program-level fault injection and error detection framework. volume 62, pages 472 – 481, July 2008.

[3] 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), October 2018.

[4] 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, August 2010.

[5] CloudFlare Blog. 2020. A Byzantine failure in the real world. https://blog.cloudflare.com/a-byzantine-failure-in-the-real-world/, Nov 2020.

[6] Google Cloud. 2019. Google Cloud Networking Incident #18003. https://status.cloud.google.com/incident/cloud-networking/18003, Feb 2019.

[7] Andrey Falko. 2019. Lyft Engineering: Operating Apache Kafka Clusters 24/7 Without a Global Ops Team, Sep 2019.

[8] Datadog. 2013. Learning from AWS Failure. https://www.datadoghq.com/blog/gray-aws-failures/, Oct 2013.

[9] 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.

[10] 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.

[11] 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.*, December 2022.

[12] Sara Qunaibi, Sreeharsha Udayashankar, and Samer Al-Kiswany. CASPR: Connectivity-aware scheduling for partition resilience. *SRDS 2023*, Sep 2023.

[13] 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.

[14] Hazelcast Documentation. https://docs.hazelcast.com/hazelcast/5.3/.

[15] Apache Flink Documentation. https://nightlies.apache.org/flink/flink-docs-stable/.

[16] ActiveMQ Artemis Documentation. https://activemq.apache.org/components/artemis/documentation/.

[17] 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.

[18] 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.

[19] Hazelcast-26208: Hazelcast fails some client requests when there is a split in the cluster. https://github.com/hazelcast/hazelcast/issues/26208.

[20] Slicify github repository. https://github.com/UWASL/slicify.

[21] 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.

[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] Tony Mills. Bnx2 cards intermittantly going offline.

[24] Paul DuBois. *MySQL*. Addison-Wesley, 2013.

[25] Josiah Carlson. *Redis in action*. Simon and Schuster, 2013.

[26] David L Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10):1482–1493, 1991.

[27] Wireshark User's Guide. https://www.wireshark.org/docs/wsug_html_chunked/.

[28] Amina Elbatoul Dinar, Boualem Merabet, and Samir Ghouali. Ntp server clock adjustment with chrony. In *Applications of Internet of Things: Proceedings of ICCCIOT 2020*, pages 177–185. Springer, 2020.

[29] Pyshark Documentation. https://github.com/KimiNewt/pyshark/.

[30] iptables(8) - Linux man page. https://linux.die.net/man/8/iptables.

[31] The CloudLab Team. Cloudlab hardware. https://docs.cloudlab.us/hardware.html.

[32] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1–14, Renton, WA, July 2019. USENIX Association.

[33] 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.

[34] Apache Kafka. https://kafka.apache.org.

[35] Roger Ignazio. *Mesos fundamentals*, page 58–62. Manning, 2018.

[36] Apache ZooKeeper. https://zookeeper.apache.org/.

[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. https://docs.hazelcast.com/hazelcast/5.3/network-partitioning/partial-network-partitions.

[39] Split-Brain Protection. https://docs.hazelcast.com/hazelcast/5.3/network-partitioning/split-brain-protection.

[40] Distributed Map. https://docs.hazelcast.com/hazelcast/5.3/data-structures/map.

[41] David T Stott, Greg Ries, Mei-Chen Hsueh, and Ravishankar K Iyer. Dependability analysis of a high-speed network using software-implemented fault injection and simulated fault injection. *IEEE Transactions on Computers*, 47(1):108–119, 1998.

[42] Pradipta De, Anindya Neogi, and T-C Chiueh. Virtualwire: A fault injection and analysis tool for network protocols. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.*, pages 214–221. IEEE, 2003.

[43] R.M. Lefever, M. Cukier, and W.H. Sanders. An experimental evaluation of correlated network partitions in the coda distributed file system. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pages 273–282, 2003.

[44] Haicheng Chen, Wensheng Dou, Dong Wang, and Feng Qin. Cofi: Consistency-guided fault injection for cloud systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 536–547, 2020.

[45] Gianni Antichi, Muhammad Shahbaz, Yilong Geng, Noa Zilberman, Adam Covington, Marc Bruyere, Nick McKeown, Nick Feamster, Bob Felderman, Michaela Blott, et al. Osnt: Open source network tester. *IEEE Network*, 28(5):6–12, 2014.

[46] Wolfgang Barth. *Nagios: System and network monitoring*. No Starch Press, 2008.